Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros Informáticos

Formal Verification of Temporal Properties for Parametrized Concurrent Programs and Concurrent Data Structures

PhD Thesis

Alejandro Sánchez, M.Sc.

Departamento de Lenguajes,
Sistemas Informáticos e Ingeniería del Software

Escuela Técnica Superior de Ingenieros Informáticos

Formal Verification of Temporal Properties for Parametrized Concurrent Programs and Concurrent Data Structures

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science

Author: Alejandro Sánchez, M.Sc.
Adviser: César Sánchez, Ph.D.

September 2015

In loving memory of my grandma Elena,
my early mentor in this path of knowledge.

To my parents, Betty and Carlos, and my sister Silvina,
constant and unremitting companions through life.

To my loving and marvelous wife Olivera,
the ever brighting star leading my way.

Defense Committee

UNIVERSIDAD POLITECNICA DE MADRID

Tribunal nombrado por el Magfco. y Excmo. Sr. Rector de la Universidad Politécnica de Madrid, el día 15 de Julio de 2015.

Presidente:	Narciso Martí Oliet
Vocal:	Volker Stolz
Vocal:	Pierre Ganty
Vocal:	Enric Rodríguez Carbonell
Secretario:	Manuel Carro Liñares
Suplente:	Viktor Kuncak
Suplente:	Bernd Finkbeiner

Realizado el acto de defensa y lectura de la Tesis el día 10 de Septiembre de 2015 en la Escuela Técnica Superior de Ingenieros Informáticos.

EL PRESIDENTE LOS VOCALES

EL SECRETARIO

Resumen

Los tipos de datos concurrentes son implementaciones concurrentes de las abstracciones de datos clásicas, con la diferencia de que han sido específicamente diseñados para aprovechar el gran paralelismo disponible en las modernas arquitecturas multiprocesador y multinúcleo. La correcta manipulación de los tipos de datos concurrentes resulta esencial para demostrar la completa corrección de los sistemas de *software* que los utilizan. Una de las mayores dificultades a la hora de diseñar y verificar tipos de datos concurrentes surge de la necesidad de tener que razonar acerca de un número arbitrario de procesos que invocan estos tipos de datos de manera concurrente. Esto requiere considerar sistemas parametrizados. En este trabajo estudiamos la verificación formal de propiedades temporales de sistemas concurrentes parametrizados, poniendo especial énfasis en programas que manipulan estructuras de datos concurrentes.

La principal dificultad a la hora de razonar acerca de sistemas concurrentes parametrizados proviene de la interacción entre el gran nivel de concurrencia que éstos poseen y la necesidad de razonar al mismo tiempo acerca de la memoria dinámica. La verificación de sistemas parametrizados resulta en sí un problema desafiante debido a que requiere razonar acerca de estructuras de datos complejas que son accedidas y modificadas por un numero ilimitado de procesos que manipulan de manera simultánea el contenido de la memoria dinámica empleando métodos de sincronización poco estructurados.

En este trabajo, presentamos un marco formal basado en métodos deductivos capaz de ocuparse de la verificación de propiedades de *safety* y *liveness* de sistemas concurrentes parametrizados que manejan estructuras de datos complejas. Nuestro marco formal incluye reglas de prueba y técnicas especialmente adaptadas para sistemas parametrizados, las cuales trabajan en colaboración con procedimientos de decisión especialmente diseñados para analizar complejas estructuras de datos concurrentes. Un aspecto novedoso de nuestro marco formal es que efectúa una clara diferenciación entre el análisis del flujo de control del programa y el análisis de los datos que se manejan.

El flujo de control del programa se analiza utilizando reglas de prueba y técnicas de verificación deductivas especialmente diseñadas para lidiar con sistemas parametrizados. Comenzando a partir de un programa concurrente y la especificación de una propiedad temporal, nuestras técnicas deductivas son capaces de generar un conjunto finito de condiciones de verificación cuya validez implican la satisfacción de dicha especificación temporal por parte de cualquier sistema, sin importar el número de procesos que formen parte del sistema.

Las condiciones de verificación generadas se corresponden con los datos manipulados. Estudiamos el diseño de procedimientos de decisión especializados capaces de lidiar con estas condiciones de verificación de manera completamente automática. Investigamos teorías decidibles capaces de describir propiedades de tipos de datos complejos que manipulan punteros, tales como implementaciones imperativas de pilas, colas, listas y *skiplists*. Para cada una de estas teorías presentamos un procedimiento de decisión y una implementación práctica construida

sobre *SMT solvers*. Estos procedimientos de decisión son finalmente utilizados para verificar de manera automática las condiciones de verificación generadas por nuestras técnicas de verificación parametrizada.

Para concluir, demostramos como utilizando nuestro marco formal es posible probar no solo propiedades de *safety* sino además de *liveness* en algunas versiones de protocolos de exclusión mutua y programas que manipulan estructuras de datos concurrentes. El enfoque que presentamos en este trabajo resulta ser muy general y puede ser aplicado para verificar un amplio rango de tipos de datos concurrentes similares.

Palabras clave:

Concurrencia, Sistemas Parametrizados, Verificación Temporal, Safety, Liveness, Procedimientos de Decisión, Estructuras de Datos

Abstract

Concurrent data types are concurrent implementations of classical data abstractions, specifically designed to exploit the great deal of parallelism available in modern multiprocessor and multi-core architectures. The correct manipulation of concurrent data types is essential for the overall correctness of the software system built using them. A major difficulty in designing and verifying concurrent data types arises by the need to reason about any number of threads invoking the data type simultaneously, which requires considering parametrized systems. In this work we study the formal verification of temporal properties of parametrized concurrent systems, with a special focus on programs that manipulate concurrent data structures.

The main difficulty to reason about concurrent parametrized systems comes from the combination of their inherently high concurrency and the manipulation of dynamic memory. This parametrized verification problem is very challenging, because it requires to reason about complex concurrent data structures being accessed and modified by threads which simultaneously manipulate the heap using unstructured synchronization methods.

In this work, we present a formal framework based on deductive methods which is capable of dealing with the verification of safety and liveness properties of concurrent parametrized systems that manipulate complex data structures. Our framework includes special proof rules and techniques adapted for parametrized systems which work in collaboration with specialized decision procedures for complex data structures. A novel aspect of our framework is that it cleanly differentiates the analysis of the program control flow from the analysis of the data being manipulated.

The program control flow is analyzed using deductive proof rules and verification techniques specifically designed for coping with parametrized systems. Starting from a concurrent program and a temporal specification, our techniques generate a finite collection of verification conditions whose validity entails the satisfaction of the temporal specification by any client system, in spite of the number of threads.

The verification conditions correspond to the data manipulation. We study the design of specialized decision procedures to deal with these verification conditions fully automatically. We investigate decidable theories capable of describing rich properties of complex pointer based data types such as stacks, queues, lists and skiplists. For each of these theories we present a decision procedure, and its practical implementation on top of existing SMT solvers. These decision procedures are ultimately used for automatically verifying the verification conditions generated by our specialized parametrized verification techniques.

Finally, we show how using our framework it is possible to prove not only safety but also liveness properties of concurrent versions of some mutual exclusion protocols and programs that manipulate concurrent data structures. The approach we present in this work is very general, and can be applied to verify a wide range of similar concurrent data types.

Keywords:

Concurrency, Parametrized Systems, Temporal Verification, Safety, Liveness, Decision Procedures, Data Structures

Disclaimer

This dissertation builds on several published works that I have co-authored.

CONFERENCE ARTICLES:

- **Decision Procedures for the Temporal Verification of Concurrent Lists**
  Co-authored with César Sánchez
  In the 12th International Conference on Formal Engineering Methods (ICFEM'10)
- **A Theory of Skiplists with Applications to the Verification of Concurrent Datatypes**
  Co-authored with César Sánchez
  In the 3rd International Symposium on NASA Formal Methods (NFM'11)
- **Invariant Generation for Parametrized Systems using Self-reflection**
  Co-authored with Sriram Sankaranarayanan, César Sánchez and Bor-Yuh Evan Chang
  In the 19th International Static Analysis Symposium (SAS'12)
- **LEAP: A Tool for the Parametrized Verification of Concurrent Datatypes**
  Co-authored with César Sánchez
  In the 26th International Conference on Computer Aided Verification (CAV'14)
- **Parametrized Verification Diagrams**
  Co-authored with César Sánchez
  In the 21st International Symposium on Temporal Representation and Reasoning (TIME'14)
- **Formal Verification of Skiplists with Arbitrary Many Levels**
  Co-authored with César Sánchez
  In the 12th International Symposium on Automated Technology for Verification and Analysis (ATVA'14)

JOURNAL ARTICLES:

- **Parametrized Invariance for Infinite State Processes**
  Co-authored with César Sánchez
  In Acta Informatica, volume 52, issue 6 (September 2015).

Acknowledgments

Being Spanish my mother tongue language, I would like to take the freedom of writing this section in such a beautiful language, as all words below comes from the bottom of my heart.

Como todos sabrán, el contenido de este trabajo no es sólo mérito propio, sino de un montón de personas que me han acompañado a lo largo del camino. Algunas durante mi etapa como alumno de doctorado, otras durante toda mi vida. La lista es larga y mi memoria frágil. Por favor, si no se encuentran aquí debajo, sientan la completa libertad de escribirme a mi nueva dirección de correo, **dr.alesanz@gmail.com**, y los añadiré en la próxima edición 😋 .

Antes que nada, quisiera agradecer a mi familia. Mis padres, **Beatriz Sánchez** y **Carlos Sánchez**, quienes desde pequeño me inculcaron lo importante que era tener una buena educación, me enseñaron los valores básicos de la vida, me acompañaron en los momentos felices, me animaron en los momentos difíciles, siempre creyeron en mí y nunca dejaron de estar a mi lado. También a mi hermana, **Silvina Sánchez**, una de las personas mas dulces que conozco, simpática, siempre sonriente, compañera de aventuras durante mi infancia, enorme apoyo durante la madurez y siempre fiel seguidora de su hermano.

La vida te va dando sorpresas y regalos. Uno de los mas preciosos y bellos que me ha dado es mi esposa, **Olivera Lazarovska Sánchez**. A mis ojos, la persona mas noble, hermosa, cariñosa, fuerte y decidida del mundo. Incansable compañera de viajes, inseparable colega de aventuras, fiel confidente de mis preocupaciones y devota seguidora de su marido. Gracias por el apoyo, respaldo, paciencia y comprensión durante todos estos años.

No puedo dejar pasar este momento sin agradecer a quienes fueron uno de mis primeros mentores cuando era pequeñito: mis abuelos. Siempre tendré presente todo lo que aprendí con ellos. Mi abuela **Edith Guidobono** y mi abuelo **Pedro Sánchez**, quien siendo pequeño me confesó que uno de sus sueños era que yo un día fuese un doctor que salvara vidas. Bueno, en realidad mi aversión hacia la sangre no lo permitió, pero en cierta forma hoy lo estoy logrando convirtiéndome en otra especie de doctor 🙂 . Mi abuela **Elena Gattoni**, la primera maestra de la vida que tuve. Con ella aprendí muchos de los valores que me han marcado a lo largo de la vida. Ella fue quien me enseñó las primeras letras y me inculcó el hábito de la lectura, abriendo en ese mismo momento infinitos horizontes de aprendizaje y descubrimiento en mí. Humilde, sencilla, tenaz, perseverante, de espíritu sólido como una roca y a la vez con un corazón tan sincero, amoroso y dulce como ninguno. Le estaré eternamente agradecido por todo. Y por supuesto que no puedo dejar a mencionar a **Juan Gattoni**, tío abuelo segundo de sangre, el mejor tío de corazón. Un enamorado del basquet, asiduo seguidor de Atenas, mi primer guía camino a la escuela, inseparable compañero en el día a día y maravillosa persona. Finalmente, no puedo dejar de pasar la oportunidad de agradecer a **Javier Sánchez**, primo de sangre, casi un hermano, compañero de numerosas aventuras a lo largo de la vida, inmensurable amigo y sin dudas una de

las personas que me influenció a estudiar esta hermosa carrera gracias a su querida *Commodore 64* que aún conservo.

Quiero también agradecer a quienes han sido mis compañeros éstos últimos años. En primer lugar, quisiera agradecer a **Julián Samborski-Forlese**, colega de despacho, compañero de piso, кумот на мојата свадба e increíble amigo. Durante todo el tiempo de mi doctorado ha estado a mi lado para ayudarme, aconsejarme y colaborar cuando lo he necesitado. Quisiera también agradecer a **Carolina Dania**, aunque mantiendo firme mis convicciones no la voy a citar , y a **Javier Valdazo** por haber sido un gran amigo, excelente compañero durante varios años de mi carrera como estudiante, fácil en el ping-pong e imposible de ganarle al FIFA.

Académicamente hablando, quisiera agradecer a quien ha sido mi supervisor, **César Sánchez**, por haber sido el mejor consejero, guía y compañero de investigación que podría imaginar. A pesar de su corta edad me ha demostrado ser una de las personas más sabias, determinadas e inteligentes que he conocido. A su lado he aprendido muchísimo, no solo en el campo académico, sino también sobre enseñanzas de la vida. Chutzpah!

Quisiera al mismo tiempo agradecer a **IMDEA Software** por haberme brindado los medios que han hecho todo esto posible. Mas importante aún, a las personas que han pasado por allí y que hicieron el día a día una experiencia maravillosa: **César Kunz**, **Federico Olmedo**, **Juan Manuel Crespo**, **Teresa Trigo**, **Miriam García**, **Joaquín Arias**, **Germán Delbianco**, **Miguel Ángel García**, **Manuel Clavel**, **Paola Huerta**, **Tania Rodríguez**, **Andrea Iannetta**, **Begoña Moreno**, **María Alcaraz**, **Inés Huertas**, **Juan Céspedes**, **Roberto Lumbreras** y **Gabriel Trujillo** entre otros. Desde luego, gracias **Gilles Barthe** por haber confiado en mí y haberme traido a IMDEA.

Quisiera también aprovechar para agradecer a **Narciso Martí Oliet** por ser presidente de mi comité de defensa, además de un extraordinario consejero y guía durante mi etapa de máster en la Universidad Complutense de Madrid. Agradezco también a **Manuel Carro** por ser secretario de mi comité de defensa y aconsejarme tan bien durante la última etapa de mi doctorado y fundamentalmente en los pasos finales de organización de mi defensa. Una defensa de tesis no sería posible sin un comité, por ello quiero agradecer especialmente a las personas que aceptaron formar parte de mi comité de defensa, por su compromiso, consejos, ideas, aportes y colaboración. Ellos son: **Viktor Kuncak**, **Bernd Finkbeiner**, **Volker Stolz**, **Enric Rodríguez Carbonell** y **Pierre Ganty**. De igual modo, muchas gracias a **Julio Mariño** por formar parte del comité de mi pre-lectura con tan poco tiempo de antelación.

Finalmente, quisiera expresar mi agradecimiento a mi familia política, quien durante toda mi estancia en Macedonia me hacen sentir como en casa. Muchas gracias **Venera Lazarova**, **Ognjancho Lazarov**, **Nikola Lazarevski**, **Maja Lazarevska**, **Vera Lazarova** y muy, pero muy especialmente a **Leo Lazarevski** y **Mario Lazarevski** por tantas tardes de entretenimiento, juegos y alegrías.

Contents

Defense Committee **v**

Resumen **vii**

Abstract **ix**

Disclaimer **xi**

Acknowledgments **xiii**

Contents **xx**

List of Figures **xxiv**

List of Tables **xxvi**

1 Introduction **1**

1.1 Related Work 5

1.1.1 Logics for Concurrent Heap Reasoning 5

1.1.2 Parametrized Verification 6

1.1.3 Automatic Parametrized Invariant Generation 8

1.1.4 Verification Tools 9

1.1.5 Decision Procedures 10

1.2 Structure of this Work 11

2 Preliminaries **17**

2.1 Simplified Programming Language (SPL) 17

2.1.1 SPL Statements 19

2.1.2 Semantics of SPL Statements 22

2.1.3 Ghost Code 26

2.2 Concurrent Data Structures 27

2.2.1 Performance Concerns 29

2.2.2 Synchronization Techniques 29

2.2.3 Verification Techniques 32

2.2.4 Synchronization Elements 34

2.3 Signatures and Theories 35

2.4 Model of Computation 38

2.4.1 Linear Temporal Logic 38

2.4.2 Non-Parametrized Fair Transition Systems 39

2.4.3 Parametrized Concurrent Programs 41

2.4.4 Parametrized Fair Transition Systems 43

2.4.5 Parametrized Formulas 45

2.4.6 Parametrized Temporal Verification 46

I Parametrized Verification Techniques **51**

3 Parametrized Invariance **53**

3.1 The Need of Parametrized Safety Proof Rules 54

3.2 Parametrized Proof Rules 57

3.2.1 The Basic Parametrized Invariance Rule: BP-INV 57

3.2.2 The Parametrized Invariance Rule: P-INV 58

3.2.3 The General Strengthening Parametrized Invariance Rule: SP-INV 63

3.2.4 The Parametrized Graph Proof Rule: G-INV 65

3.3 Summary 68

4 Parametrized Verification Diagrams **71**

4.1 GVD vs. PVD: The Need of Parametrized Verification Diagrams 72

4.2 Parametrized Verification Diagrams 75

4.2.1 Definition of PVD 76

4.3 Verification Conditions for PVD 79
4.4 Summary 85

5 Invariant Generation for Parametrized Systems 87
5.1 Self-Reflection 88
5.2 Reflective Abstractions and Inductive Invariants 91
5.2.1 Reflective Abstractions 93
5.3 Reflective Abstract Interpretation 97
5.3.1 Abstract Interpretation using Reflection 99
5.3.2 Interference Abstraction versus Reflective Abstraction 100
5.3.3 The Effect of Widening 102
5.4 Summary 102

II Decision Procedures 103

6 TL3: A Decidable Theory of Concurrent Lists 105
6.1 Concurrent Data Types that Manipulate Lists 106
6.1.1 A Concurrent Lock-coupling List 106
6.1.2 An Unbounded Queue 111
6.1.3 An Unbounded Lock-free Stack 113
6.1.4 An Unbounded Lock-free Queue 115
6.2 TL3: A Theory of Concurrent Single-Linked Lists 119
6.2.1 Sorts 119
6.2.2 Signature 120
6.2.3 Interpretations 122
6.2.4 Satisfiability of TL3 125
6.3 Decidability of TL3 129
6.3.1 Auxiliary Functions for Model Transformation 130
6.3.2 A Bounded Model Theorem for TL3 132
6.4 Summary 141

7 TSL_K: Decidable Theories of Skiplists of Bounded Height 143
7.1 Skiplists 144
7.2 TSL_K: A Family of Theories for Skiplists of Bounded Height 154
7.2.1 Sorts 155

7.2.2 Signature 156

7.2.3 Interpretations 158

7.2.4 Satisfiability of TSL_K 162

7.3 Decidability of TSL_K 169

7.3.1 Auxiliary Functions for Model Transformation 169

7.3.2 A Bounded Model Theorem for TSL_K 172

7.4 Summary 178

8 TSL: A Decidable Theory for Skiplists with Unbounded Levels 181

8.1 A Skiplist with an Unbounded Number of Levels 182

8.2 TSL: A Theory for Skiplists with Unbounded Levels 189

8.2.1 Sorts 190

8.2.2 Signature 191

8.2.3 Interpretations 193

8.3 Normalization of TSL 195

8.4 Decidability of TSL 200

8.4.1 STEP 1: Sanitization 201

8.4.2 STEP 2: Order Arrangements 202

8.4.3 STEP 3: Split 202

8.4.4 STEP 4: Presburger Constraints 214

8.4.5 STEP 5: Deciding Satisfiability of Formulas Without Constants 214

8.5 Summary 219

III Implemenentation and Experimental Results 221

9 LEAP: A Verification Tool for Parametrized Datatypes 223

9.1 General Overview 223

9.2 Features 228

9.2.1 Programs 228

9.2.2 Specifications 229

9.2.3 Domain Cut-offs 231

9.2.4 Tactics 234

9.3 Verification using LEAP 238

9.3.1 Safety Properties 238

9.3.2 Liveness Properties 241
9.3.3 Decision Procedures 245
9.4 Summary 245

10 Experimental Results **247**
10.1 Verification of Safety Properties in Numeric Programs 248
10.2 Verification of Safety Properties using TL3 249
10.2.1 Lock-Coupling Lists 250
10.2.2 Unbounded Lock-based Queue 253
10.2.3 Lock-free Stacks 255
10.2.4 Lock-free Queues 256
10.3 Verification of Safety Properties using TSL_K 258
10.4 Verification of Safety Properties using TSL 260
10.5 Verification of Liveness Properties using PVD 263
10.5.1 Mutual Exclusion Protocol 263
10.5.2 Lock-coupling Lists 266
10.6 Parametrized Invariant Generation 273
10.6.1 SIMPLEBARRIER: A Simple Barrier Algorithm 274
10.6.2 CENTRALBARRIER: A Centralized Barrier Synchronization Algorithm . . . 275
10.6.3 WORKSTEAL: An Array Processing Work Stealing Algorithm 276
10.6.4 PHILOSOPHERS: Dining Philosophers With Bounded Resources 276
10.6.5 ROBOTS: Robot Swarm 278
10.6.6 Results and Observations 278

11 Conclusion **283**
11.1 Summary 283
11.1.1 Answered Questions 285
11.2 Open Questions and Future Work 286
11.2.1 Parametrized Invariance 286
11.2.2 Parametrized Verification Diagrams 287
11.2.3 Parametrized Invariant Generation 287
11.2.4 Decision Procedures 287
11.2.5 LEAP 288

Bibliography **289**

IV Appendices 311

Appendix A Checking that a PVD Satisfies a Temporal Property 313

A.1 The Intended Meaning of $\mathcal{F}$ 313

A.2 Edge Streett Automaton on Words 314

A.3 From ESNW into NBW 314

Appendix B LEAP 319

B.1 LEAP Command Line Options 319

B.2 LEAP Syntax 322

B.2.1 LEAP Programming Language 322

B.2.2 LEAP Expressions 329

B.2.3 LEAP Proof Graphs and PVD Support 334

B.2.4 LEAP Parametrized Verification Diagrams 336

List of Figures

1.1 Structure of this work. 12

2.1 A simple example showing the structure of an SPL program. 19
2.2 Two implementations of a ticket based mutual exclusion protocol. 21
2.3 Program INTMUTEX updated with ghost code. 27
2.4 Possible implementation of a *shared variable*. 28
2.5 Description of a CAS operation. 31
2.6 Non-blocking shared counter implementation using CAS. 32

3.1 Rules B-INV and INV for non-parametrized systems. 54
3.2 Declaration of procedure TWO. 55
3.3 The basic parametrized invariance rule BP-INV. 57
3.4 Declaration of program POSITIVE. 58
3.5 The parametrized invariance rule P-INV. 60
3.6 The general strengthening parametrized invariance rule SP-INV for proving relative inductive parametrized invariants. 64
3.7 The graph parametrized invariance rule G-INV. 65
3.8 Proof graph showing dependencies between the invariants candidates for SETMUTEX. 68

4.1 Schematic representation of the use of a GVD $\mathcal{D}$ to prove that program P satisfies the temporal property φ. 72
4.2 Simplified GVD for proving eventually_critical_T1 in a system with two threads. . 74
4.3 Simplified GVD for proving eventually_critical_T1 in a system with three threads. 75

4.4 A PVD for verifying property of a system with an unbounded number of threads. 77
4.5 SETMUTEX modified for proving eventually_critical(k). 83
4.6 PVD for the proof that SETMUTEX satisfies eventually_critical(k). 84

5.1 A reflective abstraction to infer 2-indexed invariants of a parametrized system. . 88
5.2 WORKSTEAL: A parametrized array processing program. 88
5.3 Materialized and MIRROR thread for a 1-index invariant in the WORKSTEAL program. 90
5.4 COUNTINCRGLOBAL: a simple global counter example. 101

6.1 SEARCH procedure for concurrent lock-coupling lists. 108
6.2 INSERT procedure for concurrent lock-coupling lists. 109
6.3 REMOVE procedure for concurrent lock-coupling lists. 110
6.4 Most general client procedure MGCLIST for concurrent lock-coupling lists. . . . 110
6.5 A coarse-grain locking unbounded queue implementation. 112
6.6 An unbounded lock-free stack implementation. 113
6.7 Sequence of states reached changes suffered by a lock-free stack during the execution of CAS operation at procedure PUSH: (a) the state of the stack when reaching line 5; (b) the condition of the CAS operation holds; (c) after successful execution of CAS operation; (d) the condition of the CAS operation does not hold; (e) after unsuccessful execution of CAS operation. 114
6.8 An unbounded lock-free queue implementation. 116
6.9 Sequence of steps for inserting a new element into a lock-free queue implementation.117
6.10 Potential problem in the lock-free queue implementation if DEQUEUE does not assist ENQUEUE in advancing $tail$ pointer. 118

7.1 A skiplist with 4 levels. 144
7.2 Skiplist traversal while searching for value 20. 145
7.3 SEARCH procedure for concurrent skiplists of K levels. 147
7.4 Snapshots of the execution of process SEARCH over a concurrent skiplist when descending a level. 149
7.5 INSERT procedure for concurrent skiplists of K levels. 150
7.6 Example of inserting a node of 3 levels and value 15 into a skiplist. (a) shows that original skiplist, (b) shows the state before the node is connected and (c) shows the state after the node is connected. We color in gray the pointers that are modified. 151

7.7 REMOVE procedure for concurrent skiplists of K levels. 153
7.8 Most general client procedure MGCSKIPLISTK for concurrent skiplist of K levels. 154

8.1 An initialized unbounded skiplist. 183
8.2 An example of an unbounded skiplist. 184
8.3 SEARCH procedure for an unbounded skiplists. 185
8.4 Procedure INSERT for unbounded skiplists. 186
8.5 REMOVE procedure for an unbounded skiplists. 188
8.6 Most general client procedure MGCSKIPLIST for unbounded skiplists. 189
8.7 Steps of the decision procedure for the satisfiability of TSL formulas. 201
8.8 A split of φ obtained after STEP 1 into φ^{PA} and φ^{NC}. 201
8.9 Pumping a model $\mathcal{N}$ of φ^{NC} (on the left of (a), (b), (c) and (d)) to a model $\mathcal{M}$ of φ (on the right of (a), (b), (c) and (d)) is allowed thanks to the fresh level l_{new}. In (b) the truth value of $C = D\{l_1 \leftarrow e\}$ is not preserved. In (c) $C = D\{l_2 \leftarrow e\}$ is not preserved. In (d) all predicates are preserved. 210

9.1 Schematic design of LEAP. 224
9.2 Example of a LEAP input program. 226
9.3 Example of LEAP atomic statement. 229
9.4 Example of a LEAP specification. 230
9.5 Example of a LEAP specification. 230
9.6 Example of a combined LEAP specification. 231
9.7 Example of a LEAP verification condition representation. 234
9.8 Example of a LEAP proof graph. 239
9.9 Full example of PVD in LEAP format. 243

10.1 Modifications needed to apply over the lock-coupling implementation of Section 6.1 in order to verify some functional properties. 251
10.2 Modifications needed to apply over the lock-coupling implementation of Section 6.1 in order to verify funSearch properties. 252
10.3 Modifications needed to apply over the lock-based unbounded queue implementation in order to verify unbQueueInc property. 254
10.4 Modifications needed to apply over the lock-free stack implementation in order to verify lfStackInc. 255

10.5 Modifications needed to apply over the lock-free queue implementation in order to verify lfQueueInc. 257
10.6 Functional specifications for SEARCH, INSERT and REMOVE 261
10.7 LEAP input representation for the PVD showing that SETMUTEX satisfies property eventually_critical (k). 264
10.8 NuSMV input representation for checking condition (ModelCheck) for the PVD that shows that program SETMUTEX satisfies property eventually_critical(k). . . . 265
10.9 Concurrent lock-coupling single-linked lists implementation with ghost code for the verification of property eventually_insert(k). 267
10.10 PVD for verifying that concurrent lock-coupling lists satisfy eventually_insert(k). . 269
10.11 NuSMV input representation for checking condition (ModelCheck) for the PVD that shows that concurrent lock-coupling lists satisfy property eventually_insert(k). 272
10.12 SIMPLEBARRIER: a simple synchronization barrier program. 274
10.13 CENTRALBARRIER: a centralized synchronization barrier program. 275
10.14 PHILOSOPHERS: dining philosophers with a bounded number of resources. 277
10.15 ROBOTS_2_2: a set of robots moving around a 2×2 grid. 279

List of Tables

9.1 Example of support generated using tactic full. 235

10.1 Verification conditions (VCs) proved and executing time using a numeric decision procedure for verifying safety properties of a mutual exclusion protocol. 249

10.2 Verification conditions (VCs) proved and executing time using TL3 decision procedure for verifying safety properties of a concurrent lock-coupling list. . . . 253

10.3 Verification conditions (VCs) proved and executing time using TL3 decision procedure for verifying safety properties of an unbounded lock-based queue. . . 254

10.4 Verification conditions (VCs) proved and executing time using TL3 decision procedure for verifying safety properties of a lock-free stack. 256

10.5 Verification conditions (VCs) proved and executing time using TL3 decision procedure for verifying safety properties of a lock-free queue. 258

10.6 Verification conditions (VCs) proved and executing time using $\mathsf{TSL_K}$ decision procedure for verifying safety properties for bounded skiplists of 1, 2, 3, 4 and 5 levels. 259

10.7 Number of queries for the verification of unbounded skiplists. "–" means no calls to the decision procedure were required. All TSL queries were ultimately decomposed into $\mathsf{TSL_K}$ for $\mathsf{K} \leq 5$. 261

10.8 Running times for the verification of safety properties over unbounded skiplists. 262

10.9 Running times for the verification of property eventually_critical(k) in program SetMutex. 266

10.10 Running times for the verification of property eventually_insert(k) for a concurrent lock-coupling list. 273

10.11 Timing and precision results for Lazy, Eager, Eager+ and Interference abstract interpretations. Legend: **ID:** benchmark identifier, **Dom:** abstract domains, **I:** intervals, **O:** octagons, **P:** polyhedra, **Prps**: total number of properties to be proven, **Time**: seconds, **Prp**: number of properties proved, **Wid**: number of widening iterations. 280

10.12 Comparison of invariants of various schemes. 281

1 Introduction

> *It's the job that's never started as takes longest to finish.*
>
> Samwise Gamgee
> (The Fellowship of the Ring)

This work studies the formal verification of parametrized temporal properties of concurrent systems which are composed by an unbounded number of threads. In particular, we are interested in programs that dynamically modify the heap, manipulating complex pointer-based data structures. We propose the construction of a general verification framework which tackles the problem of verifying temporal properties of concurrent parametrized systems, considering the following three main aspects:

Parametrized Systems: The main concern we study on this work is the parametrized verification of concurrent systems. The systems we study are composed by an unbounded number of threads. In general, these threads could be executing arbitrary different programs. We consider a simple scenario, in which all threads are executing the same program. This is the typical case of programs manipulating data structures.

The properties we analyze on these kind of systems are also parametrized by arbitrary threads. Consider, for instance, the following property:

$$\mathsf{mutex}(i,j) : \ \square\,(i \neq j \rightarrow \neg(\mathsf{critical}(i) \wedge \mathsf{critical}(j)))$$

Property mutex is described by a parametrized temporal formula stating that it is always the case that if threads i and j are different, then they are not at the critical section simultaneously. Note that as the formula is parametrized by thread identifiers i and j,

verifying this formula implies that the property holds for any two arbitrary threads which are part of the system.

Parametrized verification remains important because it enables the verification of properties for a family of system, independently on how many threads run such system. In other words, the verification of a parametrized property implies the verification of such property for any arbitrary instance of the system under study.

As expected, parametrized verification is also difficult to perform, since the reasoning must deal with an unbounded number of processes and threads. Under special circumstances techniques based on closed or bounded instances of a parametrized system can be used to carry some parametrized verification. However, these techniques are limited and the assumption of an unbounded system makes necessary the creation of specialized verification methods for parametrized systems.

Temporal Verification: In general, most of the previous work regarding parametrized verification targets only the analysis of safety properties, like the absence of null pointer dereferences, memory leaks and invalid references inside a program. However liveness properties are also interesting for reactive systems and specially for programs that manipulate concurrent data types. For example, consider a concurrent list. We would like not only to verify that the shape of the data structure is always of a valid list, but also that if a thread tries to insert a new element into the list, then this attempt will eventually succeed. Because of that, this work studies the construction of a general framework for the verification of both, safety and liveness properties, for parametrized concurrent systems.

Complex Pointer-based Datatypes: Most of real life programs require to manipulate the heap and use complex pointer-based data structures such as stacks, queues, lists and trees to store the data they manage. Hence, it is natural to seek a general verification method which may allow to cover as many different data types as possible.

There are numerous techniques specifically developed for abstracting the behaviour of a program into an abstract representation using simple data types such as Booleans and integers. In general, when abstractions are applied, part of the original information is lost due to the abstraction process itself. Moreover, not all abstractions are usually suitable for all complex data types. Hence, we study the development of a more general framework which do not rely on any data abstraction and which is able to cover a wide range of real complex data structures.

It is clear that, due to the state space explosion and the number of interleaving, software verification is crucial for certifying the correctness of any critical system. With the advent of multi-core technology, the importance of concurrent programs and the design of efficient concurrent data structures has become a major field of study in the last years. Concurrent data types are designed to exploit the parallelism of multiprocessor architectures by employing very weak forms of synchronization, like lock-freedom and fine-grain locking, allowing multiple threads to concurrently access the underlying data. Concurrent data structures are hard to design, challenging to implement correctly and even more difficult to formally prove correct. The formal verification of these concurrent programs is a very challenging task, particularly considering that they manipulate complex data structures capable of storing unbounded data, and are executed by

an unbounded number of threads. Consequently, novel verification techniques are required in order to cope with the verification of such complex systems.

The problem we study presents itself as interesting and challenging. The main difficulty in reasoning about concurrent data types comes from the interaction of *unstructured unbounded* concurrency, and heap manipulation. “Unstructured” concurrency refers to programs that are not structured in sections protected by locks and with clear memory footprints, but to programs that allow a more liberal pattern of memory accesses. In general, programmers attempts to use more liberal and unstructured memory accesses as doing so in general improve the overall performance of concurrent data structures implementations. But the reasoning about the correctness of the program is also “unbounded”, which refers to the aforementioned lack of a-priori bound on the number of threads. Because of this, it is impossible to estimate or compute the total number of threads involved in the system execution.

An advantage of parametrized verification is that it enables the analysis of a concurrent system for any possible thread instantiation. This means that once a parametrized property is verified, then the system can grow arbitrary big while still satisfying the verified properties. Parametrized verification provides the ultimate guarantee for evolving systems which may grow in the future. Furthermore, parametrized programs are useful in many settings including device drivers, distributed algorithms, concurrent data structures, robotic swarms, and biological systems among others.

Despite a lot of research concerning parametrized and concurrent verification has been done in the last years, there are still plenty of open questions that require to be answered.

Following the success of separation logic [180], most current verification techniques for heap manipulating programs are based on this logic. However, their application to the verification of concurrent programs has not yet been fully explored. For instance, we believe that one of the major drawbacks derived from the use of separation logic comes from its most important operator. When using the heap separating conjunction operator (represented with “*”), the heap is implicitly split into two disjoint sections. The problem here is that the user has no full control on how the heap is partitioned. This limitation comes evident with data structures such as skiplists, where a fine partitioning of memory cells may be required in order to represent the skiplist layout into the memory. To prevent this problem, we advocate for the use of explicit region manipulation, using sets and heap regions to denote disjoint sections of the heap. We believe that a explicit heap partitioning adapts better to the philosophy of unstructured concurrent data manipulation.

Currently, there exists a vast range of automatic techniques aiming the verification of concurrent programs. However, they are limited to programs that reason about simple data types, such as Booleans and integers, or they are restricted to the analysis of bounded data structures, such as skiplists with at most n levels. Even though, in some cases, complex data structures can be abstracted into simpler representations which enable the use of automatic techniques, most concurrent programs deal with complex pointer-based data structures such as lists or trees in order to store their data and information. For this reason, we target a more general verification framework based on deductive methods, willing to sacrifice automation in exchange for generality on the variety of data structures we can analyze.

Finally, most of the verification work under development is restricted to the study of safety properties. We advocate for a single verification framework general enough as to tackle both, safety and liveness temporal parametrized properties.

Bearing these problems in mind, the main contribution of this work is a general verification

framework for the analysis of parametrized temporal properties, capable of dealing with the class of systems stated above. In the framework we propose, we interpret the operational semantics of the program as a state transition system and we use deductive techniques to reduce the verification problem to a proof of validity of a finite collection of verification conditions.

The framework we propose is grounded on the following basis:

- We propose the use of deductive verification methods instead of fully automatic techniques. Besides been incomplete, automatic methods are also currently limited to the analysis of simple data types. We believe that the use of deductive techniques enables the verification of richer data structures in exchange of some extra work from part of the user. Moreover, in our experience, the extra work needed to be done by the user is very close to the reasoning needed to understand the functionality of the program and the data types. In other words, deductive methods present themselves as natural candidates for reasoning about parametrized concurrent systems.

- In order to target the verification of temporal parametrized properties, we propose the use of specialized proof rules and formalisms capable of tackling both safety and liveness properties. The deductive techniques we suggest ultimately reduce the verification process to a finite collection of verification conditions.

  For safety properties, we propose novel proof rules for parametrized systems following the ideas behind well known deductive methods for the verification of invariants in non-parametrized systems. The proof rules we include in our framework extract a finite collection of verification conditions given a program description and a temporal safety property. The validity of these verification conditions entails the fact that the safety property is satisfied by any instance of systems composed by any arbitrary number of threads executing the program. A key aspect of these proof rules is that the number of generated verification conditions depends solely of the length of the program and the number of threads parametrizing the safety specification, but never on the total number of threads involved in the systems.

  In the case of liveness properties, we present an extension of a deductive formalism originally developed for verifying temporal specifications of closed systems. The method is based on the construction of a formula automaton which represents the proof that a system composed by an arbitrary number of threads executing a specific program satisfies a given liveness property. Similarly to the proof rules for safety properties, the formalism we propose for liveness verification ultimately generates a finite collection of verification conditions, whose number depends solely on the program and the temporal specification itself.

- We suggest the use of specialized decision procedures to enable the study of programs that manipulate a wide range of data structures and not only simple data types. We believe that following this idea, it is possible to use our framework to verify any program which manipulates a data type for which there exists a decision procedure. The basis behind this idea is that if a program manipulates a data structure for which there exists a decision procedure, then such decision procedure can be used to automatically validate the verification conditions generated using a specialized deductive techniques.

From our point of view, one of the main advantages of our framework is the clean separation it makes between:

1. the analysis of the program control flow, and;
2. the analysis of the data manipulated by the program.

While the control flow of the concurrent program is verified using novel deductive proof rules and formalisms, the manipulation of data is verified using specialized decision procedures. By clearly distinguishing control flow from data, our framework is even more reusable. For example, our parametrized verification techniques for safety and liveness can virtually be used for the analysis of concurrent programs manipulating any data type as far as a decision procedure exists for such data type. Similarly, the decision procedures we present here can be used for checking the satisfiability of formulas which has not been necessarily generated using our verification techniques.

1.1 Related Work

The main application goal of this work is the verification of parametrized concurrent data types [107], where the main difficulty arises, as we have stated before, from the mix of *unstructured unbounded* concurrency and heap manipulation.

This work tackles the verification of concurrent data types using the assumption of full symmetry. In fact, concurrent data types can be modeled naturally as fully symmetric parametrized systems, where each thread executes in parallel a client of the data type. Under the principle of full symmetry, all threads identifiers are interchangeable.

Related to our work, there exists a vast bibliography which deals with logics for concurrent heap reasoning, parametrized verification for concurrent systems and decision procedures. We now present a brief description of the most relevant work within each of these areas.

1.1.1 Logics for Concurrent Heap Reasoning

The literature involving decidable logics for describing mutable data structures is rather extensive. Although there exist frameworks based techniques such as model-checking [211], in general, most of the work in formal verification of sequential pointer programs is based on program logics following the Hoare tradition, equipped to deal with pointer structures in the heap [34, 132, 213]. The problem is that extending these logics to deal with concurrent programs is hard, and though some success has been accomplished this is still an open area of research, particularly for liveness properties [54, 92].

In this sense, separation logic [165, 180] is the most well known and extensively used general framework for describing dynamically allocated mutable data structures in the heap. In fact, it has been shown [117] that, in general, any separation logic formula using rather general recursively defined predicates (which in general should be enough to describe single and double linked lists, trees and a combination of them) is decidable for satisfiability.

When considering concurrency, there is a whole successful line of research involving concurrent separation logic [38, 110, 164]. Some of the techniques developed for the application of separation logic to concurrent systems include the analysis of shared variables [33, 166]; permission accounting [32]; concurrent abstract predicates [67]; impredicative concurrent abstract predicates [196]; concurrent local subjective logic [173]; granularity abstraction for modular verification of higher-order concurrent programs [202]; fine-grained concurrent separation

logic [159]; combination of rely-guarantee with separation logic (RGSep) [208] and combination of fractional permissions with abstract predicates [97] which enables the verification of data race freedom, absence of null-dereferences and partial correctness.

The main advantage of separation logic is its ability to describe compositional proofs, mainly thanks to the principle of local reasoning. This principle allows to work on disjoint parts of the global heap, combining later the result of the analysis through a composition operator which implicitly sets a partition in the heap. Interestingly, this ability of implicitly splitting the heap is also one of the main weakness of separation logic, as the user cannot refer explicitly to the partitioning of the heap. On the contrary, on this work we advocate for the use of explicit heap regions [14] which provide the user full control over the heap partitioning. Explicit regions allow the use of a classical first-order assertion language to reason about heaps, including mutation and disjointness of memory regions. Regions correspond to finite sets of object references. Unlike separation logic, the theory of sets [210] can be easily combined with other classical theories to build more powerful decision procedures. Classical theories are also amenable of integration into SMT solvers [16]. In practice, using explicit regions requires the annotation and manipulation of ghost variables of type *region*, but adding these annotations is usually straightforward.

Separation logic [164] has been successfully used in the verification of many non-blocking algorithms which manipulate data structures such as lists, stacks, queues and many mutual exclusion protocols among others [92, 167, 203, 207]. When dealing with more complex data structures, such as skiplists, an alternative for memory layouts is shape analysis [209], like forest automata [4, 112]. However, this approach can handle skiplists only of a bounded height (empirical evaluation suggests a current limit of 3). Unlike [112], our approach is not fully automatic in the sense that it requires some annotations—provided by the user—, but on the other hand our approach can handle arbitrary skiplists. The burden of additional annotation can be alleviated with methods like invariant generation, something we also explore in this work.

1.1.2 Parametrized Verification

The problem of verifying parametrized finite state systems has received a lot of attention in recent years. In general, the problem is undecidable [9], even for finite state components [195]. There are two ways to overcome this limitation:

(i) algorithmic approaches, which are necessarily incomplete; and

(ii) deductive proof methods.

Typically, algorithmic methods—in order to regain decidability—are restricted to finite state processes [44, 45, 70] and finite state shared data. Some examples are synchronous systems with guards [87]; interleaving systems with pairwise rendezvous [73]; systems with only conjunctive guards or only disjunctive guards [70]; implicit induction [72]; network invariants [137]; context-free network grammar abstractions for verifying families of state-transition systems [47]; and abstractions of configurations as words from a regular language [1, 3, 46, 147]. A related technique, used in parametrized model checking, is symmetry reduction [48, 74]. There also exist some automatic approaches designed to automatically verify specific properties such as linearizability [205]. Our approach is not automatic, but can be used to prove many properties other than linearizability.

Other approaches are based on shape analysis, like [24]. However, in general, they are limited to a fixed number of threads [6] or limited to fixed data structures or shapes, like simple linked-list data structures [204]. We follow an alternative approach, by extending temporal deductive methods like Manna-Pnueli [144] with specialized proof rules and formalisms for parametrized systems, thus sacrificing full automation to handle complex concurrency and data manipulation. This style of reasoning allows a clean separation in a proof between the temporal part (why the interleaving of actions that a set of threads can perform satisfy a certain property) and the underlying data being manipulated. Temporal deductive methods, like ours, are very powerful to reason about (structured or unstructured) concurrency, but they have been traditionally restricted to non-parametrized systems and scalar data. Our approach can be applied to any theory of data with an available decision procedure. These aforementioned automatic approaches based on shape analysis can alleviate the human intervention needed in our approach by generating intermediate invariants.

Property directed techniques can be used to automatically prove invariants without manual effort [111], but they are in general restricted to Boolean programs. Some approaches that use abstraction, like thread quantification [24] and environment abstraction [49], are based on similar principles as the full symmetry presented in this work. However, these approaches rely on building specific abstract domains that abstract symbolic states instead of using decision procedures based on SMT solvers, as in our work. A very powerful method is invisible invariants [10, 169, 214], which works by heuristically generating invariants on small instantiations and trying to generalize these to parametrized invariants. However, this method is so far restricted to finite state processes. There exist results [2] proving decidability for systems with finite control flow and infinite domain provided the infinite domain contains a well-founded preorder. A key difference is that, unlike in [169], we generate quantifier-free verification conditions, which eases the development of decision procedures for complex data types.

In contrast with these methods, the verification framework we present here can be applied to any finite or infinite data domain for which there exists a decision procedure. The price to pay is, of course, automation because one needs to provide additional program annotations in the form of supporting invariants. We see our line of research as complementary to the lines mentioned above. We start from a general method and investigate how to improve automation as opposed to starting from a restricted automatic technique and improve its applicability. The verification conditions we generate through our methods can still be verified automatically as long as there are decision procedures for the data that the program manipulates.

Regarding liveness, we extend the formalism of general verification diagrams [39, 189] so that it can deal with concurrent parametrized systems. The work that is closest to ours is [145] and chapter 8 of [28], which also use diagrams to verify temporal properties of reactive systems. However, they use quantification in the nodes and hence generate quantified verification conditions. In many cases, using quantifiers sacrifices the automation in the proof of the generated verification conditions. In this work, we present techniques for generating quantifier-free verification conditions for parametrized systems that can be handled automatically by SMT solvers.

A work close to ours is the deductive verification framework (DVF) [90], which consists on a language and a tool for verifying properties of transition systems by generating verification conditions from specific goals which are then passed to SMT engines. However, DFV is based on Hoare style reasoning and does not tackle parametrized verification.

Environment abstraction [49] and thread quantification abstractions [24] are abstraction based techniques that deal with parametrized systems, but these techniques abstract processes and data altogether, and hence are much more difficult to extend to arbitrary data and memory layouts. In contrast, our approach can be applied to any data type as long as there is a decision procedure for its memory state. This is also a key difference between our approach and parametrized model checking for symmetric systems [71]. Similarly, [96] shows a verification approach which uses abstract transition systems to simulate lock-free algorithms, however it is limited only to safety and the simulation obscures the verification. Our framework, on the other hand, is capable of dealing with liveness properties of lock-free algorithms.

A related technique for the verification of temporal properties is transition invariants [170], which characterize the validity of termination or other liveness properties by the existence of a disjunctively well-founded transition invariant. A transition invariant is a relation-based abstraction of the transition relation of a program expressed as a disjunctive set of relations. We, on the other hand, propose a formalism which denotes a state-based abstraction of the reachable state-space of a program. Other techniques for checking program termination include [64] and [89]. However, all these techniques are restricted to non-parametrized systems.

1.1.3 Automatic Parametrized Invariant Generation

As part of our verification framework, we study the automation of invariant generation for parametrized systems. We propose a method which consists of abstracting the environment as a singe thread. Our approach to deal with automatic parametrized invariant generation is an instance of the general framework of thread-modular reasoning [49, 81, 101, 142], where one reasons about a thread in isolation given some assumptions about its environment (i.e., the other concurrently executing threads). Notably, the approach we consider in this work builds the assumptions incrementally via self-reflection.

One of the main issues in verifying parametrized programs is the interaction between a given thread and its environment, consisting of the remaining threads. Abstracting this interaction finitely has been considered recently [24, 76]. In particular, the approach of [24] is very closely related. Similarities include the notion of transferring invariants from a materialized thread to the abstraction of the remaining threads. However, [24] does not explicitly specify an iteration scheme describing how the inferred candidate invariants are transferred to the environment abstraction. Besides, the effects of widening, including the potential non-monotonicity in many domains, are not studied. As we will see in Chapter 5, such considerations have a significant impact on the generated invariants. Another recent contribution is [76] which explores the interleaving of control and data-flow analysis to better model the thread interference in parametrized programs.

In our work, we abstract away the effects of interacting threads by projecting away the local variables. This idea is quite standard. For example, [155] analyzes multi-threaded embedded systems using this abstraction. Similarly, [121] presents a framework for the abstract interpretation of multi-threaded programs with finitely-many threads. In such work, a *melding operator* is used to model the effect of an interfering thread on the abstract state of the current thread.

The approach we use does not explicitly handle synchronization constructs such as locks and pairwise rendezvous. These constructs can be handled using the framework of *transaction delineation* presented in [121]. What we do is to use a single-threaded sequential analysis pass to identify sections of the program which can be executed "atomically" while safely ignoring the

interferences by the remaining threads.

Another class of approaches rely on finite model properties wherein invariants of finite instantiations generalize to the parametrized system as a whole. One such approach is that of invisible invariants pioneered by [169, 214]. This approach finds inductive invariants by fixing the number of processes and computing invariants on the instantiated system. These invariants are heuristically generalized to the parametrized system, which are then checked to be inductive. In [148], invisible invariants are generalized in the abstract interpretation framework as fixed points. In specific instances, a finite model property is used to justify the completeness of this technique. A related method is that of splitting invariants [50, 158] that ease the automation of invariant generation but also assumes finite state processes and the existence of a cut-off [70].

Other methods for automatically generating invariants consist on instrumenting the program and letting a program to guess program invariants from a collection of executions [26, 75], even in combination with genetic algorithms [82]. However, these methods cannot tackle the problem of generating parametrized invariants, as we do.

1.1.4 Verification Tools

The ideas presented in this work are implemented as part of a tool called LEAP. There exists a wide range of tools for verifying concurrent systems. *Smallfoot* [23] is an automatic verifier that uses concurrent separation logic for verifying sequential and concurrent programs. *Smallfoot* depends on built-in rules for the data types, which are typically recursive definitions in separation logic. Unlike our tool LEAP, *Smallfoot* cannot handle programs without strict separation (like shared readers) or algorithms that do not follow the unrolling that is explicit in the recursive definitions. *TLA+* [42] is able to verify temporal properties of concurrent systems with the aid of theorem provers and SMT solvers, but *TLA+* does not support decision procedures for data in the heap. Similarly, *HAVOC* [53] is capable of verifying C programs relying on Boogie as intermediate language and Z3 as backend. Neither *Frama-C* [60] nor *Jahob* [129] handle parametrized verification, which is necessary to verify concurrent data types (for any number of threads). The closest system to LEAP is *STeP* [143], but *STeP* only handles temporal proofs for simple data types. Unlike LEAP, none of these tools can reason about parametrized systems.

Chalice [135, 136] is an experimental language that explores specification and verification of concurrency in programs with dynamic thread creation and locks. *VeriCool* [191] uses dynamic framing (as *Chalice* does) to tackle the verification of concurrent programs using *Z3* as backend. *VerCors* [31] is a tool for verifying concurrent multi-threaded and vector-based programming models which relies on *Chalice*. However, none of these tools implement specialized decision procedures for complex theories of data types. *VCC* [51] is an industrial-strength verification environment for low-level concurrent system code written in C. Verification in *VCC* is done by specifying preconditions, postconditions and type invariants. In comparison to LEAP it requires a great amount of program annotation.

Other tools that employ separation logic as specification language or as an internal representation language include *Space Invader* [21], *Sleek* [163], *CAVE* [205, 206], *Predator* [68] and *Infer* [41] among others.

A specification language which can be used as the framework we present here is *Why3* [78], which can also be used as intermediated language for other verification tools. However, it is not specifically designed for tackling the verification of parametrized systems.

Specifically for dealing with liveness verification, there exist tools such as *Terminator* [55] which can check liveness properties through program termination [54], but to our knowledge it does not support parametrized systems.

1.1.5 Decision Procedures

In order to automatically verify properties of concurrent data structures, our approach relies on the use of decision procedures. Since the work carried out by Burstall [40], many approaches have been suggested and studied in order to deal with the verification of programs that manipulate pointer-based data structures [13, 20, 27, 61, 119, 131, 149, 161]. However, most of these approaches experiment serious difficulties when reasoning about theories that combines pointers with data. The key aspect in many of these approaches resides in the availability of decision procedures to reason about cells (the basic building blocks of the data structure), memories, and a reachability notion induced by following pointers. As reachability is not a first-order concept, some feature must be added in order to cope with reachability. Hence, despite the existence of precise and automatic techniques to reason about pointer reachability [119], not many approaches focus on the combination of such techniques with decision procedures capable of reasoning about data, pointers or locks. As a consequence, approximate solutions have come out and little is known about the combination of such logics with decidable first-order theories to reason about data and pointer values. In some cases, the information about data and pointers is abstracted away to make the use of reachability tools [61] possible. In other cases, a first-order approximation of reachability is used [161] so that decision procedures for the theories of pointers and data can be used.

In [20], the decidability of a logic for pointer-based data structure is proven by showing that it enjoys from finite model property. That is, given a formula, a finite model can be constructed and used to witness the satisfiability of the formula. However, this logic can only reason about reachability on lists, trees and graphs, but it cannot reason about the data stored in the data structure. A generalization of [20] is presented in [213], but the emphasis once again is put on expressing complex shape constraints rather than considering the data stored in the data type. In general, pointer logics like [34, 132, 213] are very powerful to describe pointer structures, but they require the use of quantifiers to reach their expressive power. In [22] the authors present a fragment of separation logic oriented to linked lists whose decidability depends on a small model property while [176] presents a decidable logic for heap manipulating programs. Similarly, in combination with predicate abstraction, [13, 27] describe decision procedures which abstract away the data. The problem these approaches suffer is that in general they are not expressible enough. Some decidable logics for mutable data structures [35, 157] are based on reductions to monadic second order formulas. For instance, *CSL* [34] uses first-order logic with reachability in combination with arithmetic theories to reason about shape, path lengths and data within the heap. Similarly, *STRAND* [140] combines monadic second order logic, graphs and quantified theories in order to provide decidable fragments using a reduction to monadic second order theory of bounded trees. In these cases, the use of quantifiers precludes the combination of these logics, using methods like Nelson-Oppen [162] or BAPA [128], with other aspects of the program state to obtain an automatic reasoning engine for the combined theory. In comparison, the logics we introduce in this work satisfy the conditions of the Nelson-Oppen combination method, and hence can be extended with other reasoning procedures.

Some logics such as [62] are capable of verifying pointer based sequential programs using rewriting and tableaux. For many data types one can use directly SMT solvers [63, 84], or specialized decision procedures (as we do) built on top of these solvers.

The approach we follow in this work is the construction of monolithic decision procedures for the theories we develop. However, a more efficient and elegant approach consists on constructing decision procedures for individual theories and then combine them for obtaining a decision procedure for the combined theory. Two of the most well-known combination methods for first order logic are Nelson-Oppen [162] and Shostak [181, 188]. In the Nelson-Oppen method, decision procedures for two disjoint theories are combined by introducing variables to name subterms, iteratively propagating any deduced equalities between variables. In the case of the Shostak method, it uses an optimized implementation of the congruence closure procedure for ground equality over uninterpreted function symbols to combine theories. Both methods serve the same purpose, to enable the combination of decision procedures for quantifier-free first-order theories with disjoint signatures. For these two methods, some variations [18] and extensions have been studied [12, 88]. Additionally, they are in the core of tools such as *ICS* [77], *Simplify* [65] and *Verifun* [80] among others.

As most current systems studied in software verification are typed, it is more natural to express verification problems in many-sorted first-order logics [83, 146], as we do in this work. The theories we construct are capable of expressing rich properties of pointer-based data structures, taking as starting point a theory for single-linked lists [178, 179]. The decidable logics we present in this work can be used as stand-alone decidable theories with a decision procedure built on top of SMT solvers. These decision procedures can be used to verify general quantifier-free verification conditions independently of the method used to generate these verification conditions. In fact, as we use our theories in order to verify verification conditions generated from the small-step semantics of program statements, problems such as intermediate assertion generation [8] do not affect our theories.

In order to extend decidable theories it is possible to use techniques such as local theory extensions [192,192]. Instead, we start from [178] and incrementally construct decidable theories by ensuring they meet the required combination restrictions [177, 199, 200]. A key aspect is that the theories we construct remain quantifier-free while they satisfy the necessary combination requirements, making them perfect candidates for being combined with other theories in order to verify complex systems.

1.2 Structure of this Work

The structure of this work presents the concepts and ideas incrementally. Fig. 1.1 shows a graphic representation of the structure of this work. This dissertation contains 11 chapters with the following content:

Chapter 1: presents the problem of formally verifying parametrized temporal properties of concurrent systems executed by an unbounded number of threads. This chapter presents the importance, the main challenges and the main ideas toward the solution we propose for the parametrized verification problem. Additionally, this chapter presents a survey involving current related work and how our approach relates with them.

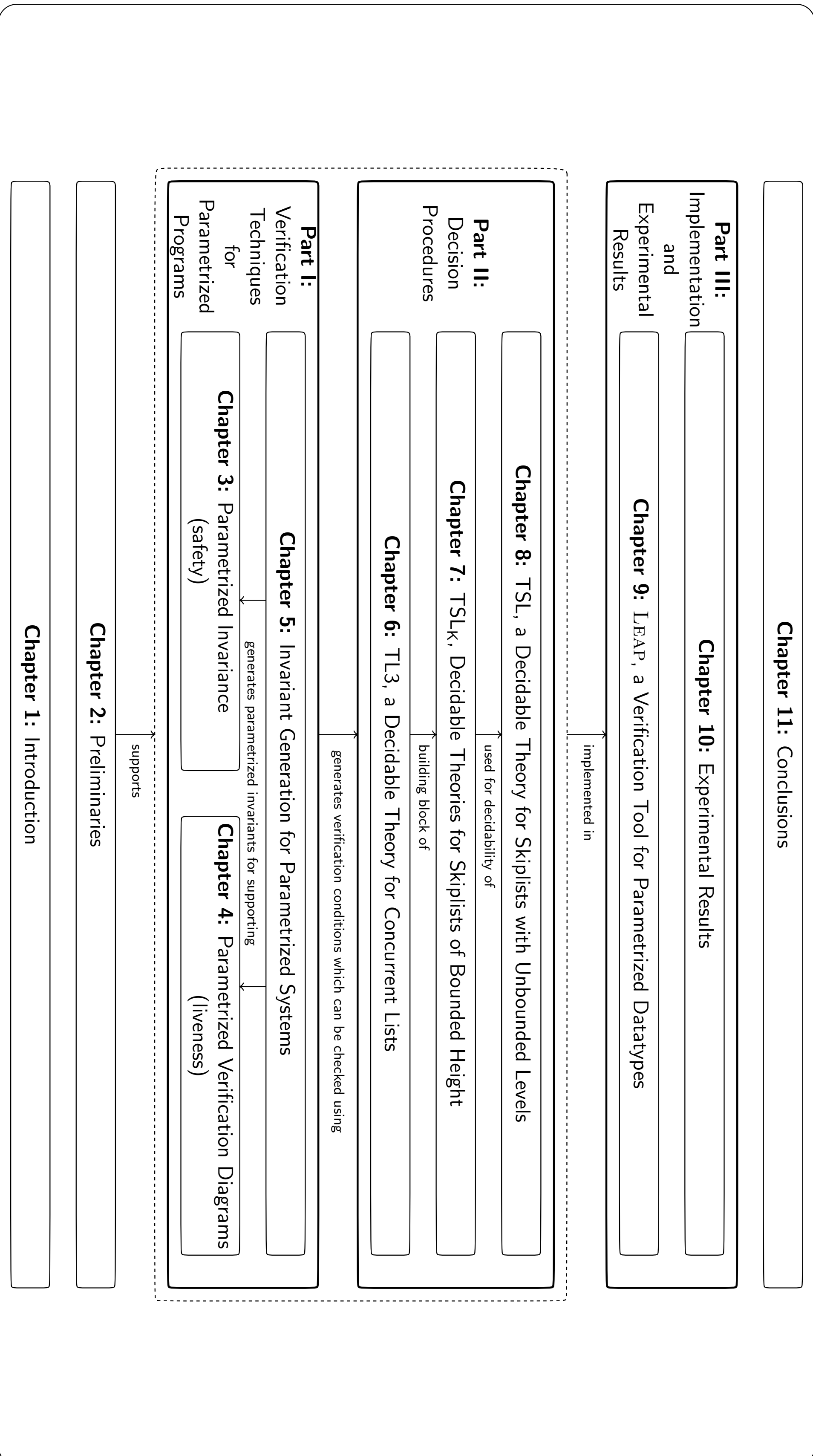


Figure 1.1: Structure of this work.

Chapter 2: presents the preliminaries concepts, definitions and notations that will be required for fully understanding the rest of the chapters. In this chapter we introduce the syntax and semantics of SPL, the *Simplified Programming Language* we use to describe the implementation of the algorithms and programs that we will later verify. Additionally, we formally introduce the concepts of theory signature and theory combination which we will require for later chapters.

This chapter also presents the theoretical model of computation we use in the rest of the work. We introduce the concept of a parametrized fair transition system, which extends conventional (non-parametrized) fair transition systems, enabling the representation of parametrized systems executed by an unbounded number of threads. Similarly, we introduce parametrized concurrent programs, parametrized formulas and the problem of the verification of temporal parametrized properties.

Finally, this chapter describes some basic concepts regarding concurrency which are required for fully understanding the development of the upcoming chapters. These problems include linearizability, lock-based algorithms, lock-free implementations and general synchronization methods.

Part I: presents the verification techniques we have developed for the verification of parametrized concurrent systems. This Part consists of three chapters:

Chapter 3: introduces *Parametrized Invariance*, which is a general deductive method based on specialized proof rules which aims at the verification of safety temporal properties of concurrent parametrized systems. In this chapter we present the traditional deductive invariance proof rules for non-parametrized systems and we show why they are unsuitable for parametrized systems. Then, we introduce novel proof rules for parametrized systems and we show them sound.

Chapter 4: presents *parametrized verification diagrams* (PVDs), a deductive diagram–based formalism that allows to prove temporal properties, including *liveness* properties, of parametrized concurrent systems. PVDs encode succinctly as a formula automaton the proof that a parametrized system satisfies a given temporal property. This chapter provides a brief introduction to *generalized verification diagrams* (the formalism that PVDs extend from) and presents its drawbacks when dealing with parametrized systems. Finally, in this chapter we formally present PVDs and we show that this formalism is sound. We also show how quantifier-free verification conditions are extracted from a PVD. The validity of these verification conditions implies that the PVD in fact represents the proof that the system satisfies the parametrized temporal property.

Chapter 5: describes an abstract-interpretation based method for automatically generating parametrized invariants, leveraging off-the-shelf existing invariant generators for sequential non-parametrized systems.

This chapter introduces the novel idea of reflective abstraction, which enables the incremental construction of a sequential transition system that abstracts a parametrized one. This chapter also describes how to construct an iterative procedure to automatically generate invariants of a parametrized system using abstract interpretation on reflective abstractions. Additionally, this chapter presents a set of alternative abstract interpretation schemes using our method.

Part II: contains a set of decidable theories for pointer based data structures. The techniques presented in Chapter 3 and Chapter 4 ultimately generate a collection of verification conditions whose validity can be automatically checked using specialized decision procedures like the ones presented in this Part. This Part consists of three chapters:

Chapter 6: introduces TL3—the *Theory of Linked Lists with Locks*— a decidable theory of single-linked lists, which is also equipped with features to reason about concurrency. We use TL3 to study the verification problem of concurrent data types that manipulate dynamic memory in the heap maintaining the shape of single-linked lists. Programs that can be analyzed using TL3 include concurrent lock-coupling lists and lock-free queues and stacks. This chapter presents some concurrent data types that manipulate single-linked lists alike data-structures, formally introduces the theory TL3 and shows that TL3 is decidable by stating and proving a bounded model theorem.

Chapter 7: introduces TSL_K—the *Family of Theories of Concurrent Skiplists with at Most K Levels*— a decidable family of theories that allow to reason about the skiplist memory layout, for skiplists of unbounded length but with a bounded number of levels. TSL_K extends the reasoning of TL3 to K levels, adds some functions and predicates to deal with order and introduces a notion of sublist between ordered lists. This chapter presents the skiplist data structure, including an implementation of skiplists with a constant number of levels, it formally presents the theory TSL_K and shows that TSL_K is also decidable by stating and proving a bounded model theorem.

Chapter 8: presents TSL—*The Theory of Skiplists with Unbounded Levels*— a decidable theory capable of dealing with skiplists of arbitrary length and height. This chapter shows an implementation of skiplists with an unbounded and growing number of levels, formally presents theory TSL, and shows that TSL is decidable by reducing the satisfiability problem of TSL quantifier-free formulas to queries to decision procedures for TSL_K and Presburger arithmetic.

Part III: This part presents LEAP, a tool that implements the verification techniques introduced in Part I and the decision procedures described in Part II along with some experimental results we have obtained so far. This Part consists of two chapters:

Chapter 9: presents LEAP, a prototype tool for the verification of concurrent data types and parametrized systems composed by an unbounded number of threads that manipulate infinite data. LEAP implements the verification techniques of parametrized invariance and verification diagrams presented in Chapter 3 and Chapter 4 respectively. Additionally, it implements all the decision procedures introduced in Part II. We have developed LEAP to show the feasibility of the application of our techniques.

Chapter 10: shows the results of the empirical evaluation we have done using LEAP to formally verify parametrized temporal properties, including safety and liveness, of algorithms and data types including mutual exclusion protocols, concurrent lock-coupling single-linked lists, lock-free stacks and queues and bounded and unbounded skiplists implementations. Additionally, we present some experimental results showing the automatic generation of invariants of a collection of parametrized systems.

Chapter 11: provides a summary of what we present in this work, the problems we have attacked, the solutions we have proposed for each of these problems, the advantages and disadvantages of our solution and some discussion about possible directions for future work and the open questions that remain to be answered.

2

Preliminaries

> *No one is dumb who is curious. The people who don't ask questions remain clueless throughout their lives.*
>
> Neil deGrasse Tyson

In this chapter we introduce all the preliminaries concepts, definitions, and notations that will be required for the rest of the chapters. Section 2.1 presents the *Simplified Programming Language* (SPL) we will use to describe algorithms through this work and describes the operational semantic of each SPL statement as a transition relation. Section 2.2 presents some basic definitions related to concurrency such as linearization, lock-based algorithms, lock-free algorithms and general synchronization methods. Section 2.3 introduces the concept of signatures and theories. Finally, Section 2.4 presents the theoretical model of computation, revisiting the traditional definition of non-parametrized fair transition systems and presenting the concepts of parametrized concurrent programs, parametrized fair transition systems, parametrized formulas and the parametrized temporal verification problem.

2.1 Simplified Programming Language (SPL)

When describing a program or algorithm we use our version of the *simplified programming language* (SPL) [144], a simple imperative language. Our version is equipped with variable declaration, procedure calls, conditionals, loops and atomic sections.

A program in SPL consists of two sections. The first section, which starts with the **global** keyword, declares all global variables. The second section declares the procedures that are part of the program.

A procedure declaration starts with the **procedure** keyword, followed by the procedure's name and a list of arguments surrounded by parenthesis. This list of arguments can be empty.

Immediately after the list of procedure's arguments it is possible to declare the set of local variables that belong to the procedure. Following the declaration of local variables, the **begin** keyword indicates the beginning of the body of the procedure, which contains the list of statements that are part of the procedure. Finally, the procedure finishes with the **end procedure** keyword.

The basic types supported by SPL are:

- $Bool$: the Boolean type. Elements of this type are $true$ and $false$.
- Int: the integer type, which accepts the classical operations over integers, such as addition $(+)$, subtraction $(-)$ and multiplication $(\times)$.
- Tid: the type of thread identifiers. Elements of this type are used to denote specific thread identifiers and can only be compared through equality. There exists a reserved word me which denotes the identifier of the thread currently executing the program. Additionally, the constant value $\oslash$ of type Tid denotes the absence of a thread identifier. In particular, if l is a lock, then the program expression $l = \oslash$ represents that the lock is not owned by any thread.
- $Addr$: the type of addresses. Addresses represent locations in the dynamic memory as well as pointers. SPL does not support pointer arithmetic.
- $Elem$: the type of generic elements stored in a data structure.
- $Lock$: the type of mutexes that can be used to protect critical sections. We assume the existence of the $lock$ and $unlock$ operations provided by the operating system to acquire and release a lock.

SPL also defines the following collection types:

- $Set\langle T\rangle$: for sets of elements of type T. This type accepts the basic operations between sets such as union $(\cup)$, intersection $(\cap)$, set difference $(-)$ and inclusion $(\subseteq)$.
- $Array\langle T\rangle$: which represents arrays containing elements of type T indexed by integers. The operations accepted by this type are array lookup and array update. In the case of array lookup, if A is an array of type $Array\langle T\rangle$ and i is an element of type Int, then $A[i]$ is the element of type T stored in A at position i. Similarly, if A is an array of type $Array\langle T\rangle$, i is an element of type Int and e is an element of type T, then $A\{i \leftarrow e\}$ is an array of type $Array\langle T\rangle$ that contains e stored at position i and coincides with A on all other positions different from i.

Example 2.1

Fig. 2.1 illustrates the main parts of an SPL program with a simple example. This example presents a program with a global variable $count$ of type Int and a procedure named MULTIPLIESTWO which receives an integer as argument and proceeds to compute the double of the value received. Additionally, the result of the operation is added to the global $count$ variable.

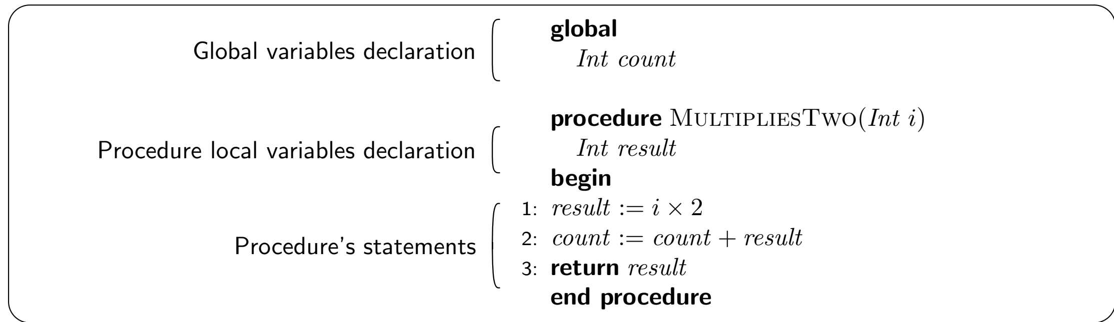


Figure 2.1: A simple example showing the structure of an SPL program.

Given a procedure $proc$, we use the notation $proc{::}v$ to refer to a variable v local to $proc$. We may also omit $proc{::}$ and simply write v when the procedure it belongs to is clear from the context.

If t is a thread identifier, we use $pc(t)$ to refer to the program counter of thread t. Similarly, we use pc' to refer to the program counter in the state reached after the statement has been executed. We use $pc(t) = i, j$ and $pc(t) = i_1..i_n$ to denote $(pc(t) = i \vee pc(t) = j)$ and $\bigvee_{j=1}^{n} pc(t) = i_j$ respectively. When needed, we use pc_P to denote the program counter when procedure P executes.

2.1.1 SPL Statements

In addition to the operations over basic and collection types, SPL provides statements for basic operations such as procedure calls, conditionals, program loops and atomic sections. We now present the program statements available in SPL:

Assignments: The operator := is used to assign values to variables. For instance $i := 1$ is the assignment of value 1 to variable i while $v := w$ corresponds to assigning the value of variable w to variable v.

Pointer access: The operator→ is used to access a data structure field though a pointer.

No operation: The operator **skip** corresponds to the statement that performs no operation.

Conditionals: SPL provides conditional statements of the form:

```
if cond then          if cond then
    S1                    S1
end if                else
                          S2
                      end if
```

These conditionals have the usual semantics. That is, if $cond$ holds then the statements within block S_1 are executed. If $cond$ does not hold, then nothing is executed in the first case and the block in S_2 is executed in the second case.

Loops: SPL provides loops based on Boolean conditions of the form:

while $cond$ **do**
    S
end while

where block S is continuously executed as long as condition $cond$ holds. We will usually use **for** loops as a syntactic sugar for conditional loops controlled by integer variables. That way:

for $i := n$ **to** m **do**
    S
end for

for $i := n$ **downto** m **do**
    S
end for

are syntactic sugar for:

$i := n$
while $i \leq m$ **do**
    S
    $i := i + 1$
end while

$i := n$
while $i \geq m$ **do**
    S
    $i := i - 1$
end while

'

Non determnistic choice: Non-deterministic choice between various statements can be expressed using the **or** operator. For example:

nondet choice
    $v := 1$
or $v := 2$
or $v := 3$
end choice

non-deterministically assigns to variable v a value between 1 and 3.

Wait on condition: A procedure can wait on a condition using the **await** operator. For instance, the statement **await** $(v > 0)$ stops the execution of the procedure until variable v becomes positive.

Critical sections: Statements **noncritical** and **critical** are schematic statements used to denote sections in mutual exclusion programs.

Procedure calls: Procedures are invoked using the special operator **call**. A procedure may return a value using a **return** statement. For instance, a call to the procedure MULTIPLIESTWO defined above may be of the form:

- $v :=$ **call** MULTIPLIESTWO(3), which assigns to variable v the value returned by the procedure MULTIPLIESTWO when invoked with value 3 as argument; or
- **call** MULTIPLIESTWO(3), which invokes and executes the procedure MULTIPLIESTWO, but ignores the value returned from such procedure.

Atomic sections: Statements may be surrounded by $\langle\rangle$ in order to denote that they are executed atomically. For instance:

$$\left\langle \begin{array}{l} result := i \times 2 \\ count := count + result \end{array} \right\rangle$$

atomically assigns to variable $result$ the double of variable i and adds the result to variable $count$.

Later, in Section 2.1.2 we will formally describe the semantics of each of the statements presented here by exhibiting the corresponding transition relation.

Example 2.2 (A Mutual Exclusion Protocol Based on Tickets)
Fig. 2.2 presents two mutual exclusion protocols based on tickets that will serve as examples of programs declared in SPL. We will also use these programs as running examples to illustrate concepts and the application of some techniques developed in later chapters.

Fig. 2.2(a) contains SETMUTEX, a parametrized mutual exclusion protocol based on tickets. Each thread that intends to access the critical section at line 5, acquires a ticket with a unique and increasing number and—atomically—announces its intention to enter the critical section by adding the ticket to a shared global set of tickets (line 3). Then, the thread waits (line 4) until its ticket becomes the lowest value in the set before entering the critical section. After a thread leaves the critical section it removes its ticket from the global set (line 6). SETMUTEX uses two global variables: $avail$, of type Int, which stores the shared counter; and bag, of type $Set\langle Int\rangle$, which stores the set of tickets owned by those threads that are trying to access the critical section. Program INTMUTEX in Fig. 2.2(b) implements a similar version of the protocol in which only the minimum value among all given tickets is maintained, in a global variable of type Int. Note that both programs are infinite state for any concrete instantiation (number of threads running in parallel) since the available ticket is ever increasing. ⌟

```
   global
     Int avail := 0
     Set⟨Int⟩ bag := ∅

   procedure SETMUTEX
     Int ticket := 0
   begin
1: while true do
2:     noncritical
3:     ⟨ ticket := avail++
         bag.add(ticket) ⟩
4:     await (bag.min == ticket)
5:     critical
6:     bag.remove(ticket)
7: end while
   end procedure
```

(a) SETMUTEX, using a set of integers

```
   global
     Int avail := 0
     Int min := 0

   procedure INTMUTEX
     Int ticket
   begin
1: while true do
2:     noncritical
3:     ticket := avail++
4:     await (min == ticket)
5:     critical
6:     min := min + 1
7: end while
   end procedure
```

(b) INTMUTEX, using two counters

Figure 2.2: Two implementations of a ticket based mutual exclusion protocol.

2.1.2 Semantics of SPL Statements

We now formally describe the semantics of each statement in SPL as a first-order formula $\rho(V, V')$ which relates the values of a set of variables V in the pre-state with the values of the set of variables V' in the post-state. Each transition relation describes the effect of executing the statement. As usual, given a transition relation $\rho(V, V')$, we say that the enabling condition of ρ is the formula $\exists X'.\rho(V, X')$. For all cases, we provide the transition relation assuming that each program statement is executed by a thread T. We also assume the existence of a global variable $heap$ which models the dynamic memory. We make $heap$ as an array of memory cells indexed by addresses. Hence, cells in the $heap$ are accessible using an array lookup operator _[_] and the heap is modified using the array update operator $_\{_ \leftarrow _\}$. As usual, we use v to refer to the values of variables in the pre-state (before the statement is executed) and v' to refer to the values of the same variable is the post-state (after the statement has been executed). We use ℓ to denote program lines. In order to simplify the notation, we assume that all variables which are not explicitly stated in the transition relation are preserved. For each SPL statement, we now present the semantics of the statement by defining the formula that describes its transition relation:

Assignments: The transition relation for a variable assignment consists of the update of the program counter for the running thread and the corresponding modification to the variable being assigned.

Statement	Transition relation
$\ell_1 :\ \ \mathtt{v} := \mathtt{w}$ $\ell_2 :\ \ \cdots$	$pc(T) = \ell_1 \wedge pc'(T) = \ell_2 \wedge \mathtt{v}' = \mathtt{w}$

Pointer access: Cell fields are accessible through the pointer operator$\rightarrow$. There are two possible scenarios for the use of pointers, depending on whether the statement accesses or modifies a cell field. We present now the semantics for both cases. The first case corresponds to the access of a cell field through an address pointer. The second case corresponds to the modification of a cell field using an address pointer. Note how, in the second case, all cells (except the one pointed by `b`) remain unchanged. Also, all fields of the cell pointed by `b` remain unmodified except for $\mathtt{field_n}$.

Statement	Transition relation
ℓ_1 : $\mathtt{v} := \mathtt{a}{\rightarrow}\mathtt{field}$	$pc(T) = \ell_1 \wedge pc'(T) = \ell_2 \wedge$
	$\mathtt{v}' = heap[\mathtt{a}].\mathtt{field}$
ℓ_2 : $\cdots$	
ℓ_1 : $\mathtt{b}{\rightarrow}\mathtt{field_n} := \mathtt{a}$	$pc(T) = \ell_1 \wedge pc'(T) = \ell_2 \quad \wedge$
	$heap' = heap\{\mathtt{b} \leftarrow c\} \quad \wedge$
	$c.\mathtt{field_n} = \mathtt{a} \quad \wedge$
	$\bigwedge_{i \neq \mathtt{n}} c.\mathtt{field}_i = heap[\mathtt{b}].\mathtt{field}_i$
ℓ_2 : $\cdots$	

No operation: The no operation statement performs no change at all except from updating the program counter of the executing thread.

Statement	Transition relation
ℓ_1 : **skip**	$pc(T) = \ell_1 \wedge pc'(T) = \ell_2$
ℓ_2 : $\cdots$	

Conditionals: We present now the two possible kinds of conditional statements in SPL. In the first case, if condition c does not hold, the execution proceeds from the statement following the end of the conditional.

Statement	Transition relation
ℓ_1 : **if** c **then**	$(pc(T) = \ell_1 \wedge \mathtt{c} \wedge pc'(T) = \ell_2) \ \vee$
	$(pc(T) = \ell_1 \wedge \neg\mathtt{c} \wedge pc'(T) = \ell_{n+1})$
ℓ_2 : $\cdots$	
$\vdots$	
ℓ_n : **end if**	
ℓ_{n+1} : $\cdots$	

In the second case, if condition c does not hold, the execution continues at the first statement in the **else** section of the conditional statement.

Statement	Transition relation	
ℓ_1 : **if** c **then**	$(pc(T) = \ell_1 \wedge \mathsf{c} \wedge pc'(T) = \ell_2) \vee$ $(pc(T) = \ell_1 \wedge \neg\mathsf{c} \wedge pc'(T) = \ell_{n+1})$	for line ℓ_1
ℓ_2 : $\cdots$		
$\vdots$		
ℓ_n : **else**	$pc(T) = \ell_n \wedge pc'(T) = \ell_{m+1}$	for line ℓ_n
ℓ_{n+1} : $\cdots$		
$\vdots$		
ℓ_m : **end if**		
ℓ_{m+1} : $\cdots$		

Loops: We consider the only loop statement available in SPL which executes the statements in the body as long as the loop condition holds.

Statement	Transition relation	
ℓ_1 : **while** c **do**	$(pc(T) = \ell_1 \wedge \mathsf{c} \wedge pc'(T) = \ell_2) \vee$	
ℓ_2 : $\cdots$	$(pc(T) = \ell_1 \wedge \neg\mathsf{c} \wedge pc'(T) = \ell_{n+1})$	for line ℓ_1
$\vdots$		
ℓ_n : **end while**	$pc(T) = \ell_n \wedge pc'(T) = \ell_1$	for line ℓ_n
ℓ_{n+1} : $\cdots$		

Non deterministic choice: The transition relation for the non-deterministic choice statement can be expressed as follows:

Statement	Transition relation
ℓ_1 : **nondet choice**	$pc(T) = \ell_1 \wedge \bigvee_{i=2..n} pc'(T) = \ell_i$
ℓ_2 : $\cdots$	
ℓ_3 : **or** $\cdots$	
$\vdots$	
ℓ_n : **or** $\cdots$	
ℓ_{n+1} : **end choice**	

Wait on condition: When waiting on a condition, the program execution does not progress until

the condition is satisfied.

Statement	Transition relation
ℓ_1 : **await** c ℓ_2 : $\cdots$	$pc(T) = \ell_1 \wedge c \wedge pc'(T) = \ell_2$

We can relate to this statement the *lock* and *unlock* operations used over locks. Even though these are not SPL statements, as they will be widely used in the rest of the chapters, we describe here the transition relation associated with these two functions.

Statement	Transition relation
ℓ_1 : $lock(\mathtt{l})$ ℓ_2 : $\cdots$	$pc(T) = \ell_1 \wedge \mathtt{l} = \emptyset \wedge \mathtt{l}' = T \wedge pc'(T) = \ell_2$
ℓ_1 : $unlock(\mathtt{l})$ ℓ_2 : $\cdots$	$pc(T) = \ell_1 \wedge \mathtt{l}' = \emptyset \wedge pc'(T) = \ell_2$

Critical sections: The statements **noncritical** and **critical** are just schematic statements to denote the presence of a non-critical or a critical section, respectively. Hence, their transition relations are similar to the one of a no operation statement.

Statement	Transition relation
ℓ_1 : **noncritical** ℓ_2 : $\cdots$	$pc(T) = \ell_1 \wedge pc'(T) = \ell_2$
ℓ_1 : **critical** ℓ_2 : $\cdots$	$pc(T) = \ell_1 \wedge pc'(T) = \ell_2$

Procedure calls: We need to consider two statements related to procedure calls. The **call** statement which invokes a procedure and the **return** statement which returns a value from a procedure.

Statement	Transition relation
ℓ_1 : $\mathtt{v} :=$ **call** $\mathtt{P}(\mathtt{v_1}, ...\mathtt{v_n})$	For **call** statement:
ℓ_2 : $\cdots$	$pc(T) = \ell_1 \wedge pc'(T) = \ell_p \quad \wedge$
$\vdots$	$\mathtt{P} :: \mathtt{ret} = \ell_2 \quad \wedge$
procedure $\mathtt{P}(\mathtt{arg_1}, ..., \mathtt{arg_n})$	$\bigwedge_{i=1..\mathtt{n}} \mathtt{P} :: \mathtt{arg_i} = \mathtt{v_i}$
begin	
ℓ_p : $\cdots$	For **return** statement:
$\vdots$	$pc(T) = \ell_{p+m} \wedge pc'(T) = \mathtt{ret} \quad \wedge$
ℓ_{p+m} : **return(w)**	$\mathtt{v}' = \mathtt{w}$

We use the auxiliary local variable `ret` to store the program location where procedure `P` should return. As expected, this approach does not support recursion but it supports multiple procedure calls.

Atomic sections: Atomic sections group a set of SPL statements into a single statement which is executed in a single step of execution. Hence, the transition relation for an atomic statement is the combination of the transition relations of all statements that are part of it. For example, consider the case of an atomic section $\langle S_1; S_2 \rangle$ which consists of a sequence of two statements S_1 and S_2. Then, the transition relation $\rho_{\langle\rangle}(V, V'')$ corresponds to a formula which states a relation between the values of variables in the pre-state V and the post-state V''. These valuations are obtained from the transition relations $\rho_{S_1}(V, V')$ and $\rho_{S_2}(V', V'')$ which are associated to the statements S_1 and S_2 respectively. That is:

$$\rho_{\langle\rangle}(V, V'') \quad \stackrel{\text{def}}{=} \quad \rho_{S_1}(V, V') \wedge \rho_{S_2}(V', V'')$$

2.1.3 Ghost Code

When performing program verification it is sometimes convenient to enlarge the set of program variables with auxiliary variables, called *ghost variables*, to store interesting information about the history of the computation. If extra notation fields are added, they are called *ghost fields*. If code is added to the algorithm, it is called *ghost code*. We will usually refer to all of them simply as *ghost code*.

The idea to use ghost code in a program to ease specification is well known [139]. Ghost code is used for verification only and its execution does not interfere with the actual program code.

The variables added as ghost code are not allowed in the enabling condition of statements occurring in the actual program, and are only used to update other ghost variables in ghost code. Ghost instructions are executed atomically together with the real code they annotate but they do not affect the original control flow of the program. In particular, ghost variables do not occur in the normal program code nor in assignments to program variables.

We will use the symbol @ to denote a ghost field which is part of a class. We uses classes to

describe the signature of concrete program structures. Similarly, we use dashed boxes around the ghost code added to program statements.

Example 2.3
Fig. 2.3 presents a variation of the program SETMUTEX in which some ghost code has been added. We have added a ghost variable $tids$ of type $Set\langle Tid \rangle$ to keep track of the set of thread identifiers that have required access to the critical section. The ghost variable $tids$ is updated at two points. First, when a thread requires access to the critical section by asking for a new ticket at line 3. At this point, the thread identifier of the running thread (me) is inserted into the set $tids$. Second, the set $tids$ is updated at line 6, when a thread leaves the critical section. At this point, the running thread removes its thread identifier from the set to indicate that the running thread has effectively left the critical section. ⌟

2.2 Concurrent Data Structures

Multiprocessor shared-memory systems are capable of running multiple threads of execution simultaneously, where all threads communicate and synchronize using shared data. Concurrent data structures are much more difficult to design than sequential data structures due to the interleaving between the executions. Each possible interleaving may lead to a different, and possibly unexpected, behavior. In this section we build on some of the aspects described in [156] on concurrent systems. We provide an overview of the challenges involved in the design of concurrent data structures.

The main sources of difficulty when designing these kind of data structures comes from concurrent interaction. As threads may run concurrently on different processors, they are subject

```
   global
     Int avail := 0
     Set⟨Int⟩ bag := ∅
     Set⟨Tid⟩ tids := ∅

   procedure SETMUTEX
     Int ticket := 0
   begin
1: while true do
2:     noncritical
3:     ⟨ ticket := avail++
         bag.add(ticket) ⟩
       tids := tids ∪ {me}
4:     await (bag.min == ticket)
5:     critical
6:     bag.remove(ticket)
       tids := tids − {me}
7: end while
   end procedure
```

Figure 2.3: Program INTMUTEX updated with ghost code.

to operating system scheduling policies, page fault, interruptions, etc. Hence, reasoning about the computation must consider how all different threads can be arbitrarily interleaved.

Example 2.4
Imagine we want to implement a *shared counter*. A *shared counter* consists of a shared variable that is concurrently incremented by many threads. A sequential version of this algorithm is depicted in Fig. 2.4(a). This version just fetches the value from counter c and increments it by one, returning the previous value. If many threads run this implementation concurrently, then the result may not be the expected one. For instance, consider the scenario at which c is 0 and two threads execute this implementation concurrently. Then, there is the possibility that both threads read the value 0 from memory and thus both return 0 as a result, which is clearly wrong.

To avoid this problem, a common solution is the use of mutual exclusion locks (usually simply called *mutexes* or *locks*). A lock is a construction with the property that, at any given time, no thread owns the lock or the lock is owned by a single thread. If a thread T_1 wants to acquire a lock owned by thread T_2, then T_1 must wait until T_2 releases the lock.

A correct implementation of the *shared counter* using locks is shown in Fig. 2.4(b). This new version uses a lock to allow access to the critical section in mutual exclusion. ⌟

We refer to a program section as *critical* when it provides access to a shared data structure which must not be concurrently executed by more that one thread simultaneously. In the example above, the lock prevents two different threads from accessing the critical section simultaneously. While this is a correct solution, this version lacks good performance. It is usually easier to achieve a correct concurrent implementation from a sequential one by protecting large blocks of code with locks. However, the performance of such implementation is usually very poor.

In the following, we describe some concepts to design faster concurrent solutions as well as some techniques for verifying the correctness of concurrent data structures.

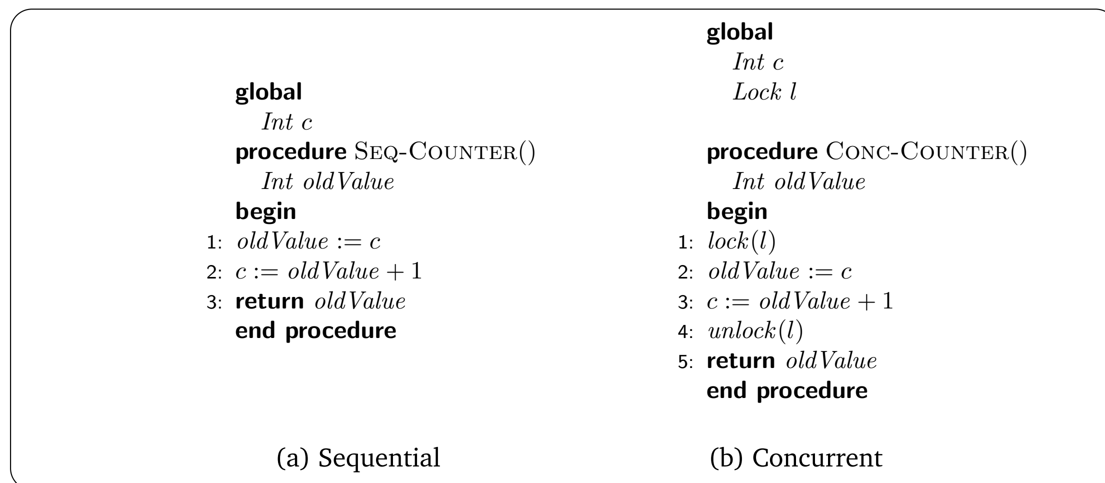


Figure 2.4: Possible implementation of a *shared variable*.

2.2.1 Performance Concerns

The *speedup* of a program when run on P different processors is the relation between its execution time on a single processor and its execution time on P processors concurrently. The speedup provides a way to measure how efficiently a program uses the processors on which it runs. Ideally, we would like to have linear speedup, but this is not always possible. Data structures whose speedup grows with P are called *scalable*. When designing concurrent data structures, scalability must be taken into account. Naïve implementations, as CONC-COUNTER will surely undermine scalability.

In Example 2.4 it is easy to see that the lock introduces a sequential bottleneck: at every instant, only a single thread is allowed to perform useful work. Hence, reducing the size of code sections that are forced to execute sequentially is crucial in order to achieve good performance. In the context of locking, we are interested in:

1. reducing the number of acquired locks, and
2. reducing the lock granularity.

Lock *granularity* is essentially the number of instructions executed while holding a lock. The fewer instructions, the finer lock granularity. Implementations like the shared counter considered above represent an example of a coarse-grain solution.

Another aspect to keep in mind is *memory contention*. Memory contention refers to the slowdown in memory accesses as a result of multiple threads concurrently trying to access the same memory location. For instance, if the lock protecting a critical section is implemented in a single memory location, then a thread trying to access that section must continuously try to access the memory portion where the lock is stored. This problem may be solved if we consider *cache-coherent* multiprocessors. However, using this technique may lead to long waiting times each time a location needs to be modified. As a result, the exclusive ownership of the cache line containing the lock must be repeatedly transfered from one processor to other.

A final problem with lock-based solutions is that, if a thread holding a lock is delayed then all other threads that intend to acquire the same lock are also delayed. This phenomenon is known as *blocking* and it is quite common on systems with many threads per processor. A possible solution is provided by *non-blocking* algorithms. Non-blocking programs do not use locks and thus the delay of a thread does not affect the delay of other threads. In the following section we describe the main characteristics of lock-based and non-blocking systems.

2.2.2 Synchronization Techniques

In Example 2.4, program CONC-COUNTER uses a lock to prevent many threads to access the critical section simultaneously. However, sometimes not using locks at all is a better solution. Here, we describe the two major techniques for accomplishing mutual exclusion on modern concurrent systems: blocking techniques and non-blocking techniques.

Blocking Techniques

In many cases memory contention and sequential bottlenecks can be reduced by reducing the granularity of locking schemes. In fine-grained locking approaches, multiple locks of small

granularity are used to protect the critical sections that modify the data structure. The idea is to maximize time that threads are allowed to run in parallel, as far as they do not require to access the same portions of the data structure. This approach can also be used to reduce the contention for individual memory locations. In some cases, this solution is very natural, as in the case of hash maps. In hash maps, values are often hashed to independent different buckets, in order to reduce the number of potential conflicts. Hence, simply placing individual locks on each bucket reduces granularity easily.

On the other hand, in cases like the shared counter, it is not quite intuitive how contention and sequential bottleneck can be reduced since, abstractly, all operations modify the same part of the data structure. One approach to deal with contention is based on spreading each operation to access the counter at a separate time interval. A widely used technique to accomplish this is *backoff* [5]. However, even reducing the contention, the lock-based implementation of the shared counter still lacks effective parallelism and hence it is not scalable. Therefore, more powerful techniques are required.

A possible solution is the use of a method known as *combining trees* [91, 93, 103, 212]. *Combining trees* employ a binary tree with one leaf for each thread. The root of the tree keeps the current value of the counter and intermediate nodes are used to coordinate the access to the root. The idea is that while a thread climbs up to the root of the tree, it combines the effect of its operations with the operations of other threads. Hence, every time two threads meet on an internal node, their operations are combined into a single operation. Then, one thread waits in the node until a return value is delivered to it while the other thread proceeds further up the tree carrying the operation obtained from the combination.

In the particular case of our shared counter, the winner thread that reaches the root of the tree modifies the counter in a single atomic operation, performing the action of all combined operations. Then, this winning thread traverses down the tree, delivering the return value to each thread waiting on an internal node. The values returned are distributed in such a way that the final effect is as if all operations were executed one after the other at the moment the counter in the root was modified.

Threads waiting on internal nodes repeatedly read a memory location, waiting for the return value. This sort of waiting is known as *spinning*. An important consequence in the case of a cache-coherent multiprocessor is that the accessed location resides in the local cache of the processor of the waiting thread. Following this approach, no extra traffic is generated. This waiting technique, known as *local spinning*, is very important to achieve scalable performance [151].

A drawback of the combining tree method is that if coordination between threads going up and down the tree is done incorrectly, it may lead to deadlocks. A *deadlock* is a situation at which two or more threads are executing tasks such that all of them are blocked in a circular fashion, so none of them can progress. Deadlock avoidance is a crucial technique to guarantee when designing correct and efficient blocking concurrent data structures and algorithms. When designing blocking implementations, the number of locks to be taken is a key factor to consider. Enough locks should be used as to ensure absence of data races while minimizing blocking, allowing threads to perform concurrent operations in parallel.

Non-blocking Techniques

Non-blocking implementations are designed in order to solve some of the inconveniences present on blocking implementations, by avoiding the use of locks. To formalize the idea of non-blocking algorithms, some non-blocking progress conditions have been proposed in the literature. The best known conditions are known as *wait-freedom*, *lock-freedom* and *obstruction-freedom*. A wait-free operation [102, 133] must terminate on its own, after a finite number of steps, no matter the behavior of the competing concurrent operations. A lock-free operation [102] guarantees that after a finite number of steps of a given thread, some operation terminates: maybe a given operation of interest, or maybe the operation performed by some other thread. Finally, an obstruction-free operation [104] guarantees that any given operation of a thread terminates after a finite number of its own steps, assuming that it runs continuously without interruptions. There exists a relation between all these properties. Wait-freedom is stronger than lock-freedom, and in turn lock-freedom is stronger than obstruction-freedom. Clearly, strong progress conditions are desired over weaker ones. However, weaker conditions are in general simpler and more efficient to design and verify.

Non-blocking algorithms use no locks. This means, one needs to find a different way to implement the concurrent version of the shared counter. The work in [79] (extended to shared memory by [102] and [138]) shows that it is not possible to implement a concurrent shared counter using just *load* and *store* instructions. The problem can be solved using a hardware operation that atomically combines both a *load* and a *store*. In fact, most modern multiprocessors provide one kind of such synchronization primitives. The most well known are *compare-and-swap* (CAS) [113,116,194] and *load-linked/store-conditional* (LL/SC) [114,122,190]. Fig. 2.5 describes the behaviour of the CAS operation as an algorithm. This algorithm receives as arguments a memory location L and two values: E and N. The operation is performed atomically: if the value stored at location L matches E, then the value in L is replaced with N, and the operation returns $true$. If the value at L does not match E, *false* is returned, and the value at L remains unchanged.

In [102] it is shown that instructions such as CAS and LL/SC are universal in the following sense. For any data structure, it exists a wait-free implementation in a system that supports these instructions.

Example 2.5

Using the CAS instruction it is now possible to implement a non-blocking version of the shared counter program presented in Example 2.4. The idea is to perform the operation locally and then use CAS to atomically update the result. In case the read value does not coincide with the

```
   procedure CAS(L, E, N) : Bool
1: if *L = E then
2:     *L := N
3:     return true
4: else
5:     return false
6: end if
   end procedure
```

Figure 2.5: Description of a CAS operation.

```
   procedure NonBlock-Counter()
      Int oldValue
      Int newValue
   begin
1: repeat
2:     oldValue := c
3:     newValue := oldValue + 1
4: until CAS(&c, oldValue, newValue)
5: return oldValue
   end procedure
```

Figure 2.6: Non-blocking shared counter implementation using CAS.

value required in the CAS, the CAS operation returns *false* performing no modification to the counter. Hence, the operation needs to be retried until a successful call to CAS is accomplished. Fig. 2.6 shows a non-blocking implementation of the shared counter using CAS. Because the CAS can only fail due to another succeeding *fetch and increment* operation, the implementation is clearly lock-free. However, it is not wait-free as the *fetch and increment* operation successfully accomplished by other threads can continuously prevent a given CAS to succeed. ⌟

The example shown here is simple, but in general, designing non-blocking algorithms is more complex than blocking solutions, since the complexity is greatly reduced when a thread can use a lock to prevent other threads from interfering while it performs some action. If locks are not used, then the algorithm must be designed in order to be correct despite the actions performed by all other concurrent threads. Currently, in modern architectures, using non-blocking algorithms that maximize performance requires the use of complicated and error-prone techniques.

2.2.3 Verification Techniques

In our case, it is quite easy to see that the lock-based implementation of the shared counter behaves exactly as the sequential implementation. However, if we consider a more complicated data structure, such as a binary tree for instance, then the design and verification is significantly more difficult. Due to the huge number of interleaving, it is quite easy to introduce mistakes and propose an incorrect implementation. Hence, it becomes imperative to rigorously prove that a particular design correctly implements the desired concurrent data type.

Operations on a sequential data structure are executed one after the other, in order. Then, we simply require that the resulting sequence of operations respects the sequential semantics denoted by the set of states and transitions, as described above. On the contrary, in the case of concurrent data structures, operations are not necessarily totally ordered. Typical correctness criteria for concurrent implementations usually requires the existence of some total order of operations that respects the sequential semantics.

A common condition is Lamport's *sequential consistency* [134]. This condition requires that the total order preserves the order of the operations run by each thread. However, this condition has a drawback: a data structure constructed by sequential consistent components may not be sequential consistent.

Another widely used concept is *linearizability* [108, 109], a variation of the concept of *serializability* [25] used in database transactions. Linearizability requires that:

1. the data structure is sequentially consistent, and
2. the total ordering which makes it sequentially consistent, respects the real-time ordering between the operations in the execution.

A way of thinking about linearizability is that it requires the user to identify specific points within the algorithms, called *linearization points*, such that if we order the operations according to the execution turn of their linearization points , the resulting order satisfies the desired sequential semantics of the data structure.

Example 2.6
It is easy to see that CONC-COUNTER presented in Example 2.4 is linearizable. We just need to define the point after the increment of c as the linearization point. In the case of NONBLOCK-COUNTER presented in Example 2.5, the argument is similar, except that we must define the linearization point considering the semantics of CAS. ⌟

The intuitive simplicity and the modularity of linearizability makes it a very popular correctness condition. Although most of the concurrent data structures can be shown to be linearizable, on some situations, better performance and scalability can be achieved by considering a weaker condition: *quiescent consistency*.

The quiescent consistency condition [11] relaxes the restriction that the total order of operations needs to respect the real-time order of the executed operations, but it requires that every operation executed after a quiescent state must be ordered after every operation executed before the quiescent state. A state is said quiescent if no operations are in progress.

In general, obtaining a formal proof of the correctness for the implementation of a data structure requires:

- a mathematical method for specifying correctness requirements,
- an accurate model of the data structure implementation, and
- a proof system to show that the implementation is correct.

For instance, most linearizability arguments in the literature treat some of these aspects in an informal way. This makes proofs easier to follow and understand. However, the use of some informal description is prone to introduce some error (based for example in implicit assumptions), miss some cases or make some incorrect inferences. On the other hand, rigorous proofs usually contain a great amount of details regarding trivial properties that makes them difficult to write and tedious to read. Hence, computer assisted methods are desired for the formal verification of concurrent implementations. One approach is based on the use of theorem provers to aid in the verification. Another approach consists in the use of model checking. Model checking tools exhaustively verify all possible executions of an implementation, to ensure that all reachable states meet specified correctness conditions. However, some limitations exists on both approaches. Theorem provers usually require significant human insight, while model checking is generally limited by the number of states it can consider. Therefore, a verification method involving as few human interaction as possible, while being able to verify systems with a possible infinite number of states is desired.

2.2.4 Synchronization Elements

In this section, we describe some basic mechanisms commonly used to achieve correct concurrent implementations: locks, barriers and transactional synchronization mechanisms. Locks and barriers are traditional low level synchronization mechanisms used to prevent some interleaving to happen. For instance, preventing two different threads to access the same section of code at the same time. On the other hand, transactional synchronization mechanisms are used to hide the complex reasoning required to design concurrent data algorithms, letting programmers to think in a more sequential fashion.

As explained before, locks are a low level mechanism used to prevent processes to access the same region of memory concurrently. A key issue when designing lock based solutions is what to do when a thread tries to get a lock that is already owned by other thread. A possibility is to let threads keep on trying to get the lock. Locks based on this technique are called *spinlocks*. A simple spinlock may use a primitive such as CAS to atomically change the value of a lock from unowned to owned. However, such spinning may cause heavy contention for the lock. An alternative to avoid contention is the use of exponential backoff. In exponential backoff [5] a thread that fails in its attempt of getting a lock waits for some time before a new attempt. With every failed attempt, the waiting time is increased. The idea is that threads will spread themselves out in time, resulting in a reduction of contention and memory traffic.

A disadvantage of exponential backoff is that it may happen that a lock remains unlocked for a long time, since all interested threads have been backed-off too much in time. A possible solution to this problem may consist in the use of a queue of interested threads. Locks based on this approach are known as *queuelocks*. Some implementations of queuelocks based on arrays are introduced in [7, 94] and then improved using list-based MCS queue locks [151] and CLH queuelocks [59, 141].

Queuelocks also come in many flavors. There are abortable implementations of queuelocks where a thread can give up if it is delayed beyond some time limit [184, 185] or if they just fall into deadlock. On the other hand, preemptive-safe locks [154] ensure that an enqueued preempted thread does not prevent the lock to be taken by another running thread.

In some cases, we would like to have locks letting multiple readers access the concurrent data structure. A reader is a process that only extracts information from the data structure, without modifying it. Such locks are known as *reader-writer* locks. There exist many kinds of these locks. For instance, reader-writer queuelock algorithms [152] use a MCS queuelock, a counter for reads and a special pointer for writes. In [126] readers remove themselves from the lock's queue, keeping a double-linked list and some special locks on the list's nodes. In this case, when a thread removes itself from the list, it acquires a small lock on its neighbor nodes, and redirects the pointers removing itself from the list.

The reader-writer approach can be also generalized to *group mutual exclusion* or *room synchronization*. Under this approach, operations are divided into groups. Operations within the same group can be performed simultaneously with each other, while operations belonging to different groups cannot be executed concurrently. An application of such technique, for instance, could classify the PUSH and POP operations over stacks on different groups [30]. Group mutual exclusion is introduced in [120] and implementations for fetch and increment counters (as the example shown at the beginning of this chapter) are described in [30, 124].

Another mechanism are barriers. A barrier stops all threads at a given point, allowing them to

proceed only after all threads have reached that point. Barriers are used when the access to the data structure is layered, preventing layer overlapping between different phases.

A barrier can simply be implemented using a counter to keep track of the number of threads that have achieved the barrier position. The counter is initialized with the total number of threads that must synchronize. Then, every time a thread reaches the barrier, it decrements the counter. Once the counter has reached zero, all threads are allowed to proceed. This approach still displays the problem of contention, as many threads may be spinning, waiting for the barrier to reach zero and let them proceed. Therefore, special implementations of barriers exist to alleviate this problem, making threads spin on different locations [100, 115, 186]. An alternative approach consists in implementing barriers using diffusing computation trees [66].

The main purpose for using locks in concurrent programming is to let threads modify multiple memory locations atomically in such a way that no partial result of the computation can be observed by other threads. In this aspect, transactional memory is a mechanism that lets programmers model sections of the code accessing multiple memory locations as a single atomic operation. The use of transactional mechanisms is inspired on the idea of transactions in databases, even though that the problem in memory management is slightly different to the one existing on databases.

An example of a transactional memory mechanism for concurrent data structures is *optimistic concurrency control* [130]. This approach uses a global lock which is held for a short time at the end of the transaction. However, such a lock is a cause of sequential bottleneck. Ideally, transaction synchronization should be accomplished without the use of global locks. For example, transactions accessing disjoint sections of the memory should not synchronize with each other at all. A hardware-based transactional memory mechanism was first proposed in [106]. An extension to this idea is *lock elision* [174, 175] where the hardware can automatically translate accesses to critical section into transactions that can be executed in parallel.

Despite the effort, up to this moment no hardware support for transactional memory has been developed. Even though, many software based transactional memory approaches have been proposed [98, 99, 105, 187].

2.3 Signatures and Theories

We use many-sorted first order logic to define the theories presented in Chapter 6, 7 and 8 as a combination of theories. Hence, we begin with a brief description of the basic notation and concepts.

A signature Σ is a triple (S, F, P) where S is a set of sorts, F is a set of function symbols and P is a set of predicate symbols constructed using the sorts in S. If $\Sigma_1 = (S_1, F_1, P_1)$ and $\Sigma_2 = (S_2, F_2, P_2)$ are two signatures, we define their union $\Sigma_1 \cup \Sigma_2 = (S_1 \cup S_2, F_1 \cup F_2, P_1 \cup P_2)$. Similarly we say that $\Sigma_1 \subseteq \Sigma_2$ when $S_1 \subseteq S_2$, $F_1 \subseteq F_2$ and $P_1 \subseteq P_2$.

Example 2.7
We can describe the signature of the theory of naturals with addition, equality and total order

with the following signature:

$$\Sigma_{\mathsf{nat}} = \begin{pmatrix} \{\mathsf{nat}\} & , \\ \{0 : \mathsf{nat}, s : \mathsf{nat} \to \mathsf{nat}, \ + : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}\} & , \\ \{=: \mathsf{nat} \times \mathsf{nat}, \ \ <: \mathsf{nat} \times \mathsf{nat}\} & \end{pmatrix}$$ ⌟

Example 2.8
Consider now the signature of the theory of sets of integers with union, set difference and minimum:

$$\Sigma_{\mathsf{setnat}} = \begin{pmatrix} \{ \ \mathsf{nat}, \mathsf{setnat} \ \} & & & , \\ \{ \quad \emptyset & : & \mathsf{setnat}, & \\ \quad \{_\} & : & \mathsf{nat} \to \mathsf{setnat}, & \\ \quad \cup & : & \mathsf{setnat} \times \mathsf{setnat} \to \mathsf{setnat}, & \\ \quad - & : & \mathsf{setnat} \times \mathsf{setnat} \to \mathsf{setnat}, & \\ \quad min & : & \mathsf{setnat} \to \mathsf{nat} \ \} & , \\ \{ \quad = & : & \mathsf{setnat} \times \mathsf{setnat} \ \} & \end{pmatrix}$$

For analyzing program SETMUTEX presented in Example 2.2 we could use a theory whose signature may be the union of Σ_{setnat}, presented here, and Σ_{nat}, presented in Example 2.7. ⌟

Given a signature Σ, we assume the standard notions of Σ-term, Σ-literal and Σ-formula. A literal is *flat* if it is of the form $x = y$, $x \neq y$, $x = f(y_1, \dots, y_n)$, $p(y_1, \dots, y_n)$ or $\neg p(y_1, \dots, y_n)$, where $x, y, y_1, \dots, y_n$ are variables, f is a function symbol and p is a predicate symbol. Every literal can be converted into an equivalent conjunction of flat literals, simply by introducing fresh variables for each of the sub-terms of a non-flat literal, and equating the variable with the corresponding sub-term. For example $p(t(x, y))$ becomes $p(z) \wedge z = t(x, y)$ for a fresh variable z of the appropriate sort. The formula $\exists * .p(t(x, y))$ is equivalent to $\exists * . \Big(p(z) \wedge z = t(x, y)\Big)$.

Given a term t, we use $V_\sigma(t)$ for the set of variables of sort σ occurring in t and $C_\sigma(t)$ for the set of constants of sort σ occurring in t. Similarly, given a formula φ, we use $V_\sigma(\varphi)$ for the set of variables of sort σ occurring in φ and $C_\sigma(\varphi)$ for the set of constants of sort σ occurring in φ.

A Σ-interpretation is a map from symbols in Σ to values (see, e.g., [83]). Let $\mathcal{A}$ be a Σ-interpretation over V. Then, for each $s \in S$, $\mathcal{A}_s$ is a set of elements called the domain of s. For each symbol $f : s_1 \times \cdots \times s_n \to s$ in F, $\mathcal{A}_f$ is a function that goes from elements in the tuple $\mathcal{A}_{s_1} \times \cdots \times \mathcal{A}_{s_n}$ to elements in $\mathcal{A}_s$. For each symbol $p : s_1 \times \cdots \times s_n$ in P, $\mathcal{A}_p$ is a relation over elements of $\mathcal{A}_{s_1} \times \cdots \times \mathcal{A}_{s_n}$. Given a model $\mathcal{A}$ and a sort σ, $V_\sigma^{\mathcal{A}}$ denotes the set of values interpreted by model $\mathcal{A}$ for all variables of sort σ.

A Σ-structure is a Σ-interpretation over an empty set of variables. A Σ-formula over a set V of variables is satisfiable whenever it is true in some Σ-interpretation over V. Two Σ-formulas φ and ψ over a set V of variables are equivalent is their truth values, in symbols $\varphi^{\mathcal{A}}$ and $\psi^{\mathcal{A}}$, are identical (i.e. $\varphi^{\mathcal{A}} = \psi^{\mathcal{A}}$) for all Σ-interpretations over V.

Let Ω be a signature, $\mathcal{A}$ a Ω-interpretation over a set V of variables, $\Sigma \subseteq \Omega$ and $U \subseteq V$. $\mathcal{A}^{\Sigma, U}$ denotes the interpretation obtained from $\mathcal{A}$ restricting it to interpret only the symbols in Σ and

the variables in U. We use $\mathcal{A}^{\Sigma}$ to denote $\mathcal{A}^{\Sigma,\emptyset}$. If Φ is a set of Σ-formulas, we use $Mod^{\Sigma}(\Phi)$ to denote the class of many-sorted Σ-structures satisfying all the formulas in the set Φ. A Σ-theory is a pair $(\Sigma, \mathbf{A})$ where Σ is a signature and $\mathbf{A}$ is a class of Σ-structures. Given a theory $T = (\Sigma, \mathbf{A})$, a T-interpretation is a Σ-interpretation $\mathcal{A}$ such that $\mathcal{A}^{\Sigma} \in \mathbf{A}$. Given a Σ-theory T, a Σ-formula φ over a set of variables V is T-satisfiable if it is true on a T-interpretation over V. We define $F_{\Sigma}^{(V)}$ as the set of Σ-formula over variables V. Finally, given a sequence of variables $v_1, \ldots, v_n$, we use $v_{0..n}$ to denote such sequence.

Example 2.9
We can now formally define the theory of natural numbers with addition using the Σ_{nat} signature presented in above Example 2.7. Let T_{nat} be the theory of natural numbers defined by:

$$T_{\mathsf{nat}} = (\Sigma_{\mathsf{nat}}, \mathbf{A}_{\mathsf{nat}})$$

where $\mathbf{A}_{\mathsf{nat}}$ is a class of Σ_{nat}-structures that restricts the models of Σ_{nat} such that every interpretation $\mathcal{A}$ of Σ_{nat} must satisfy that for every variable n, n_1 and n_2 of sort nat:

$$\begin{array}{rcl} 0^{\mathcal{A}} & = & 0 \\ s^{\mathcal{A}}(n) & = & n+1 \\ +^{\mathcal{A}}(n_1, n_2) & = & n_1 + n_2 \\ =^{\mathcal{A}}(n_1, n_2) & \leftrightarrow & n_1 = n_2 \\ <^{\mathcal{A}}(n_1, n_2) & \leftrightarrow & n_1 < n_2 \end{array}$$

┘

Through this work we will usually refer to a *combination of theories*. Let $T_i = (\Sigma_i, \mathbf{A}_i)$ be a theory, for $i = 1, 2$. The combination of theories T_1 and T_2 is the theory $T_1 \oplus T_2 = (\Sigma, \mathbf{A})$ where $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\mathbf{A} = \{\mathcal{A} \mid \mathcal{A}^{\Sigma_1} \in \mathbf{A}_1 \text{ and } \mathcal{A}^{\Sigma_2} \in \mathbf{A}_2\}$. If Φ_i is a set of Σ_i-formulas with no free variables and $T_i = (\Sigma_i, Mod^{\Sigma_i}(\Phi_i))$ for $i = 1, 2$ such that their signature is disjoint (that is, $\Sigma_1 \cap \Sigma_2 = \emptyset$), then $T_1 \oplus T_2 = (\Sigma_1 \cup \Sigma_2, Mod^{\Sigma_1 \cup \Sigma_2}(\Phi_1 \cup \Phi_2))$.

We now present two definitions we will require later when proving that a theory is decidable thanks to a finite or bounded model theorem.

Definition 2.1 (Finite Model Property).
Let Σ be a signature, $S_0 \subseteq S$ be a set of sorts, and T be a Σ-theory. T has the finite model property with respect to S_0 if for every T-satisfiable quantifier-free Σ-formula φ there exists a T-interpretation $\mathcal{A}$ satisfying φ such that for each sort $\sigma \in S_0$, $\mathcal{A}_\sigma$ is finite. ┘

Definition 2.2 (Bounded Model Property).
Theory T has the bounded model property with respect to a set of sorts S if there are computable functions $f_\sigma : F_\Sigma \mapsto \mathbb{N}$ for every sort σ in S such that if φ has a model, then it has a model $\mathcal{B}$ for which $|\mathcal{B}_\sigma| \leq f_\sigma(\varphi)$. ┘

2.4 Model of Computation

We study the verification of temporal properties in parametrized systems, that is, systems composed by an arbitrary number of threads. We now present the general model of computation we will use as well as the main differences between non-parametrized and parametrized systems. The verification framework we propose is based on deductive techniques. That is, starting from a system specification and a temporal property, we advocate for the generation of a set of proof obligations, whose validity imply the conformance of the system to the given temporal property. In the case of safety properties, for example, the formal verification problem for non-parametrized systems takes a system described as a program and a specification of a property expressed as a state predicate. A system satisfies its specification if all states reachable in all the traces of the transition system that models the set of executions of the program satisfy the property.

We will now proceed to present the logic we use to express the temporal properties we verify. Moreover, we will revisit the notion of a non-parametrized transition system and a non-parametrized fair transition system. Then, we will introduce parametrized programs and parametrized fair transition systems, which in turn provide the vocabulary to define parametrized temporal formulas. Finally, we define our notion of correctness of concurrent programs by associating parametrized fair transition systems with parametrized temporal formulas.

2.4.1 Linear Temporal Logic

To express the temporal properties we verify, we use linear temporal logic (LTL). LTL is a modal temporal logic introduced by Amir Pnueli [168] which allows to describe logical properties through time.

LTL formulas are constructed form a finite set of propositional variables AP, the logical unary operator $\neg$ (negation), the logical binary operator $\vee$ (disjunction), the temporal modal unary operator $\bigcirc$ (read as *next*) and the temporal modal binary operator $\mathcal{U}$ (read as *until*). Formally speaking, the set of LTL formulas over the set of propositional variables AP is inductively defined as:

- if $p \in AP$, then p is an LTL formula;
- if φ and ψ are LTL formulas, then $\neg\varphi$, $\varphi \vee \psi$, $\bigcirc\varphi$ and $\varphi\,\mathcal{U}\,\psi$ are LTL formulas.

In addition to the operators presented above, there are other logical and temporal operators which can be defined in terms of the fundamental operators. The additional logical operators are $\wedge$ (conjunction), $\rightarrow$ (implication), $\leftrightarrow$ (equivalence), true and false. On the other hand, the additional temporal operators are the unary operators $\square$ (read as *always*) and $\diamond$ (read as *eventually*), and the binary operators $\mathcal{R}$ (read as *release*) and $\mathcal{W}$ (read as *wait for*).

An LTL formula can be satisfied by an infinite sequence of truth evaluations of variables in AP. These sequences can be viewed as a word on a path of a Kripke structure [127] or as a ω-word over the 2^{AP} alphabet. We can formally define the satisfaction relation $\vDash$ between a ω-word $w = a_0, a_1, a_2, \ldots$ and an LTL formula as:

- $w \vDash p$, if and only if $p \in a_0$. That is, p must be true in the current step of time.
- $w \vDash \neg\varphi$, if and only if $w \nvDash \varphi$. That is, w must not make φ true.

- $w \vDash \varphi \vee \psi$, if and only if $w \vDash \varphi$ or $w \vDash \psi$. That is, w must satisfy φ or ψ.
- $w \vDash \bigcirc\varphi$, if and only if $a_1, a_2, \ldots \vDash \varphi$. That is, in the next step of time, φ must be true.
- $w \vDash \varphi\,\mathcal{U}\,\psi$, if and only if there exists an $i \geq 0$ such that $a_i, a_{i+1}, \ldots \vDash \psi$ and for all k such that $0 \leq k < i$, $a_k, a_{k+1}, \ldots \vDash \varphi$. That is, φ must be true until ψ becomes true. Note that, according to the definition, φ may never be true at all provided ψ is always true.

An LTL formula φ is satisfiable if there exists an ω-word w such that $w \vDash \varphi$. The additional logical operators are defined as usual. The additional temporal operators $\mathcal{R}$, $\diamondsuit$, $\square$ and $\mathcal{W}$ are defined using the $\bigcirc$ and $\mathcal{U}$ operators as follows:

- $\varphi\,\mathcal{R}\,\psi = \neg(\neg\varphi\,\mathcal{U}\,\neg\psi)$. That is, ψ remains true until φ becomes true, bearing in mind that φ may never become true.
- $\diamondsuit\varphi = \mathit{true}\,\mathcal{U}\,\varphi$. That is, eventually φ becomes true.
- $\square\varphi = \mathit{false}\,\mathcal{R}\,\varphi = \neg(\mathit{true}\,\mathcal{U}\,\neg\varphi)$. That is, φ always remains true.
- $\varphi\,\mathcal{W}\,\psi = \psi\,\mathcal{R}\,(\psi \vee \varphi)$. That is, φ remains true until ψ becomes true, or ψ is always true.

2.4.2 Non-Parametrized Fair Transition Systems

The execution of a non-parametrized system is modeled by a *non-parametrized fair transition system*. A non-parametrized fair transition system is expressed as a tuple

$$\mathcal{S} \quad : \quad \langle \Sigma_{\mathsf{prog}}, V, \Theta, \mathcal{T}, \mathcal{J} \rangle$$

where Σ_{prog} is a theory signature, V is a set of program variables, Θ is the initial condition of the system, $\mathcal{T}$ is a set of transition relations and $\mathcal{J}$ is a set of fair transitions. More detailed:

Signature: The signature Σ_{prog} is a first-order signature modeling the data manipulated in a given program. We will use T_{prog} for denoting the theory that allows to reason about formulas in Σ_{prog}.

Program Variables: V denotes a finite set of (typed) variables, whose types are taken from sorts in Σ_{prog}.

Initial Condition: Θ is the initial condition of the transition system, expressed as a first-order assertion over the variables V. Values of V satisfying Θ correspond to initial states of the system.

Transition Relation: $\mathcal{T}$ is a finite set of transitions. Each transition τ in $\mathcal{T}$ is expressed as a first-order formula $\tau(V, V')$ that refers to program variables from V. The set V' contains a fresh copy of v' of each variable v from V. As usual, the variable v' denotes the value of variable v after a transition is taken while v denotes the value before the transition is taken. Moreover, we assume that every system is equipped with an idle transition whose transition relation is $\tau_\epsilon(V, V')$ and which describes the preservation of all system variables, that is, $v = v'$ for all $v \in V$. This idle transition allows to reason only about infinite runs even for deadlocked systems.

Fairness condition: $\mathcal{J}$ is a subset of $\mathcal{T}$, denoting the set of fair transitions.

A *state* is an interpretation of V, which assigns to each program variable a value form the corresponding type. We use $\mathbb{S}$ to denote the set of all possible states. A transition between two states s and s' satisfies a transition relation τ when the combined valuation, that assigns values to variables in V according to s and to variables in V' according to s', satisfies the formula $\tau(V, V')$. In this case, we write $\tau(s, s')$, and we say that the system reaches state s' from state s by taking transition τ. We say that a transition τ is *enabled* in state s if there is a state s' for which $\tau(s, s')$. The *enabling condition* of transition τ is then the formula $\exists V'.\tau(V, V')$. We will generally use the predicate $pres$ over a set of variables to denote variable preservation. That is, if U is a set of variables, then $pres(U)$ is a short for $u' = u$ for all $u \in U$.

Example 2.10
Consider the following program statement:

$$1\!: \ x := x + 1$$

Such statement is modeled by the following formula:

$$pc = 1 \quad \wedge \quad pc' = 2 \quad \wedge \quad x' = x + 1 \quad \wedge \quad pres(V \setminus \{pc, x\}) \qquad \lrcorner$$

Given a transition τ, the state predicate $En(\tau)$, called the enabling condition, captures whether τ can be taken from s, that is, whether there exists a successor state s' such that $\tau(s, s')$. In the example above, $En(\tau)$ is equivalent to $pc = 1$, because the statement "$1\!: \ x := x + 1$" can always be taken if the program is at location 1.

A *run* of a transition system $\mathcal{S}$ is then an infinite sequence $s_0\tau_0s_1\tau_1s_2\ldots$ of states and transitions such that:

1. the first state is initial, that is $s_0 \vDash \Theta$; and
2. all steps are legal. That is, for all i, the relation $\tau_i(s_i, s_{i+1})$ holds. In such case, we say that τ_i is taken at s_i, leading to state s_{i+1}.

A *computation* of $\mathcal{S}$ is a run of $\mathcal{S}$ such that for each transition $\tau \in \mathcal{J}$, if τ is continuously enabled after some point, then τ is taken infinitely many times. We use $\mathcal{L}(\mathcal{S})$ to denote the set of computations of $\mathcal{S}$. Given an LTL formula φ over a propositional vocabulary AP, $\mathcal{L}(\varphi)$ denotes the set of sequences (of elements of 2^{AP}) satisfying φ. Given a computation $\pi : s_0\tau_0s_1\ldots$ of a system $\mathcal{S}$, the corresponding run π^{AP} for a given propositional vocabulary AP is the sequence $P_1P_2\ldots$ with $P_i \subseteq AP$, such that for all instants i:

$$s_i \vDash p_i \text{ for all } p_i \in P_i \qquad \text{and} \qquad s_i \vDash \neg p_i \text{ for all } p_i \notin P_i$$

We use $\mathcal{L}^{AP}(\mathcal{S})$ for the set of sequences of propositions from AP that result from $\mathcal{L}(\mathcal{S})$. A system $\mathcal{S}$ satisfies a temporal formula φ over AP whenever all computations of $\mathcal{S}$ when interpreted over AP satisfy φ, that is $\mathcal{L}^{AP}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$. In this case we write $\mathcal{S} \vDash \varphi$.

2.4.3 Parametrized Concurrent Programs

Programs manipulate data during their execution. In our approach, the reasoning about the program data and its manipulation is handled by specialized decision procedures for specific theories of data.

Example 2.11
Consider once again the program SETMUTEX introduced in Example 2.2. For that program, T_{prog} is the combined theory of Presburger arithmetic, finite sets of integers with minimum, and finite values (to reason about locations). ⌟

The parametrized programs we consider consist of the parallel execution of processes running the same program. It is easy to extend this framework to systems where processes can run programs taken from a finite collection —to model for example, reader/writers. We assume asynchronous interleaving semantics for parallel composition, so precisely one process executes atomically a single statement at a given point it time. The effect of the execution of a statement is fully visible to all other processes after the statement finishes. A program is described by a sequence of statements as described in Section 2.1.1. We use *Locs* to represent the set of program locations. Each statement in SPL is assigned to a program location in the range *Locs* $: 1 \dots L$. Each instruction can manipulate a collection of typed variables partitioned into V_{global}, the set of *global* variables, and V_{local}, the set of *local* variables. A running program contains one shared copy of each global variable, and each thread manipulates its own copy of each local variable. As we mentioned before, there is one special local variable pc of sort *Loc* that stores the program counter of each thread.

Given a parametrized program P, we associate P to an *instance family* $\{\mathcal{S}_P[M]\}$, a collection of non-parametrized transition systems indexed by $M \geq 1$, the number of running threads. This family is called the *parametrized system* corresponding to program P. We use $[M]$ to denote the set $\{0, \dots, M-1\}$ of concrete thread identifiers. Given a value M we refer to $P[M]$ as the *instance* of P with M threads.

For each M, the concrete non-parametrized transition system

$$P[M] \quad : \quad \langle \Sigma_{\text{prog}}, V, \Theta, \mathcal{T}, \mathcal{J} \rangle$$

consists of:

Signature: Signature Σ_{prog}, as in non-parametrized fair transition systems.

Program Variables: The set V of typed variables is:

$$\begin{aligned} V = V_{global} \quad & \cup \ \{v[k] \quad | \ \text{for every } v \in V_{local}, k \in [M]\} \\ & \cup \ \{pc[k] \ | \ \text{for every } k \in [M]\}. \end{aligned}$$

Note that in this case, "$v[k]$" is an indivisible variable name. Alternative names could have been v_k or vk. The set $\{pc[k] \mid k \in [M]\}$ contains one variable of sort *Loc* for each thread id k in $[M]$. The variable $pc[k]$ stores the program counter of thread k. Similarly, for each local program variable v and thread k there is one variable $v[k]$ of the appropriate sort in the set $\{v[k] \mid v \in V_{local} \text{ and } k \in [M]\}$.

Initial Condition: The initial condition Θ is described by two predicates Θ_g (that only refers to variables from V_{global}) and Θ_l (that can refer to variables in V_{global} and V_{local}). These expressions are extracted from the semantics of the programming language. Given a thread identifier $a \in [M]$ for a concrete system $\mathcal{S}_P[M]$, $\Theta_l[a]$ is the initial condition for thread a, obtained by replacing in Θ_l every occurrence of a local variable v from V_{local} for $v[a]$. The initial condition of the concrete transition system $\mathcal{S}_P[M]$ is:

$$\Theta \quad : \quad \Theta_g \wedge \bigwedge_{i \in M} \Theta_l[i]$$

Transition Relation: $\mathcal{T}$ contains a transition $\tau_\ell[a]$ for each program location ℓ and thread identifier a in $[M]$, which are obtained from the semantics of the programming language. The formula $\tau_\ell[a]$ is obtained from τ_ℓ by replacing every occurrence of a local variable v for $v[a]$, and v' for $v[a]'$. Note again that "$v[a]'$" is an indivisible variable name, denoting the primed version of $v[a]$.

Fairness Condition: We consider all transitions fair, that is $\mathcal{J} = \mathcal{T}$.

In general, we will use parenthesis for denoting formal parameters of parametrized formulas and terms. Similarly, we will use brackets for concretes values of parametrized formulas and terms. For instance, "$pc(i) = 5$" is a formula parametrized by a generic thread identifier i, while "$pc[T_1] = 5$" is the same formula but concretized for the specific thread T_1.

Example 2.12
Consider again program SETMUTEX presented in Example 2.2. The instance consisting of two running threads, SETMUTEX[2], contains the following variables:

$$V \quad = \quad \{avail, bag, ticket[0], ticket[1], pc[0], pc[1]\}$$

Global variable $avail$ has type Int, and global variable bag has type $Set\langle Int\rangle$. The instances of local variable $ticket$ for threads 0 and 1, $ticket[0]$ and $ticket[1]$, have type Int. The program counters $pc[0]$ and $pc[1]$ have type $Loc = \{1 \dots 7\}$. The initial condition of SETMUTEX[2] specifies that:

$$\begin{array}{llll} \Theta_g: & avail = 0 \ \wedge \ bag = \emptyset & \Theta_l[0]: & ticket[0] = 0 \ \wedge \ pc[0] = 1 \\ & & \Theta_l[1]: & ticket[1] = 0 \ \wedge \ pc[1] = 1 \end{array} \tag{2.1}$$

There are fourteen transitions in SETMUTEX[2], seven transitions for each thread: $\tau_1[0] \dots \tau_7[0]$ and $\tau_1[1] \dots \tau_7[1]$. The transitions corresponding to thread 0 are:

$$
\begin{array}{llll}
\tau_1[0]: \begin{pmatrix} pc[0] = 1 \quad \wedge \\ pc[0]' = 2 \end{pmatrix} \wedge & & & pres(V \setminus \{pc[0]\}) \\
\tau_2[0]: \begin{pmatrix} pc[0] = 2 \quad \wedge \\ pc[0]' = 3 \end{pmatrix} \wedge & & & pres(V \setminus \{pc[0]\}) \\
\tau_3[0]: \begin{pmatrix} pc[0] = 3 \quad \wedge \\ pc[0]' = 4 \end{pmatrix} \wedge & \begin{pmatrix} ticket[0]' = avail \\ avail' = avail + 1 \\ bag' = bag \cup \{avail\} \end{pmatrix} & \wedge & pres(\{pc[1], ticket[1]\}) \\
\tau_4[0]: \begin{pmatrix} pc[0] = 4 \quad \wedge \\ pc[0]' = 5 \end{pmatrix} \wedge & bag.min = ticket[0] & \wedge & pres(V \setminus \{pc[0]\}) \\
\tau_5[0]: \begin{pmatrix} pc[0] = 5 \quad \wedge \\ pc[0]' = 6 \end{pmatrix} \wedge & & & pres(V \setminus \{pc[0]\}) \\
\tau_6[0]: \begin{pmatrix} pc[0] = 6 \quad \wedge \\ pc[0]' = 7 \end{pmatrix} \wedge & bag' = bag \setminus \{ticket[0]\} & \wedge & pres(V \setminus \{bag, pc[0]\}) \\
\tau_7[0]: \begin{pmatrix} pc[0] = 7 \quad \wedge \\ pc[0]' = 1 \end{pmatrix} \wedge & & & pres(V \setminus \{pc[0]\})
\end{array}
$$

The transitions for thread 1 are analogous. As explained before, the predicate $pres$ encodes the preservation of the values of its argument variables, allowing to describe what a program statement does not modify. For example, in SETMUTEX[2], the predicate $pres(V \setminus \{bag, pc[0]\})$ is:

$$avail' = avail \quad \wedge \quad ticket[0]' = ticket[0] \quad \wedge \quad pc[1]' = pc[1] \quad \wedge \quad ticket[1]' = ticket[1].$$

Using $pres$, the idle transition τ_ϵ implicitly added to every system is $pres(V)$. Note that each transition in SETMUTEX[2] is quantifier free, and involve a combination of theories, including Presburger arithmetic and a theory of finite sets of integers with minimum. ⌟

An alternative model of computation consists of including only one transition per program location, independently of the number of threads. Each transition then would choose one thread and manipulate the local variables for that thread only. There is an advantage in our choice to include a separate transition for each thread and program location. Fairness of a closed fair transition system guarantees that a fair transition must be taken if enabled continuously. In the alternative model of computation, this simple notion of fairness would not guarantee that *each thread* must eventually execute, but only that *each transition* is taken for some thread. Obtaining thread fairness in this alternative model would require to extend the temporal reasoning specifically for this purpose.

2.4.4 Parametrized Fair Transition Systems

A parametrized system consists of a large, unbounded, set of threads that interact with each other using some synchronization primitives. The model presented in this work is based on concurrent

systems that communicate and synchronize through shared memory. Under the assumption of full symmetry, as we do in this work, all thread identifiers of a parametrized system are interchangeable.

Formally, a parametrized transition system associated with a program P is a tuple

$$\mathcal{P}_P \quad : \quad \langle \Sigma_{\mathsf{param}}, V_{\mathsf{param}}, \Theta_{\mathsf{param}}, \mathcal{T}_{\mathsf{param}} \rangle$$

where Σ_{param} is the first-order signature used to reason about data, V_{param} is the set of system variables, Θ_{param} describes the initial condition and $\mathcal{T}_{\mathsf{param}}$ is the parametrized transition relation. We assume all transitions in $\mathcal{T}_{\mathsf{param}}$ to be fair. The intention of parametrized transition systems is not to define program runs directly but to serve as a modeling language for the definition of parametrized formulas and to enable the definition of proof rules and verification diagrams for parametrized systems. We describe each component separately:

Parametrized Program Signature: To capture thread identifiers in an arbitrary instantiation of the parametrized system we introduce a new sort tid interpreted as an unbounded discrete set. The signature Σ_{tid} contains only $=$ and $\neq$, and no constructor. Theory T_{tid} is then the theory of thread identifiers defined over the signature Σ_{tid}. We extend the theory T_{prog}—used to reason about the data in the program— with $Array$, the theory of arrays from [37], with indices from tid and elements ranging over sorts σ of the local variables of program P. We use T_{param} for the union of theories T_{prog}, T_{tid} and $Array$, and Σ_{param} for the combined signature.

Parametrized Program Variables: For each local variable v of type σ in the program, we introduce a variable name a_v of sort $\mathsf{array}\langle\sigma\rangle$, including a_{pc} for the program counter pc. Using the theory of arrays, the expression $a_v(k)$ denotes the elements of sort σ stored in array a_v at position given by expression k of sort tid. The expression $a_v\{k \leftarrow e\}$ corresponds to an array update, and denotes the array that results from a_v by replacing the element at position k with e. For clarity, we abuse notation using $v(k)$ for $a_v(k)$, and $v\{k \leftarrow e\}$ for $a_v\{k \leftarrow e\}$. Note how $v[0]$ is different from $v(k)$: the term $v[0]$ is an atomic term in V (for a concrete system $\mathcal{S}_P[M]$) referring to the local program variable v of a concrete thread with id 0. On the other hand, $v(k)$ is a non-atomic term built using the signature of arrays, where k is a variable (logical variable, not program variable) of sort tid serving as index of the array v. The use we make of $Array$ is very limited: we do not use arithmetic over indices or nested arrays, so the conditions for decidability in [37] are trivially met. Variables of sort tid indexing arrays play a special role, so we classify formulas depending on the number of free variables of sort tid. The parametrized set of program variables with index variables X of sort tid is defined as:

$$V_{\mathsf{param}}(X) = V_{global} \cup \{a_v \mid v \in V_{local}\} \cup \{a_{pc}\} \cup X$$

We use $F_{\mathsf{param}}(X)$ for the set of first-order formulas constructed using predicates and symbols from T_{param} and variables from $V_{\mathsf{param}}(X)$. Given a formula φ from $F_{\mathsf{param}}(X)$ we use $Voc(\varphi)$ to refer to the set of variables of type tid free in φ. We usually refer to $Voc(\varphi)$ as the *vocabulary* of formula φ. Since we restrict to the quantifier-free fragment of $F_{\mathsf{param}}(X)$ then $Voc(\varphi)$ corresponds to the subset of variables from X actually occurring in φ. We say that φ is a 1-index formula if the cardinality of $Voc(\varphi)$ is 1 (similarly for 0, 2, 3, etc).

Parametrized Transition Relation: The set $\mathcal{T}_{\mathsf{param}}$ contains for each statement ℓ in the program one formula $\tau_\ell^{(k)}$ indexed by a fresh tid variable k. These formulas are built using the semantics of the program statements, as for concrete systems except that we now use array reads and updates (to position k) instead of concrete local variable reads and updates. The predicate $pres$ is now defined with array extensional equality for unmodified local variables. Note that there is a finite number of parametrized transitions $\tau_\ell^{(k)}$ for a given program because $\ell \in$ *Locs* and *Locs* is finite.

Parametrized Initial Condition: We similarly define the parametrized initial condition for a given set of thread identifiers X as:

$$\Theta_{\mathsf{param}}(X) \quad : \quad \Theta_g \wedge \bigwedge_{k \in X} \Theta_l(k)$$

where $\Theta_l(k)$ is obtained by replacing every local variable v in Θ_l by $v(k)$.

Example 2.13
Consider once again the program SETMUTEX presented in Example 2.2. The parametrized transition $\tau_4^{(k)}$, for thread k in line 4, is the following formula from $F_{\mathsf{param}}(\{k\})$:

$$\begin{pmatrix} pc(k) = 4 & \wedge \\ pc' = pc\{k \leftarrow 5\} & \end{pmatrix} \wedge \Big(bag.min = ticket(k)\Big) \wedge pres(ticket, bag, avail)$$

where $pres(bag, avail, ticket)$ stands for the equalities:

$$bag' = bag \quad \wedge \quad avail' = avail \quad \wedge \quad ticket' = ticket$$

Note that the last equality $(ticket' = ticket)$ is an array equality. The parametrized initial condition of SETMUTEX for two thread ids i and j is the formula $\Theta_{\mathsf{param}}(\{i, j\})$:

$$avail = 0 \ \wedge \ bag = \emptyset \ \wedge \begin{pmatrix} ticket(i) = 0 \\ \wedge \\ pc(i) = 1 \end{pmatrix} \wedge \begin{pmatrix} ticket(j) = 0 \\ \wedge \\ pc(j) = 1 \end{pmatrix} \qquad (2.2)$$

┘

2.4.5 Parametrized Formulas

A *parametrized formula* $\varphi(\{k_0, \ldots, k_n\})$ with free variables $\{k_0, \ldots, k_n\}$ of sort tid is simply a formula from $F_{\mathsf{param}}(\{k_0, \ldots, k_n\})$. For clarity, we use $\overline{k}$ for $\{k_0, \ldots, k_n\}$ when the size and index of the set of tid variables is not relevant. Parametrized formulas can only compare thread identifiers using equality and inequality, and no constant thread identifier exists.

We are interested in verifying temporal properties of parametrized programs, so we extend parametrized formulas to *temporal parametrized formulas*, by taking predicates from $F_{\mathsf{param}}(\{k_0 \ldots, k_n\})$ and combining them using temporal operators from LTL ($\bigcirc$, $\mathcal{U}$, $\square$, etc).

Example 2.14
We revisit the SETMUTEX program presented in Example 2.2. For such program, the following formula —which is a 2-index safety formula— expresses mutual exclusion for SETMUTEX:

$$\varphi_{\mathsf{mutex}}(i,j) \quad = \quad \Box\Big(i \neq j \rightarrow \neg(pc(i) = 6 \wedge pc(j) = 6)\Big) \tag{2.3}$$

Similarly, progress of each individual thread is expressed by the following 1-index temporal formula:

$$\varphi_{\mathsf{progress}}(i) \quad = \quad \Box\Big(pc(i) = 3 \rightarrow \Diamond pc(i) = 6\Big)$$

⌟

We will usually refer to a function to be in its conjunctive or disjunctive normal form. The definition we will manage about these two concepts is the standard one.

Definition 2.3 (Disjunctive and Conjunctive Normal Form).
A clause is a literal or its negation. A formula is in disjunctive normal form *if it is expressed as a disjunction consisting of one or more disjuncts, each of which is a conjunction of one or more clauses.*

Similarly, a formula is in conjunctive normal form *if it is expressed as a conjunction consisting of one or more conjuncts, each of which is a disjunction of one or more clauses.* ⌟

We now formally present the concept of vocabulary of a formula (Voc).

Definition 2.4 (Vocabulary).
The vocabulary of a formula is defined as the set of free variables of type tid *appearing in a formula. Formally:*

$$Voc(c) = \begin{cases} \{c\} & \text{if } c \in C^{\mathsf{tid}} \\ \emptyset & \text{otherwise} \end{cases} \qquad Voc(pc(k)) = \{k\}$$

$$Voc(v) = \begin{cases} \{v\} & \text{if } v \in V^{\mathsf{tid}}_{global} \\ \emptyset & \text{otherwise} \end{cases} \qquad Voc(v(k)) = \begin{cases} \{v(k), k\} & \text{if } v(k) \in V^{\mathsf{tid}}_{local} \\ \{k\} & \text{otherwise} \end{cases}$$

$$Voc(\varphi_1 \bowtie \varphi_2) = Voc(\varphi_1) \cup Voc(\varphi_2) \qquad Voc(\rhd\varphi) = Voc(\varphi)$$

where $\bowtie$ represents any binary operators like $\wedge$, $\vee$, $\rightarrow$, $\mathcal{U}$ or $\mathcal{W}$, and $\rhd$ denotes any unary operator such as $\neg$, $\bigcirc$, $\Box$, $\Diamond$, etc. ⌟

Let φ_i be a formula for $i = 1, \ldots, n$. In general, we use $Voc(\varphi_1, \ldots, \varphi_n)$ to denote $\bigcup_{i=1}^{n} Voc(\varphi_i)$.

2.4.6 Parametrized Temporal Verification

In order to define the parametrized temporal verification problem we need to introduce the notion of concretization. Let us fix a program P, and let $\{\mathcal{S}_P[M]\}$ be its instance family and $\mathcal{P}_P$ be the parametrized transition system.

Definition 2.5 (Concretization).
Given a parametrized formula φ and a concrete number of threads M, a concretization of φ is a substitution that maps tid variables in φ into concrete thread identifiers in $[M]$:

$$\alpha : Voc(\varphi) \to [M]$$ ⌟

In this manner, elementary propositions from the parametrized formula φ are in T_{param} but the corresponding elementary propositions of the concrete $\alpha(\varphi)$ are in T_{prog} using the variables of the concrete system $\mathcal{S}[M]$. We use Arr_M^{φ} for the set of concretizations of φ and M.

A concretization α can be lifted inductively to convert Σ_{param} expressions (parametrized expressions) into Σ_{prog} expressions (non-parametrized expressions for $\mathcal{S}_P[M]$). All function symbols F and predicate symbols P in Σ_{param} that are not in the theory of arrays are translated to the same symbols in Σ_{prog}:

$$\begin{array}{rcl} \alpha(F(t_1,\ldots,t_n)) & \mapsto & F(\alpha(t_1),\ldots,\alpha(t_n)) \\ \alpha(P(t_1,\ldots,t_m)) & \mapsto & P(\alpha(t_1),\ldots,\alpha(t_m)) \end{array}$$

For symbols in the theory of arrays, we first translate all literals of sort array in a formula φ to:

(a) either variables of sort array, or

(b) array updates in the right of equalities $w = v\{k \leftarrow e\}$

This translation can be easily achieved by introducing a fresh array variable v for every more complex term t of type array occurring in φ, conjoining $v = t$ to the root of φ and substituting in φ all occurrences of t for v. Then, α can be defined for the remaining array cases:

$$\begin{array}{rcl} \alpha(v(k_i)) & \mapsto & v[\alpha(k_i)] \\ \alpha(w = v\{k \leftarrow e\}) & \mapsto & \Big(w[\alpha(k)] = e \wedge \bigwedge_{a \in M \setminus \alpha(k)} w[a] = v[a]\Big) \\ \alpha(w = v) & \mapsto & \bigwedge_{a \in M} w[a] = v[a] \\ \alpha(v \neq w) & \mapsto & v[k] \neq w[k] \text{ for a fresh } k \end{array}$$

Finally, this map can be extended to formulas in the usual manner, extending to Boolean and temporal connectives. In fact, for fully symmetric systems, we have the following lemma:

Lemma 2.1:
Let ψ be a parametrized formula and α a concretization of ψ for a given number of threads M. Then,

if ψ is a valid formula, then $\alpha(\psi)$ is also a valid formula. ⌟

Proof. The proof proceeds by showing that if $\alpha(\psi)$ has a model then ψ also has a model. The lemma then follows because if ψ is valid, then $\neg\psi$ has no model. Consequently, $\alpha(\neg\psi)$ can have no model either and $\alpha(\psi)$ must be valid too.

Starting from a model $\mathcal{A}$ of $\alpha(\psi)$ we build the model $\mathcal{B}$ of ψ as follows:

- The *domains* of all sorts in $\mathcal{A}$ and $\mathcal{B}$ coincide, except for arrays, in which case indices are $\mathbb{Z}$ and values are the corresponding domain in $\mathcal{A}$. In particular, the domain of the variable pc is that of arrays indexed by $\mathbb{Z}$ with values ranging over locations in *Loc*.

- For *terms*: the only terms of sort tid occurring in ψ are variables. For these, $\mathcal{B}$ assigns the integer within range $[M]$ given by α:

$$k^{\mathcal{B}} = \alpha(k)$$

Array terms in ψ can be either a variable v or a term $v\{k \leftarrow e\}$, but the function symbol $\{\cdot \leftarrow e\}$ is interpreted, so we only need to specify the valuation in $\mathcal{B}$ of array variables. We let $\mathcal{B}$ assign, for indices within $[M]$ the value of the corresponding variable in $\mathcal{A}$ and for values out of the range $[M]$, the array is filled with a fixed value d_σ (an arbitrary value) in the domain of the sort σ of elements of the array. Formally:

$$v^{\mathcal{B}}(n) = \begin{cases} (v[n])^{\mathcal{A}} & \text{if } n \in [M] \\ d_\sigma & \text{if } n \notin [M] \end{cases} \quad (2.4)$$

Note, in particular, that for tid variable k,

$$(v(k))^{\mathcal{B}} \quad = \quad v^{\mathcal{B}}(k^{\mathcal{B}}) \quad = \quad (v[\alpha(k)])^{\mathcal{A}}$$

All other function symbols are interpreted in $\mathcal{B}$ as in $\mathcal{A}$. This is well-defined since all domains (and signatures) coincide. It follows that, with the possible exception of arrays:

$$\text{for all terms } t \qquad t^{\mathcal{B}} = (\alpha(t))^{\mathcal{A}}.$$

- The only *predicate* in the extended theory that is not in the concrete theory is array equality. We first show that for all array variables v and w:

$$(w = v)^{\mathcal{B}} \iff (\alpha(w = v))^{\mathcal{A}} \quad (2.5)$$

Since for all $n \notin [M]$, by (2.4), $w^{\mathcal{B}}(n) = d_\sigma = v^{\mathcal{B}}(n)$, it follows that:

$$\begin{aligned} (w = v)^{\mathcal{B}} &\iff (d_\sigma = d_\sigma) \wedge \bigwedge_{a \in [M]} (w^{\mathcal{B}}(a) = v^{\mathcal{B}}(a)) \\ &\iff \bigwedge_{a \in [M]} (w[a]^{\mathcal{A}} = v[a]^{\mathcal{A}}) \\ &\iff \bigwedge_{a \in [M]} (w[a] = v[a])^{\mathcal{A}} \\ &\iff (\alpha(w = v))^{\mathcal{A}} \end{aligned}$$

Therefore, (2.5) holds.

Second, we show that:

$$(w = v\{k \leftarrow e\})^{\mathcal{B}} \iff (\alpha(w = v\{k \leftarrow e\}))^{\mathcal{A}} \quad (2.6)$$

Given array variables v and w, tid variable k, and term e (of the appropriate sort of elements stored in arrays v and w) by (2.4) all indices not in $[M]$ are mapped to the same value d_σ. Then, $w^{\mathcal{B}}(n) = v^{\mathcal{B}}(n)$ for all $n \notin [M]$. It follows that:

$$\begin{aligned}
(w = v\{k \leftarrow e\})^{\mathcal{B}} &\iff w^{\mathcal{B}}(n) = (v\{k \leftarrow e\})^{\mathcal{B}}(n) \quad \text{for all } n \in [M] \\
&\iff w^{\mathcal{B}}(\alpha(k)) = e^{\mathcal{B}} \wedge \bigwedge_{a \in [M] \setminus \alpha(k)} (w(a)^{\mathcal{B}} = v(a)^{\mathcal{B}}) \\
&\iff w[\alpha(k)]^{\mathcal{A}} = e^{\mathcal{A}} \wedge \bigwedge_{a \in [M] \setminus \alpha(k)} (w[a]^{\mathcal{A}} = v[a]^{\mathcal{A}}) \\
&\iff (\alpha(w = v\{k \leftarrow e\}))^{\mathcal{A}}
\end{aligned}$$

Therefore, (2.6) holds as well.

Finally, for all common predicates P, that is, for all predicates except those in the theory of arrays we let:

$$P^{\mathcal{B}}(a_1, \ldots, a_n) \iff P^{\mathcal{A}}(a_1, \ldots, a_n)$$

Hence, by (2.4), (2.5) and (2.6) it follows that for all predicates, including those in the theory of arrays:

$$(P(t_1, \ldots, t_n))^{\mathcal{B}} \iff \alpha(P(t_1, \ldots, t_n))^{\mathcal{A}}$$

Since all atomic predicates of ψ have the same truth value in $\mathcal{B}$ as the corresponding predicates of $\alpha(\psi)$ in $\mathcal{A}$, it follows that $\mathcal{B}$ is a model of ψ because $\mathcal{A}$ is a model of $\alpha(\psi)$.

This finishes the proof. $\square$

In general, we use parenthesis for parameters of parametrized formulas and brackets for parameters of formulas on concrete systems. That is, if $\varphi(t)$ is an 1-index parametrized formula, then $\varphi(i)$ is formula $\varphi(t)$ with all occurrences of t replaced by i. Note that $\varphi(i)$ is still a parametrized formula. Instead, $\varphi[T]$ denotes formula φ where all occurrences of t has been instantiated for the concrete thread identifier T. The same notation applies for transitions. For instance, $\tau_\ell^{(t)}$ is the transition relation associate to program location ℓ for an arbitrary thread t, while $\tau_\ell^{[T]}$ is the same transition relation but instantiated for the concrete thread identifier T.

Essentially, a concretization computes the predicate $\alpha(\varphi)$ for system $\mathcal{S}_P[M]$ that results from φ by instantiating its variables $Voc(\varphi)$ according to the map α.

Example 2.15

Consider the formula $\Theta_{\mathsf{param}}(\{i, j\})$ shown as Equation (2.2) in Example 2.13 above. The concretization of $\Theta_{\mathsf{param}}(\{i, j\})$ by the map $\alpha : \{i \leftarrow 0, j \leftarrow 1\}$ is the concrete initial condition expressed by Equation (2.1) in Example 2.12.

Similarly, if we consider the formula φ_{mutex} from Example 2.14, its concretization according to the map $\alpha_1 : \{i \rightarrow 0, j \rightarrow 1\}$ is:

$$\alpha_1(\varphi_{\mathsf{mutex}}) \quad = \quad \square\neg(pc[0] = 6 \wedge pc[1] = 6)$$

However, with the concretization $\alpha_2 : \{i \rightarrow 0, j \rightarrow 0\}$, the resulting formula $\alpha_2(\varphi_{\mathsf{mutex}})$ is:

$$\alpha_2(\varphi_{\mathsf{mutex}}) \quad = \quad \square\top$$

⌟

We are now ready to define formally the *parametrized temporal verification problem*.

Definition 2.6.
Given a parametrized system $\mathcal{S}$ *and a parametrized temporal formula* $\varphi(\overline{k})$ *we say that* $\mathcal{S} \vDash \varphi(\overline{k})$ *whenever for all concrete instances* $\mathcal{S}[M]$ *and concretizations* α, $\mathcal{S}[M] \vDash \alpha(\varphi(\overline{k}))$. ⌟

Part I

Parametrized Verification Techniques

3

Parametrized Invariance

> *Absence of evidence*
> *is not evidence of absence.*
>
> Carl Sagan

In this chapter we present *Parametrized Invariance*, a general method to verify safety temporal properties in concurrent parametrized programs. The parametrized concurrent programs we are interested in are executed by an unbounded number of threads and they can manipulate complex data, including unbounded local and shared state. We propose in this chapter a method that solves the *uniform verification* problem for safety properties. That is, given a parametrized system $\mathcal{P} : P(1) \parallel P(2) \parallel \ldots \parallel P(N)$ and a safety property φ, establish whether

$$\mathcal{S}_P[M] \vDash \varphi \qquad \text{for all instances } M \geq 1$$

In particular, we solve this problem for systems processes that manipulate arbitrary *infinite data*.

Parametrized Invariance is a generalization of the inductive invariance proof rules for temporal deductive verification [144], in which each verification condition corresponds to a small-step (a single transition) in the execution of a system. The applicability of these proof rules, without adding quantifiers, is restricted to non-parametrized systems. In fact, non-parametrized systems can be described by a finite number of transitions, so one can generate one verification condition per transition. However, in parametrized systems, the number of transitions depends on the concrete number of processes in each particular instantiation, which is unbounded: the larger the instantiation, the larger the set of transitions to consider.

To tackle this problem, our parametrized invariance rules automatically discharge a *finite* collection of verification conditions. The validity of these verification conditions implies the correctness of all concrete system instantiations. In order to generate a finite set of verification

conditions, the proof rules we present in this chapter capture the effect of single steps of the following:

- all threads explicitly referred to in the property, and
- an arbitrary thread not involved in the property definition.

All generated verification conditions using the parametrized invariance proof rules are quantifier-free as long as the transition relations and the specifications are quantifier-free. The verification conditions generated using parametrized invariance can then be automatically checked as valid using appropriate decision procedures for each manipulated data type.

The rest of this chapter is structured as follows. Section 3.1 presents the traditional deductive invariance rules for non-parametrized systems and shows why they are unsuitable for parametrized systems. Section 3.2 introduces the novel proof rules for *parametrized invariance*, the method we propose for verifying safety properties on parametrized systems, and proves these new proof rules sound. Finally, Section 3.3 summarizes this chapter.

3.1 The Need of Parametrized Safety Proof Rules

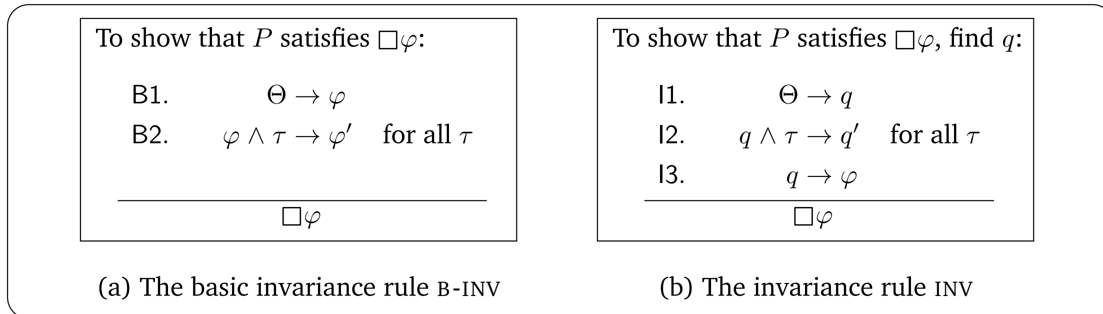
We begin by showing the need of specialized proof rules for the verification of safety properties of parametrized systems. To do so, we first formulate the uniform verification problem in terms of concretizations.

Definition 3.1 (Uniform Verification Problem).
Given a program P and a parametrized formula $\varphi(\overline{k})$, we say that P satisfies the universal safety property:

$$\left(\forall \overline{k} \,.\, \Box\varphi(\overline{k})\right)$$

whenever for every M and substitution $\alpha : \overline{k} \rightarrow [M]$, the concrete closed system $\mathcal{S}_P[M]$ satisfies $\mathcal{S}_P[M] \vDash \Box\alpha(\varphi(\overline{k}))$. In this case we write $P \vDash \forall\overline{k}.\Box\varphi(\overline{k})$, or simply $P \vDash \Box\varphi$ and say that φ is a parametrized invariant *of P.* ⌟

We say that a system $\mathcal{S}$ satisfies a safety property p, which we write $\mathcal{S} \vDash \Box p$, whenever all runs of $\mathcal{S}$ satisfy p at all states. Fig. 3.1 presents the classical invariance rules [144] for



(a) The basic invariance rule B-INV (b) The invariance rule INV

Figure 3.1: Rules B-INV and INV for non-parametrized systems.

non-parametrized systems. The formula φ' in the consequent of premise B2 refers to the formula obtained from φ by replacing every variable v in φ by v'. The formula q' in the consequent of premise I2 is obtained from q similarly. The basic rule B-INV establishes that if the candidate invariant φ holds initially and is preserved by every transition, then φ is indeed an invariant. In this case we call φ an *inductive invariant*.

A naïve approach to prove parametrized invariants is to try to enumerate all concrete instances and repeatedly use rule B-INV for each resulting instance to show that each possible concretization is an invariant. However, as we show below, this approach requires proving an unbounded number of verification conditions because one (potentially different) verification condition is discharged per transition and per thread that is present in each instantiated closed system.

Example 3.1
Fig. 3.2 presents procedure TWO. This procedure operates over a local variable x which is initialized to 1 and then it is multiplied by 2 (line 1).

A simple invariant property we would like to verify is alwaysPositive:

$$\text{alwaysPositive}: \ \square(x > 0)$$

In this case, we can use the basic proof rule B-INV presented in Fig. 3.1(a), since, for premise B1 we have that:

$$pc = 1 \quad \wedge \quad x = 1 \quad \rightarrow x > 0$$

And following premise B2, every transition of TWO preserves the invariant candidate:

$$x > 0 \quad \wedge \quad pc = 1 \quad \wedge \quad pc' = 2 \quad \wedge \quad x' = x \times 2 \quad \rightarrow \quad x' > 0$$

Hence, alwaysPositive is an inductive invariant. ⌟

A stronger rule is INV, shown in Fig. 3.1(b), which uses an intermediate strengthening invariant q. If q implies φ and q is an invariant, then φ is also an invariant. An alternative characterization of rule INV requires finding q and proving that $(q \wedge \varphi)$ is an inductive invariant using rule B-INV.

Example 3.2
Consider again procedure TWO defined in Example 3.1. We would like to prove as invariant the formula finallyTwo, which says that the procedure always returns value 2. That is:

$$\text{finallyTwo}: \ \square(pc = 2 \rightarrow x = 2)$$

```
   procedure Two()
     Int x = 1
   begin
1: x := x × 2
2: return (x)
   end procedure
```

Figure 3.2: Declaration of procedure TWO.

We cannot prove finallyTwo invariant using B-INV because it is not inductive. In particular, the following verification condition is not valid:

$$(pc = 2 \rightarrow x = 2) \wedge \ pc = 1 \wedge \ pc' = 2 \wedge \ x' = x \times 2 \quad \rightarrow \quad (pc' = 2 \rightarrow x' = 2)$$

So, in this case we require to use proof rule INV and find a stronger inductive invariant in order to prove finallyTwo invariant.

Let strgFinallyTwo be th following candidate invariant:

$$\mathsf{strgFinallyTwo} : \ \Box(pc = 1 \rightarrow x = 1 \wedge pc = 2 \rightarrow x = 2)$$

We will use strgFinallyTwo as q in the INV rule. Clearly, strgFinallyTwo is inductive and can be verified using premise I1:

$$pc = 1 \quad \wedge \quad x = 1 \quad \rightarrow \quad (pc = 1 \rightarrow x = 1 \wedge pc = 2 \rightarrow x = 2)$$

and premise I2:

$$\begin{pmatrix} pc = 1 \rightarrow x = 1 \\ \wedge \\ pc = 2 \rightarrow x = 2 \end{pmatrix} \quad \wedge \quad \begin{pmatrix} pc = 1 \\ \wedge \\ pc' = 2 \end{pmatrix} \quad \wedge \quad x' = x \times 2 \quad \rightarrow \quad \begin{pmatrix} pc' = 1 \rightarrow x' = 1 \\ \wedge \\ pc' = 2 \rightarrow x' = 2 \end{pmatrix}$$

Finally, premise I3 holds:

$$(pc = 1 \rightarrow x = 1 \wedge pc = 2 \rightarrow x = 2) \quad \rightarrow \quad (pc = 2 \rightarrow x = 2)$$

which proves that finallyTwo is an invariant, as desired. ⌟

For non-parametrized systems, premises B1 and I1 —called *initiation*— discharge one verification condition, and premises B2 and I2 —called *consecution*— discharge a collection of verification conditions whose size is linear in the number of transitions. To use these invariance rules directly for parametrized systems, one either needs to use quantification (as in [169]) or apply the rules once for each concrete system instantiation, which requires to discharge and prove an unbounded number of verification conditions.

Example 3.3
We now revisit program SETMUTEX presented in Example 2.2. Consider the following specification availNotNeg, which ensures that $avail$ is never negative:

$$\mathsf{availNotNeg} : \ \Box(avail \geq 0)$$

Specification availNotNeg is in fact an inductive invariant. But proving it invariant using the traditional B-INV rule would require to verify that:

$$avail \geq 0 \wedge \tau_\ell[t] \rightarrow \ avail' \geq 0$$

for all transition relations $\tau_\ell[t]$ presented in Example 2.12 with $\ell \in \{1..7\}$ and each thread t in the system. Hence, for a system with M threads we would require to verify $7 \times M + 1$ verification conditions. ⌟

The novelty of the *parametrized invariance* rules we present in this chapter is that they allow to tackle parametrized systems while discharging only a finite number of verification conditions.

3.2 Parametrized Proof Rules

We now present *parametrized invariance*, a set of specialized proof rules for parametrized systems, which allow to prove parametrized invariants discharging only a finite number of verification conditions. As happens with non-parametrized invariance rules, some parametrized proof rules can be applied to inductive invariants, while more sophisticated parametrized proof rules are required for non inductive invariants. We now proceed to describe each of the new parametrized proof rules.

3.2.1 The Basic Parametrized Invariance Rule: BP-INV

The simplest proof rule is BP-INV, called the *basic parametrized invariance rule*, which is shown in Fig. 3.3. Basically, in BP-INV:

Premise P1: guarantees that the initial condition holds for all instantiations. This premise discharges only one verification condition.

Premise P2: guarantees that φ is preserved under transitions taken by all threads referred in the formula and considering all possible transitions of the system. This premise discharges one verification condition per transition in the description of the system (each statement in the program) and per index variable in the formula φ.

Premise P3: guarantees that φ is preserved for all transitions taken by *any other thread*. This is achieved by taking a *fresh* thread identifier in P3. A *fresh* variable of type thread identifier refers to a variable not appearing in φ. Premise P3 generates one extra verification condition per transition in the system.

All the generated verification conditions using BP-INV are quantifier-free provided that φ is quantifier-free. Premises P2 and P3 must be checked for all system transitions. Moreover, premise

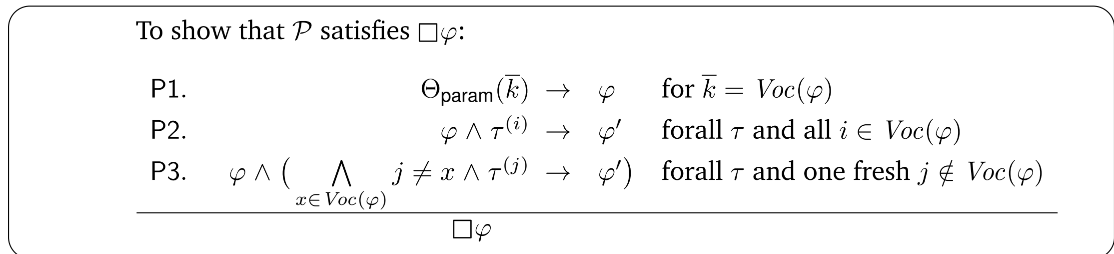


Figure 3.3: The basic parametrized invariance rule BP-INV.

P2 requires each transition to be checked for every thread identifier appearing in the formula φ. Corollary 3.1 in page 63 justifies the soundness of rule BP-INV.

3.2.2 The Parametrized Invariance Rule: P-INV

There are cases in which premise P3 from proof rule B-INV presented in Section 3.2.1 cannot be proven, even if φ is initial and preserved by all transitions of all threads in all system instantiations.

Example 3.4
Consider the program POSITIVE presented in Fig. 3.4. The program simply adds a positive number c to a global positive variable x and returns the result.

Consider now the 1-index property:

$$\varphi_{\text{POS}}(i) = (x > 0 \wedge c(i) > 0)$$

Property φ_{POS} is trivially a parametrized invariant. However, premise P3 from BP-INV is not valid for this property when a fresh thread identifier j takes the transition at line 1, as we require to prove the validity of the following formula:

$$\Big((x > 0 \wedge c(i) > 0) \quad \wedge \quad (j \neq i) \quad \wedge \quad x' = x + c(j) \quad \wedge \quad c' = c\Big) \quad \rightarrow \quad x' > 0 \wedge c'(i) > 0$$

The formula above is not valid as we can construct, for example, the following counter-model:

$$x = 1 \qquad i = 0 \qquad j = 1 \qquad c(0) = 1 \qquad c(1) = -1 \qquad x' = 0 \qquad c' = c$$

Essentially, the problem is that formula φ_{POS} does not imply that $c(j) > 0$ before the transition $\tau_1^{(j)}$ is taken (although it states that $c(i) > 0$). Because of that, the counter-example can assign $c(j) = -1$. A transition for which the corresponding verification condition is not valid is known as an *offending transition* (see [144]), or more modernly as a *counter-example to induction* [36].⌟

The problem exposed in Example 3.4 is that in the antecedent of premise P3, φ does not refer to the fresh arbitrary thread introduced. In other words, BP-INV tries to prove a property for the threads referred to in the formula, without assuming anything about any other thread. It is sound, however, to assume that in the pre-state of the verification condition the property one intends to prove holds *for all processes*, and not only for the processes explicitly mentioned in the formula.

```
   global
      Int x > 0
   procedure POSITIVE()
      Int c > 0
   begin
1:    x = x + c
2: return (x)
   end procedure
```

Figure 3.4: Declaration of program POSITIVE.

Intuitively speaking, the justification of this assumption is based on:

$$\forall k \,.\, \Box\varphi(k) \qquad \text{being equivalent to} \qquad \Box(\forall k \,.\, \varphi(k))$$

However, we want to avoid quantification in all verification conditions. Instead, we propose to instantiate the formula φ in the antecedent of premises, not only to threads in the formula, but also to other threads. In particular, for all threads appearing in the transition relation. The notion of *support* allows to formally capture this intuition. We use the conventional notion of *substitution* in first-order-logic (as a map from variables to terms), and restrict our attention to maps from a set of tid variables X into set of tid variables Y. Substitutions can be extended to maps from terms to terms and (formulas to formulas) homomorphically in the usual way, preserving all symbols except the replaced variables. A partial substitution is then a partial map.

Definition 3.2 (Support).
Let ψ, A and B be parametrized formulas, and let S be the set of partial substitutions from $Voc(\psi)$ into $Voc(A \to B)$. We say that ψ supports $(A \to B)$, whenever

$$\left(\Big(\bigwedge_{\sigma \in S} \sigma(\psi) \Big) \wedge A \right) \to B \qquad \textit{is valid}$$

We use $\psi \rhd (A \to B)$ as a short notation for $\left((\bigwedge_{\sigma \in S} \sigma(\psi)) \wedge A \right) \to B$. ⌟

Note that if $S' \subseteq S$ is a subset of the substitutions, and

$$\left(\Big(\bigwedge_{\sigma \in S'} \sigma(\psi) \Big) \wedge A \right) \to B \qquad \text{is valid}$$

then

$$\left(\Big(\bigwedge_{\sigma \in S} \sigma(\psi) \Big) \wedge A \right) \to B \qquad \text{is also valid}$$

Essentially, if one is successful in proving the validity of a formula obtained by removing some of the conjuncts from the antecedent, the validity of the full formula is guaranteed. Hence, in practice, it is enough to consider only some of the partial substitutions to show that a support formula is valid.

Example 3.5
Consider again program POSITIVE presented in Example 3.4 above, and let A and B be the formulas:

$$A \,:\, (i \neq j) \wedge \tau_1^{(j)} \qquad\qquad B \,:\, \varphi_{\text{POS}}(i)'$$

The formula $\varphi_{\text{POS}} \rhd (A \to B)$ is:

$$\Big(\varphi_{\text{POS}}(i) \;\wedge\; \varphi_{\text{POS}}(j) \;\wedge\; \varphi_{\text{POS}}(k) \Big) \;\wedge (i \neq j) \;\wedge\; \tau_1^{(j)} \;\to\; \varphi_{\text{POS}}(i)'$$

This formula is valid. Note that the subformula $\varphi_{\text{POS}}(k)$ in the antecedent is obtained by applying the empty substitution to φ_{POS}. ⌟

The main motivation for introducing the notion of support is to instantiate a formula ψ which is an assumed fact in the pre-state. This way, we strengthen the antecedent of an implication (the verification condition) without extending the vocabulary (the free tid variables) in the resulting strengthened implication. If we consider again premise P3 from rule BP-INV, we can now construct a new premise which strengthens premise P3, so that the target invariant candidate φ can be assumed in the pre-state *for every thread*, in particular for the fresh thread that takes the transition. We name this new premise S3:

$$\text{S3.}\quad \varphi \rhd \Big(\bigwedge_{x \in Voc(\varphi)} j \neq x \wedge \tau^{(j)} \;\rightarrow\; \varphi' \Big) \quad \text{forall } \tau \text{ and one fresh } j \notin Voc(\varphi)$$

Example 3.6

Let $\varphi(i)$ be a candidate invariant parametrized by one thread variable i, that is, a 1-index invariant candidate. For a transition taken by thread j, premise S3 described above is:

$$\Big(\varphi \rhd (j \neq i \wedge \tau^{(j)} \rightarrow \varphi'(i))\Big)$$

or equivalently:

$$\Big(\varphi(j) \;\wedge\; \varphi(i) \;\wedge\; j \neq i \;\wedge\; \tau^{(j)}\Big) \;\rightarrow\; \varphi'(i)$$

Note how $\varphi(j)$ in the antecedent is the result of instantiating φ for the fresh thread j introduced by the premise. ⌟

Using the notion of support introduced in Definition 3.2, we can now rewrite the basic parametrized invariance rule BP-INV into the parametrized invariance rule P-INV, shown in Fig. 3.5.

Unfortunately, rule P-INV can still fail to prove invariants if they are not inductive, as the following example shows.

To show that $\mathcal{P}$ satisfies $\Box\varphi$:

$$\begin{array}{lrcll}
\text{S1.} & \Theta_{\text{param}} & \rhd & \varphi & \\
\text{S2.}\ \varphi \rhd & \tau^{(i)} & \rightarrow & \varphi' & \text{forall } \tau \text{ and all } i \in Voc(\varphi) \\
\text{S3.}\ \varphi \rhd & \big(\bigwedge_{x \in Voc(\varphi)} j \neq x \wedge \tau^{(j)} & \rightarrow & \varphi' \big) & \text{forall } \tau \text{ and one fresh } j \notin Voc(\varphi) \\
\hline
 & & \Box\varphi & &
\end{array}$$

Figure 3.5: The parametrized invariance rule P-INV.

Example 3.7
Consider again program SETMUTEX presented in Example 2.2 in page 21. Let notsame be the following 2-index property parametrized by thread identifiers i and j:

$$\mathsf{notsame}(i,j) \;:\; \Big(i \neq j \wedge \mathsf{active}(i) \wedge \mathsf{active}(j)\Big) \;\rightarrow\; \mathit{ticket}(i) \neq \mathit{ticket}(j)$$

where $\mathsf{active}(t)$ is a short notation for $(pc(t) = 4 \vee pc(t) = 5 \vee pc(t) = 6)$.

Property $\mathsf{notsame}(i,j)$ is a parametrized invariant of SETMUTEX, but it cannot be proven using P-INV. Premise S2 fails when taking $\tau_3^{(j)}$ (this transition is an offending transition for proving the property invariant) as witnessed by a model from a pre-state in which:

$$pc(i) = 4 \qquad pc(j) = 3 \qquad \mathit{ticket}(i) = 1 \qquad \mathit{avail} = 1$$

A true fact of the program that eliminates this spurious counter-example is that when a thread i is in $\mathsf{active}(i)$, then $\mathit{ticket}(i) < \mathit{avail}$. But here, neither the goal invariant $\mathsf{notsame}(i,j)$ nor the transition relation for $\tau_3^{(j)}$ directly imply this fact. This is a limitation of P-INV. Later, in Section 3.2.3, we will present an additional proof rule named SP-INV which overcomes this limitation. ⌟

We now present the theorem which proves the soundness of parametrized proof rule P-INV.

Theorem 3.1 (Soundness of P-INV):
Let $\mathcal{P}$ be a parametrized system and $\Box\varphi$ a parametrized safety property. If premises S1, S2 *and* S3 *hold, then $\mathcal{P} \vDash \Box\varphi$.*

Proof. Given φ, let M be an arbitrary bound. We will show that the premises B1 and B2 of the basic non-parametrized invariance rule B-INV hold for the concrete non-parametrized system $\mathcal{S}_P[M]$ and the concrete formula Ψ:

$$\Psi \;\stackrel{\text{def}}{=}\; \bigwedge_{\alpha \in \mathit{Arr}_M^{\varphi}} \alpha(\varphi)$$

Since for an arbitrary concretization α, the formula $\alpha(\varphi)$ is one of the conjuncts of Ψ, it follows that if Ψ is an invariant of the concrete non-parametrized system $\mathcal{S}_P[M]$ then $\alpha(\varphi)$ is also an invariant of $\mathcal{S}_P[M]$. An alternative model-theoretic proof would consist on showing that there is no violating trace of $\alpha(\varphi)$ in $\mathcal{S}_P[M]$. We present here the proof-theoretic argument, that shows additionally that Ψ is inductive, and not only that Ψ is invariant as the model-theoretic proof would show. We use *Img* α for denoting those concrete indices in $[M]$ that are in the image of α, that is, those concretes indices that α maps from tid variables in φ. We now need to show that both premises of B-INV are valid.

Premise B1: Since premise S1 is valid, then $\Theta_{\mathsf{param}}(\overline{k}) \rhd \varphi$ is valid, or equivalently:

$$\Theta_{\mathsf{param}}(\overline{k}) \;\rightarrow\; \varphi \qquad\qquad \text{where } \overline{k} = \mathit{Voc}(\varphi)$$

Consequently, by Lemma 2.1, $\alpha(\Theta_{\mathsf{param}}(\overline{k}) \rightarrow \varphi)$ is valid for an arbitrary α, and then, $\alpha(\Theta_{\mathsf{param}}(\overline{k})) \rightarrow \alpha(\varphi))$ is valid for an arbitrary α. Then:

$$\Theta_g \wedge \bigwedge_{n\in[M]} \Theta_l[n] \quad \rightarrow \quad \bigwedge_{\alpha\in Arr^{\varphi}_M} \alpha(\varphi) \qquad \text{is valid}$$

and, finally:

$$\Theta_g \wedge \bigwedge_{n\in[M]} \Theta_l[n] \quad \rightarrow \quad \Psi \qquad \text{is valid}$$

Premise B2**:** We need to show that for all $n \in [M]$ and all transitions $\tau[n]$:

$$\Psi \wedge \tau[n] \rightarrow \Psi' \qquad \text{is valid} \tag{3.1}$$

Let α be an arbitrary concretization in Arr^{φ}_M. We will show that:

$$\Psi \wedge \tau[n] \rightarrow \alpha(\varphi) \qquad \text{is valid}$$

which implies (3.1) because α is arbitrary. We now consider two cases depending on whether the concrete n is in the image of α or not:

1. Case $n \in Img\,\alpha$: That is, there is an $i \in Voc(\varphi)$ for which $\alpha(i) = n$. In this case, since premise S2 for $\tau^{(i)}$ is valid, due to Lemma 2.1, we have:

$$\alpha(\varphi \rhd \tau^{(i)} \;\rightarrow\; \varphi') \qquad \text{is valid}$$

   or, equivalently, for the set S of partial substitutions:

$$\alpha(\bigwedge_{\sigma\in S} \sigma(\varphi)) \wedge \alpha(\tau^{(i)}) \;\rightarrow\; \alpha(\varphi') \qquad \text{is valid}$$

   Then, since the application of concretization α after a substitution is a concretization:

$$\bigwedge_{\alpha_2\, \in\, Arr^{\varphi}_M} \alpha_2(\varphi) \wedge \tau[n] \;\rightarrow\; \alpha(\varphi') \qquad \text{is valid}$$

   which implies that:

$$\Big(\Psi \wedge \tau[n]\Big) \;\rightarrow\; \alpha(\varphi') \qquad \text{is valid}$$

2. Case $n \notin Img\,\alpha$: That is, there is not an $i \in Voc(\varphi)$ for which $\alpha(i) = n$: Let j be a fresh tid identifier, and let α_3 be the following concretization of $Voc(\varphi) \cup \{j\}$:

$$\alpha_3(k) \;=\; \begin{cases} n & \text{if } k = j \\ \alpha(k) & \text{if } k \neq j \end{cases}$$

Now, since premise S3 is valid, by Lemma 2.1 for α_3, we have:

$$\alpha_3\Big(\varphi \rhd \bigwedge_{x\in Voc(\varphi)} j \neq x \wedge \tau^{(j)} \ \rightarrow\ \varphi'\Big) \qquad \text{is valid}$$

Then, for the set of substitutions S:

$$\Big(\alpha_3(\bigwedge_{\sigma\in S} \sigma(\varphi)) \wedge \alpha_3\Big(\bigwedge_{x\in Voc(\varphi)} j \neq x\Big) \wedge \alpha_3(\tau^{(j)})\Big) \ \rightarrow\ \alpha_3(\varphi') \qquad \text{is valid}$$

and, since substitutions followed by concretizations are concretizations from Arr_M^{φ}, and $\alpha_3\Big(\bigwedge_{x\in Voc(\varphi)} j \neq x\Big)$ simplifies to $true$, $\alpha_3(\tau^{(j)})$ simplifies to $\tau[n]$, and $\alpha_3(\varphi)$ simplifies to $\alpha(\varphi)$:

$$\Big(\bigwedge_{\alpha_4\ \in Arr_M^{\varphi}} \alpha_4(\varphi) \wedge \tau[n]\Big) \ \rightarrow\ \alpha(\varphi') \qquad \text{is valid}$$

and then

$$\Big(\Psi \wedge \tau[n]\Big) \ \rightarrow\ \alpha(\varphi') \qquad \text{is valid}$$

Hence, premise B2 is valid for $\mathcal{S}_P[M]$ and Ψ. Since both premise B1 and premise B2 are valid, then Ψ is an inductive invariant of $\mathcal{S}_P[M]$, and $\alpha(\varphi)$ is an invariant of $\mathcal{S}_P[M]$ for an arbitrary α.□

The following corollary establishes the soundness of rule BP-INV, and follows immediately from Theorem 3.1 by observing that if $(A \wedge B) \rightarrow C$ is valid then $A \rhd B \rightarrow C$ is also valid.

Corollary 3.1 (Soundness of BP-INV):
Let $\mathcal{P}$ be a parametrized system and $\Box\varphi$ a parametrized safety property. If premises P1, P2 *and* P3 *hold, then $\mathcal{P} \vDash \Box\varphi$.*

3.2.3 The General Strengthening Parametrized Invariance Rule: SP-INV

As happens with closed systems, there are two reasons that explain the failure to prove using the invariance proof rules that a candidate invariant is indeed an invariant:

1. the candidate is actually not an invariant;
2. the candidate is invariant but not inductive, so one needs to use strengthening invariants, or to prove the candidate is inductive relative to other invariants.

However, in parametrized systems it is not necessary the case that by simply conjoining the candidate and its strengthening one obtains a BP-INV inductive invariant, because one may need to instantiate the candidate formulas for all thread identifiers in their shared vocabulary.

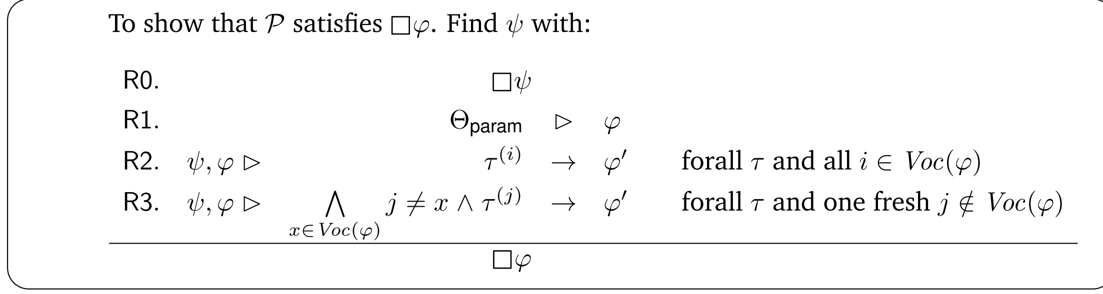


Figure 3.6: The general strengthening parametrized invariance rule SP-INV for proving relative inductive parametrized invariants.

One solution is to prove the invariants incrementally, and use support to instantiate to freshly introduced thread identifiers. This idea is captured by the general strengthening parameterized invariance rule SP-INV shown in Fig. 3.6.

The following theorem shows the soundness of the general strengthening parameterized invariance rule SP-INV.

Theorem 3.2:
Let $\mathcal{P}$ be a parametrized system and $\Box\varphi$ a parametrized safety property. If premises R0*,* R1*,* R2 *and* R3 *from* SP-INV *hold, then $\mathcal{P} \vDash \Box\varphi$.*

Proof. Assume that $Voc(\varphi) \cap Voc(\psi) = \emptyset$ which can be easily achieved by renaming tid variables. The proof is very similar to the proof of Theorem 3.1 showing that:

$$\Psi \stackrel{\text{def}}{=} \bigwedge_{\alpha_1 \in Arr^{\varphi}_{M}} \alpha_1(\varphi) \wedge \bigwedge_{\alpha_2 \in Arr^{\psi}_{M}} \alpha_2(\psi)$$

satisfies the premises B1 and B2 of rule B-INV for $\mathcal{S}_P[M]$ for an arbitrary M. In this case, one can use, by premise R0, that $\alpha_2(\psi)$ holds for every concretization α_2 of ψ. □

Example 3.8
When we presented Example 3.7, we saw that the parametrized proof rule P-INV was not enough to prove that notsame is invariant. In fact, we concluded that while it was not possible to prove notsame invariant using P-INV, it would be possible to prove notsame invariant if we consider the fact that any thread running SETMUTEX has a ticket lower than $avail$ when located at line 4, 5 or 6. Let activelow be the 1-index specification:

$$\text{activelow}(i) \;:\; \Big(\text{active}(i) \rightarrow ticket(i) < avail\Big)$$

We can now apply the parametrized proof rule SP-INV considering notsame and activelow as φ and ψ respectively. Premise R0 holds, as activelow is inductive and can be proven using P-INV;

premise R1 is trivially satisfied. Now, when we consider premise R2 we have, for thread i:

$$\begin{pmatrix}\mathsf{activelow}(i)\\ \wedge\\ \mathsf{activelow}(j)\end{pmatrix} \wedge \mathsf{notsame}(i,j) \wedge \begin{pmatrix}pc(i)=3\\ \wedge\\ pc'(i)=4\end{pmatrix} \wedge \begin{pmatrix}ticket(i)' = avail & \wedge\\ avail' = avail + 1 & \wedge\\ bag' = bag \cup \{avail\} & \end{pmatrix} \to \mathsf{notsame}(i,j)'$$

For thread j and for premise R3 the reasoning is analog. Note that in the verification condition above we consider only a subset of the substitutions from $Voc(\psi) \cup Voc(\varphi) = \{i,j\}$ into $Voc(\tau(i)) \cup Voc(\mathsf{notsame}(i,j)) = \{i,j\}$. ⌟

3.2.4 The Parametrized Graph Proof Rule: G-INV

Finally, we introduce a specialized proof rule for parametrized systems, called the graph proof rule. The main motivation is that carrying out incremental invariance proofs using SP-INV requires in premise R0 to start from an already proved invariant, and it is often the case that invariants mutually depend on each other.

A naïve solution attempt would be to write down all necessary candidates in a single large formula and prove this large formula invariant using P-INV. While theoretically correct, when dealing with parametrized systems this approach quickly leads to formulas with many duplications due to thread renaming which in turn jeopardizes the scalability of the decision procedures for sophisticated data by requiring to prove large formulas, which requires to search for large models. A more efficient approach consists of building the proof modularly, splitting invariants into meaningful sub-formulas to be used only when required. This sort of proof modularity is captured by the parametrized graph proof rule G-INV shown in Fig. 3.7. This rule handles cases in which invariant candidates mutually dependent on each other.

A proof graph is a finite directed graph $(Invs, Supp)$ where nodes in $Invs$ are labeled with candidate invariant formulas. An edge in $Supp$ between two nodes indicates that in order to prove the formula pointed by the edge it is useful to use the formula at the origin of the edge as support. As a particular case, a formula with no incident edges is inductive and can be shown

To show that $\mathcal{P}$ satisfies $\square\varphi$ find a proof graph $(Invs, Supp)$ with $\varphi \in Invs$ such that:

G1.		Θ_{param}	$\rhd$	ψ	forall $\psi \in Invs$
G2.	$\Phi, \psi \rhd$	$\tau^{(k)}$	$\to$	ψ'	forall $\psi \in Invs$, forall τ, and all $k \in Voc(\psi)$, and $\Phi = \{\psi_i \mid (\psi_i, \psi) \in Supp\}$
G3.	$\Phi, \psi \rhd$	$\bigwedge_{x \in v} k \neq x \wedge \tau^{(k)}$	$\to$	ψ'	forall $\psi \in Invs$, forall τ, one fresh $k \notin v = Voc(\psi)$, and $\Phi = \{\psi_i \mid (\psi_i, \psi) \in Supp\}$

$$\square\varphi$$

Figure 3.7: The graph parametrized invariance rule G-INV.

directly using P-INV. Note that a proof graph can be, and in practice it usually is, a cyclic graph. A proof graph encodes the proof that *all* the formulas labeling nodes are invariants of the system. The edges encode the information of which sub-formulas, described by the set of predecessor nodes, are needed to prove a particular node.

We now present the theorem which shows that the parametrized graph proof rule is sound.

Theorem 3.3 (Soundness of Proof Graphs):
Let $\mathcal{P}$ be a parametrized system and $(Invs, Supp)$ a proof graph. If premises G1*,* G2*, and* G3 *from* G-INV *hold, then $\mathcal{P} \vDash \Box\psi$ for all $\psi \in Invs$.* ⌟

Proof. Again, we present a proof theoretic argument to show that, for an arbitrary M, the following is a concrete non-parametrized inductive invariant of $\mathcal{S}_P[M]$:

$$\Psi \stackrel{\text{def}}{=} \bigwedge_{\psi \in Invs} \bigwedge_{\alpha \in Arr_M^{\psi}} \alpha(\psi)$$

The argument to show that premise B1 follows from premise G1 is identical to the argument that premise B1 follows from premise S1, in the proof of Theorem 3.1 above.

For premise B2, we consider an arbitrary ψ in $Invs$ and an arbitrary concretization α from Arr_M^{ψ}. We now need to show that:

$$\Psi \wedge \tau[n] \ \rightarrow \ \alpha(\psi) \qquad \text{is valid}$$

Again, to do so, we consider two cases depending on whether n is in the image of α or not.

1. Case $n \in Img\,\alpha$: Let k in $Voc(\psi)$ be such that $\alpha(k) = n$. In this case, by premise G2 with $\Phi = \{\psi_i \mid (\psi_i, \psi) \in E\}$:

   $$\Phi, \psi \rhd \tau^{(k)} \ \rightarrow \ \psi' \qquad \text{is valid}$$

   and hence

   $$\alpha\Big(\Phi, \psi \rhd \tau^{(k)} \ \rightarrow \ \psi'\Big) \qquad \text{is valid}$$

   Now, by considering the definition of Φ, considering that $\alpha(\tau^{(k)}) = \tau[n]$, and adding conjuncts to the antecedent (which keeps a valid implication valid)

   $$\Psi \wedge \tau[n] \ \rightarrow \ \alpha(\psi') \qquad \text{is valid}$$

2. Case $n \notin Img\,\alpha$: Then, let α_2 be α extended by mapping a fresh tid j with $\alpha_2(j) = n$. Then, by premise G3 of rule G-INV:

   $$\Phi, \psi \rhd \bigwedge_{x \in Voc(\psi)} j \neq x \wedge \tau^{(j)} \ \rightarrow \ \psi' \qquad \text{is valid}$$

or, for α_2 by Lemma 2.1,

$$\alpha_2\Bigg(\Phi, \psi \rhd \bigwedge_{x \in Voc(\psi)} j \neq x \wedge \tau^{(j)} \;\rightarrow\; \psi'\Bigg) \qquad\qquad \text{is valid}$$

Now, by considering the definition of Ψ, that $\alpha_2(\bigwedge_{x \in Voc(\psi)} j \neq x)$ simplifies to $true$, that $\alpha_2(\tau^{(j)}) = \tau[n]$, adding conjuncts to the antecedent (which keeps a valid implication valid), and that $\alpha_2(\psi') = \alpha(\psi')$:

$$\Psi \wedge true \wedge \tau[n] \rightarrow \alpha(\psi') \qquad\qquad \text{is valid}$$

Hence, in both cases $(\Psi \wedge \tau[n] \rightarrow \alpha(\psi'))$ is valid, which finishes the proof. □

Example 3.9 (Mutual Exclusion of SETMUTEX)
We revisit program SETMUTEX presented in Example 2.2 in page 21. Remember we use $\mathsf{active}(k)$ and $\mathsf{critical}(k)$ to denote $(pc(k) = 4 \vee pc(k) = 5 \vee pc(k) = 6)$ and $(pc(k) = 5 \vee pc(k) = 6)$ respectively.

Mutual exclusion for SETMUTEX is specified as the following 2-index formula:

$$\mathsf{mutex}(i, j) \overset{\text{def}}{=} \Big(i \neq j \rightarrow \neg(\mathsf{critical}(i) \wedge \mathsf{critical}(j))\Big)$$

Attempting to use the P-INV rule to prove $\Box\mathsf{mutex}$ fails due to offending transition $\tau_4^{(i)}$. For such transition, using P-INV, we need to verify the following verification condition:

$$\mathsf{mutex}(i, j) \wedge \left(\begin{array}{cc} pc(i) = 4 \wedge pc' = pc\{i \leftarrow 5\} & \wedge \\ ticket(i) = min & \wedge \\ pres(avail, min, ticket(i), ticket(j)) & \end{array}\right) \rightarrow \mathsf{mutex}'(i, j) \tag{3.2}$$

However, this verification condition in not valid. Two counter models that negate Equation (3.2) are:

$$pc(j) = 5 \wedge min = 1 \wedge avail = 2 \wedge ticket(i) = 1 \wedge ticket(j) = 3 \tag{3.3}$$

and

$$pc(j) = 5 \wedge min = 1 \wedge avail = 2 \wedge ticket(i) = 1 \wedge ticket(j) = 1 \tag{3.4}$$

Each of these models illustrate that the verification condition is not valid. Property mutex cannot be verified using P-INV because it is not inductive. The main problem is that the formula $\mathsf{mutex}(i, j)$ by itself does not include information about two important facts of the program. First, if a thread is in the critical section, then it owns the minimum announced ticket, unlike in the counter-model described by Equation (3.3). Second, the same ticket cannot be given to two different threads, unlike in the second counter-model described by Equation (3.4). Two new auxiliary support

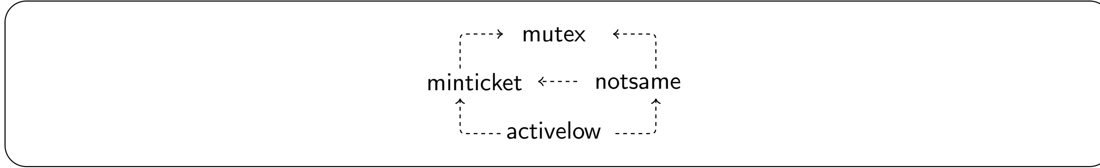


Figure 3.8: Proof graph showing dependencies between the invariants candidates for SETMUTEX.

invariants encode these facts:

$$\mathsf{minticket}(i) \stackrel{\text{def}}{=} \Big(\mathsf{critical}(i) \to min = ticket(i)\Big)$$
$$\mathsf{notsame}(i,j) \stackrel{\text{def}}{=} \Big(i \neq j \land \mathsf{active}(i) \land \mathsf{active}(j) \to ticket(i) \neq ticket(j)\Big)$$

Using SP-INV, we can now prove that mutex is invariant, using minticket and notsame as support, except for the fact that minticket is also not inductive. When trying to use P-INV to prove minticket invariant, the verification fails at the case in which two different threads i and j are in the critical section with the same ticket and $\tau_6^{(j)}$ is taken. When this happens, minticket(i) does not hold in the post state. Even if we use notsame as support for minticket, we still need to encode the fact that when a thread takes transition τ_3, it adds to bag a value strictly greater than any other previously assigned ticket. The following invariant rules out this spurious case:

$$\mathsf{activelow}(i) \stackrel{\text{def}}{=} \Big(\mathsf{active}(i) \to ticket(i) < avail\Big)$$

The formula activelow is inductive and can be proven directly using P-INV. Also, activelow is enough to support notsame and minticket, so the proof is completed.

Alternatively, we could use the G-INV rule and the proof graph depicted in Fig. 3.8. In this graph, invariant candidates mutex, minticket, notsame and activelow are the nodes, and the dashed arrows denote dependencies between them. ⌟

More detailed examples showing the application of parametrized invariance rules for the verification of safety properties of programs manipulating more complex data structures such as lists and skiplists are presented later in Chapter 10.

3.3 Summary

In this chapter we introduced a temporal deductive technique for the uniform verification problem of safety properties of parametrized infinite state processes, in particular for the verification of concurrent data types that manipulate data in the heap. The parametrized proof rules we presented automatically discharge a finite collection of verification conditions. A novelty is that the size of this collection of verification conditions depends on the program description and the index of the formula to prove, but not on the number of threads in a particular instance. Each verification condition describes a small-step in the execution of all corresponding instances. Moreover, the verification conditions are quantifier-free as long as the formulas are quantifier free.

Here we used the theory of arrays [37] to encode the local variables of a system with an arbitrary number of threads, but the dependencies with arrays can be eliminated under the assumption of full symmetry. The proof rules presented here are amenable for fully symmetric systems in which thread identifiers are only compared with equality, which encompasses many real systems. However, other topologies like rings of processes or totally ordered collections of processes can be handled with variations of our proof rules. In fact, it is immediate to extend our framework to a finite family of process classes, for example to model client/server systems.

The parametrized proof rules we presented here are designed for tackling the parametrized verification of safety properties. Later, in Chapter 4 we will introduce *Parametrized Verification Diagrams*, a formalism amenable for the verification of liveness temporal properties of parametrized systems.

In Chapters 6, 7 and 8 we will present decision procedures for theories of lists and skiplists, which can be used to automatically check the validity of the verification conditions generated by the proof rules presented in this chapter. Our parametrized proof rules has been implemented as part of LEAP, a theorem prover developed at the IMDEA Software Institute. LEAP is described in detail in Chapter 9. Finally, Chapter 10 presents some of the experimental results we have obtained using the parametrized invariance rules implemented in LEAP.

4

Parametrized Verification Diagrams

> *What about liveness?*
>
> César Sánchez

In chapter 3 we presented *parametrized invariance*, a deductive method to check *safety* properties of parametrized systems, that is, systems in which a given program is executed by an unbounded number of processes. Now, in this chapter we introduce *parametrized verification diagrams* (PVDS), a deductive formalism that allows to prove temporal properties, specially *liveness* properties of parametrized concurrent systems.

PVDS extend *Generalized Verification Diagrams* (GVDS) [39, 189]. A GVD encodes succinctly a proof that a non-parametrized reactive system satisfies a given temporal property. Even though GVDS are known to be sound and complete for non-parametrized systems, proving temporal properties of parametrized systems potentially requires to find a different a GVD for each instantiation with a concrete number of processes. In turn, each GVD would require to discharge and prove a different collection of verification conditions.

The PVDS we present here allow a *single* diagram to represent the proof that all instances of the parametrized system for an arbitrary number of threads running concurrently satisfy the temporal specification. To do so, PVDS exploit the symmetry assumption in a system, under which process identifiers are interchangeable. This assumption covers a large class of concurrent systems, including concurrent data types.

PVDS can be seen as a formula automaton with some extra notations which act as the witness of the proof that a parametrized system does satisfy a temporal property. Using a PVD, we generate a finite collection of quantifier-free verification conditions that can then be automatically proven with the assistance of an appropriate decision procedure. Hence, at the end, checking the

proof represented by a PVD reduces to proving only these finite set of quantifier-free verification conditions.

The rest of the chapter is structured as follows. Section 4.1 gives a brief introduction to *Generalized Verification Diagrams* and presents its drawbacks when dealing with parametrized systems. Section 4.2 formally presents *Parametrized Verification Diagrams*. Section 4.3 describes the quantifier-free verification conditions which are generated from a PVD, whose validity imply the correction of the proof described though the PVD, and presents the proof of soundness for PVDs. Finally, Section 4.4 presents a summary of what has been discussed in this chapter.

4.1 GVD vs. PVD: The Need of Parametrized Verification Diagrams

Generalized verification diagrams [39, 189] are a formalism to prove temporal specifications of reactive systems involving either, finite or infinite state. In particular, GVDs are suitable for the analysis of concurrent programs. A diagram is essentially an abstraction of the system and it is built specifically for the property under consideration. The diagram is precise enough to formally represent the temporal proof, and it allows the mechanical check of the correctness of the proof.

Formal verification using general verification diagrams starts from a program and a specification in linear temporal logic. The semantics of the program are represented as a *fair transition system* (FTS) that encompasses all executions of the program. In a nutshell, a verification diagram encodes a proof that all the executions covered by the FTS satisfy the given temporal property. Given a program P and a temporal specification φ, as shown in Fig. 4.1, a GVD $\mathcal{D}$ acts as a witness of the proof that in fact all fair traces of P satisfies the temporal specification φ. Hence, checking the proof encoded in the diagram requires two activities:

1. Check the validity of a finite collection of verification conditions, which are automatically generated from the program and the diagram. This guarantees that the diagram covers all (fair) executions of the system.
2. Execute a finite state model checking algorithm to ensure that every fair path of the diagram satisfies the temporal property. This analysis can be fully automated using model checkers.

The first part can be handled using suitable decision procedures for the underlying data that the program manipulates, like Boolean, integers, lists in the heap, skiplists, etc. This way, GVDs

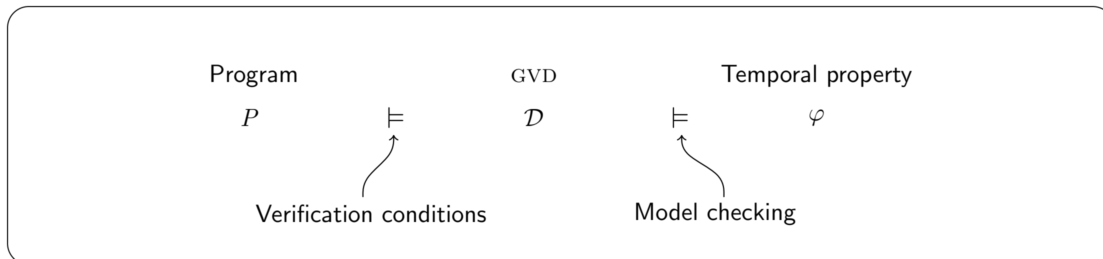


Figure 4.1: Schematic representation of the use of a GVD $\mathcal{D}$ to prove that program P satisfies the temporal property φ.

cleanly separates two concerns:

1. the temporal reasoning, and
2. the data-manipulation

GVDs are complete in the following sense: if a (closed) reactive system satisfies a given temporal property then there is a diagram that encodes a proof. Unfortunately, GVDs cannot be used directly to verify concurrent programs that involve an arbitrary number of threads, which are naturally modeled as parametrized systems where the parameter is the number of threads involved. This problem arises because each instantiation of the parameter produces a different closed system, and in turn a different FTS. Hence, in principle, each closed system requires finding a different diagram, discharging a different collection of verification conditions, and solving a different model-checking problem.

Example 4.1
Consider the mutual exclusion protocol SETMUTEX presented in Example 2.2 of Section 2.1.1 in page 21. The protocol ensures mutual exclusion using a ticketing system. In this protocol, any thread with the intention to enter the critical section is assigned an unique increasing ticket number. Additionally, a copy of such ticket is stored in a shared set of tickets. Before accessing the critical section, a thread needs to wait until its ticket matches the minimum ticket in the shared set. Hence, only threads with the minimum ticket are allowed to enter the critical section. After one such thread leaves the critical section, it removes its ticket from the set so that a new thread can access later the critical section.

Now, consider a system composed by only 2 threads, T_1 and T_2, running program SETMUTEX. We use the following notation:

- $\mathsf{wants}(t)$, to denote that thread t is about to the get a ticket, so that it wants to access the critical section. That is, $pc(t) = 3$ in SETMUTEX.
- $\mathsf{awaits}(t)$, to represent the fact that threat t is waiting for its turn to access the critical section. This corresponds to $pc(t) = 4$ in SETMUTEX.
- $\mathsf{critical}(t)$, to denote that thread t is inside the critical section. That is $pc(t) = 5 \vee pc(t) = 6$.

A temporal property we would like to verify is that if thread T_1 requires access to the critical section (i.e., claims a ticket) then it eventually gains access to the critical section. This in fact is a liveness property we can describe with the following temporal specification:

$$\mathsf{eventually_critical_T1} \quad = \quad \square\,(\mathsf{wants}(T_1) \rightarrow \diamond\mathsf{critical}(T_1))$$

A simple GVD to prove this property is sketched in Fig. 4.2. A real GVD contains more elements than the ones depicted in Fig. 4.2, but this simplified GVD will help us understand where GVDs fail when dealing with an unbounded number of threads. It should also be clear from the figure that a diagram naturally follows the reasoning about the data structure user. That makes GVDs a simple formalism to verify temporal properties.

The initial node of the diagram presented in Fig. 4.2 is n_0. This node presents the situation in which thread T_1 has still not shown interested in entering the critical section. When thread

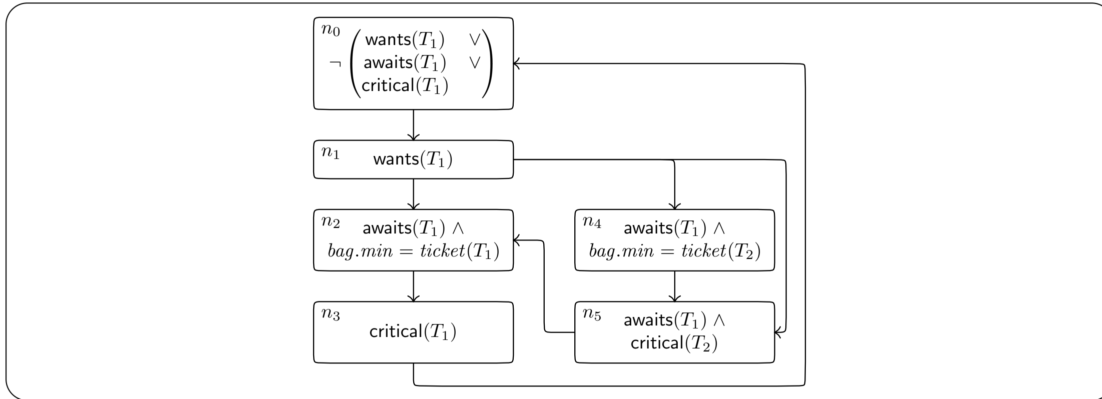


Figure 4.2: Simplified GVD for proving eventually_critical_T1 in a system with two threads.

T_1 reaches line 3 of SETMUTEX, a ticket ($ticket(T_1)$) is assigned to thread T_1 and a copy of this ticket is included into the bag. This situation is depicted by node n_1 in the diagram. Once line 3 of SETMUTEX is executed by thread T_1, for sure awaits(T_1) holds. Now, depending on whether thread T_1 has the minimum ticket in the bag or not, there are two possible scenarios:

1. if thread T_1 has the minimum ticket in bag (node n_2), then thread T_1 will enter the critical section (node n_3), satisfying the eventually_critical_T1 property stated above.

2. if thread T_2 has the minimum ticket of bag, because T_2 required access to the critical section before T_1, then we are in one of the situations described by node n_4 or n_5. That is, thread T_1 must wait while thread T_2 has the minimum ticket and T_2 is about to enter the critical section (node n_4) or thread T_2 is already in the critical section (node n_5). In any case, thread T_1 needs to wait until thread T_2 comes out of the critical section and removes its ticket from the bag. Only after thread T_2 removes its ticket from bag, thread T_1 will have the lowest ticket in bag and thus T_1 will be allowed to enter the critical section.

Now, imagine we would like to consider the scenario of a system composed by three different threads: T_1, T_2 and T_3. For a system with 3 threads, the diagram shown in Fig. 4.2 is no longer suitable, as we would require to construct a similar, but slightly different new diagram. The simplified GVD for a system with 3 threads is shown in Fig. 4.3. Note that, basically, we were required to duplicate the reasoning associated to nodes n_4 and n_5 to the new nodes n_6 and n_7. This change is necessary as we need to consider now the case in which thread T_3 has the minimum ticket of the bag.

As should result evident at this point, the problem is that every time we consider a system with one more thread, a slightly different GVD is required. Hence, GVDs do not scale well when dealing with parametrized systems composed by an unbounded number of threads. Even more, we have made theses diagrams to prove property eventually_critical_T1, which is associated to the termination of just thread T_1. What if we want to prove that any thread that wants to access the critical section eventually succeeds? We would require an even more complex diagram which should be parametrized by a thread identifier, something that is not currently possible in GVDs.⌟

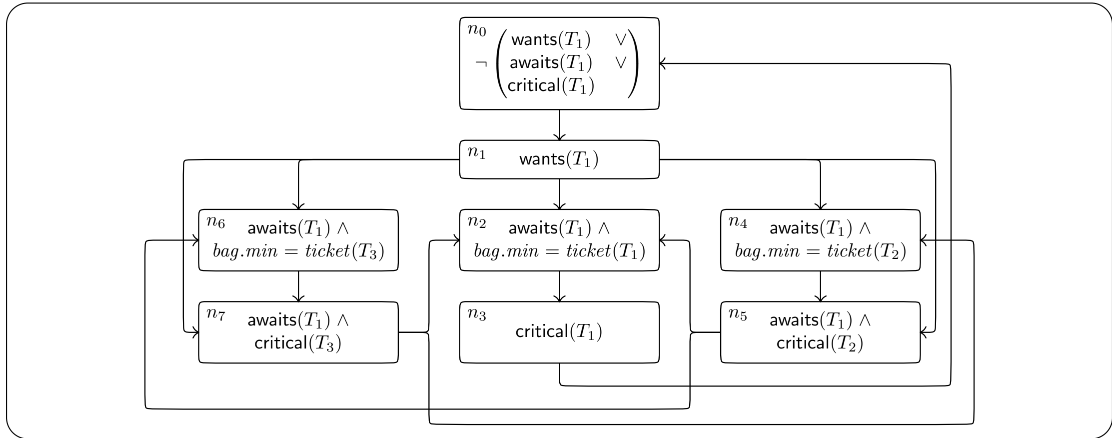


Figure 4.3: Simplified GVD for proving eventually_critical_T1 in a system with three threads.

In order to be able to verify parametrized temporal properties of systems composed by an unbounded number of threads we develop the idea of *Parametrized Verification Diagrams* (PVDs). PVDs enrich verification diagrams with capabilities to reason about executions with an arbitrary number of *symmetric* threads. Checking the proof represented by a PVD requires to handle a single finite collection of verification conditions and to solve a single finite-state model-checking problem. Success in proving each of these obligations guarantees that the property holds for *all* parameter instances.

The key idea behind PVDs is that a proof usually only requires to reason about a finite number of processes at each time instant. This finite collection includes:

1. the processes referred to in the property;
2. some other processes with particular remarkable characteristic in the given state (like a leader); and
3. one fresh thread identifier representing an arbitrary process that executes a small step in the global execution.

As happens with the *Parametrized Invariance* rules presented in Chapter 3, the PVDs we present in this chapter rely on the symmetry assumption. This assumption states that process identifiers are interchangeable and are only compared for equality and inequality. Under this assumption, swapping identifiers in a given legal execution produces another legal execution. Even though some protocols are not symmetric, full symmetry covers an important class of concurrent systems: concurrent data types [107]. In this line of work we propose PVD as a formalism aiming the verification of liveness properties of concurrent data types.

4.2 Parametrized Verification Diagrams

In this section, we formally define *Parametrized Verification Diagrams* and we present them as an effective method to solve the parametrized temporal verification problem, specially for liveness temporal properties. The aim of PVDs is to capture formally the proof that *all instances* of a

parametrized program satisfy a temporal specification. Essentially, for each value of M, the diagram over-approximates the set of runs of $\mathcal{S}[M]$. In turn, fair runs of the diagram are covered by the executions allowed by the temporal formula.

4.2.1 Definition of PVD

Given a parametrized temporal formula $\varphi(\overline{k})$ and a parametrized system $\mathcal{P}$, a PVD is a tuple $\langle N, N_0, E, \mathcal{B}, \mu, \eta, \mathcal{F}, f \rangle$ where:

- N is the finite set of nodes that conform the PVD.
- $N_0 \subseteq N$ is the set of initial notes, which is a subset of all PVD nodes N.
- $\mathcal{B}$ is a finite collection of pairs $\{(\mathcal{B}_1, b_1), \dots, (\mathcal{B}_q, b_q)\}$, where $\mathcal{B}_i \subseteq N$, $\mathcal{B}_i \cap \mathcal{B}_j = \emptyset$ for $i \neq j$, and b_i is a fresh tid variable. Each pair $(\mathcal{B}_i, b_i)$ is called a *box* and the set $V_{box} = \{b_1, \dots, b_q\}$ is called the set of box variables. Boxes group nodes and label them with thread identifiers. While inside a box, the thread identifier b_i labeling the box is preserved and it can only be modified when entering a box from an outside edge. Finally, we use $V_{tidParams}$ to denote the set of thread identifiers formed by the parameters of the parametrized temporal formula $\varphi(\overline{k})$ and the thread identifiers which label boxes in a PVD. That is, $V_{tidParams} = \overline{k} \cup V_{box}$.
- E is a finite set of edges, each connecting two nodes. Edges are equipped with the following functions and predicates:
  - *in* $: E \to N$ and *out* $: E \to N$, which given an edge indicates the incoming and outcoming node respectively for such edge.
  - *within* $\subseteq E$, is a predicate which indicates whether a transition modeled by an edge which connects two nodes within the same box must preserve the box variable. That is, for all $e \in$ *within*, we require that there exists a box $\mathcal{B}_i$ such that both *in*$(e) \in \mathcal{B}_i$ and *out*$(e) \in \mathcal{B}_i$. In the case that $e \in$ *within*, then the box variable i must be preserved when edge e is taken. Otherwise the box variable i can arbitrarily change.
- μ is a labeling function for nodes which assigns to each node n a formula $\mu(n)$ in the theory $\mathcal{F}_T(\overline{k} \cup V_{box})$, with the restriction that $\mu(n)$ can only contain b_i whenever node n is in box $\mathcal{B}_i$.
- $\eta : E \rightharpoonup \mathcal{T} \times V_{tidParams}$ is a partial function labeling some edges with transitions of the system to indicate that these edges label fair transitions. That is, all these transitions must be taken by fairness.
- $\mathcal{F}$ is the acceptance condition of the diagram. It consists of a finite collection:

  $$\langle \langle B_1, G_1, \delta_1 \rangle \ \dots \ \langle B_m, G_m, \delta_m \rangle \rangle$$

  Each triplet acceptance condition $\langle B_i, G_i, \delta_i \rangle$ is formed by an edge Streett condition $B_i, G_1 \subseteq E$ and a ranking function $\delta : N \to \mathcal{O}$, where $\mathcal{O}$ is a well founded domain. The Streett condition requires that some edge in G_i is visited infinitely often or all edges in B_i are visit only finitely often. Without loss of generality we can assume $G_i \cap B_i = \emptyset$. Edges in G_i are called *good* edges, and edges in B_i are called *bad* edges.

- f is a map from nodes into Boolean combinations of elementary propositions from $\varphi(k)$.

Additionally, we ask for an extra requirement over the transitions labeling edges which come out of boxes. Transitions labeling an edge according to η can be parametrized by a thread identifier in $\overline{k}$. Otherwise, the transition is allowed to be parametrized by a box variable b_i only if the edge begins at a node belonging to the box $(\mathcal{B}_i, b_i)$.

Conceptually, once a parameter instance M is fixed, every box can be populated M times, assigning in each expansion one of the possible tid values within $[M]$ to the box variable. The resulting diagram is a conventional non-parametrized generalized verification diagram.

Example 4.2
Consider again the example of a GVD for the eventually_critical_T1 property we presented in Example 4.1. The GVD shown in Fig. 4.4 was designed for a system involving only 2 threads. Later, in Fig. 4.3 we considered a new GVD for a system made of 3 threads. We now present, in Fig. 4.4 a PVD that encodes the satisfaction of the temporal formula eventually_critical_T1 in a system composed by an unbounded number of threads executing program SETMUTEX.

Note that, according to our definition of a PVD, formulas labeling nodes can be parametrized only by box variables and threads parametrizing the temporal formula under verification. Clearly, specification eventually_critical_T1 is not parametrized, as it only represents the termination of thread T_1. Equation 4.1 presents a parametrized version of specification eventually_critical_T1:

$$\mathsf{eventually_critical}(k) \ = \ \Box\,(\mathsf{wants}(k) \rightarrow \Diamond \mathsf{critical}(k)) \tag{4.1}$$

Note that, in fact, eventually_critical_T1 is eventually_critical(T_1).

Note that the structure of the diagram presented in Fig. 4.4 is very similar to the GVD depicted in Fig. 4.2. A key difference is that in the PVD, the formulas labeling nodes n_0, n_1, n_2, n_3, n_4 and n_5 are now parametrized by k (the parameter of eventually_critical(k)) instead of T_1. Another difference is the presence of a *box* surrounding nodes n_4 and n_5. Such box is labeled with the box variable b. This means that b is preserved by all transitions within the box. As b can be assigned any thread identifier in the system, if b is T_2 in a given execution state then the box captures the situation in which T_2 has the minimum ticket in the bag. The edge connecting node n_1 to the box

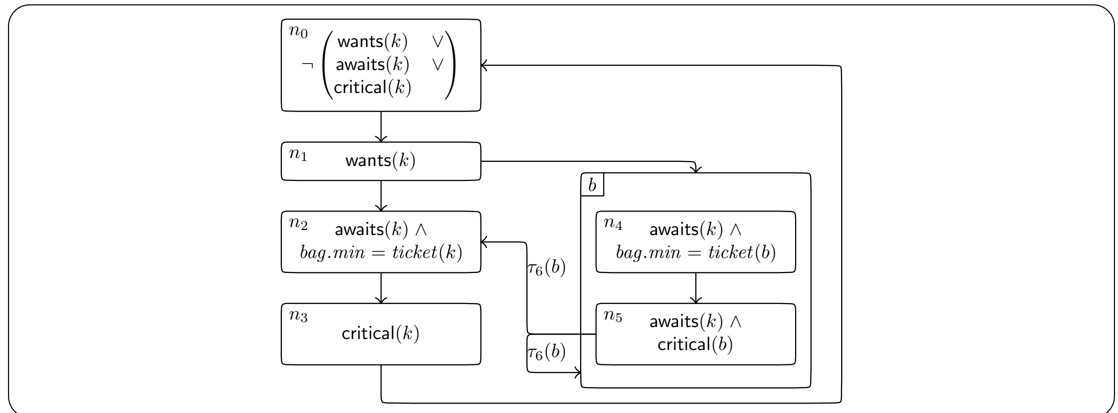


Figure 4.4: A PVD for verifying property of a system with an unbounded number of threads.

in fact symbolizes two edges, one connecting node n_1 to node n_4 and another edge connection node n_1 with node n_5.

Now, consider the two edges labeled with transition $\tau_6(b)$. We are allowed to label these edges with a transition parametrized by b since in both cases their incoming node (n_5) resides within a box whose variable is b. Each of these two labeled edges represent:

1. the edge that connects node n_5 to node n_2, denotes the situation in which a thread different from k leaves the critical section and removes its ticket from the bag. At this moment, thread k becomes the thread with the lowest ticket in the bag.

2. the edge connecting node n_5 with the bag represents the situation in which a new thread —different from k— becomes the thread with the lowest ticket in the bag. Note that as the edge is leaving the box, we have that the value of b in the pre-state and in the post-state does not need to be the same. This is exactly modeling the situation in which a new thread, different from k or the thread leaving the critical section, is taking the role of being the thread with the lowest ticket in the bag.

Finally, note how a PVD represents many GVDs for a system with an unbounded number of threads. If we consider parameter k to be T_1 and we instantiate the box only once (replacing b with T_2) or twice (by replacing b by T_2 and T_3) we obtain the GVD for system with 2 and 3 threads respectively. ⌟

In a PVD, the intended meaning of each edge Streett condition $\langle B_i, G_i, \delta_i \rangle$ is to ensure that in any accepting trail of the diagram either some edge from G_i is visited infinitely often, or all edges from B_i are visited finitely often.

A *path* in the diagram is a sequence of states and edges $n_0 e_0 n_1 e_1 \ldots$ such that for every i, $n_i \rightarrow_{e_i} n_{i+1}$. A path is *fair* whenever if after some point i all nodes n_i in the diagram have an outgoing edge labeled with (τ, v) then edges labeled (τ, v) are taken infinitely often. A path is *accepting* whenever for every acceptance condition (B_i, G_i, δ_i) either all edges from B_i are traversed finitely often, or some edge from G_i is traversed infinitely often.

Given a concretization function $\alpha : \overline{k} \rightarrow [M]$ for some concrete system $\mathcal{S}[M]$ and a path π of the diagram, we define an extended concretization of the path as a sequence of functions $\alpha_i : (\overline{k} \cup V_{box}) \rightarrow [M]$ that coincide with α on all $k \in \overline{k}$, and such that if $e_i \in$ *within* then $\alpha_{i+1} = \alpha_i$. Essentially, the extended concretizations choose concrete indices for the box variables whenever these are free to choose.

Given a run $\pi : s_0 \tau_0 s_1 \tau_1 \ldots$ of a concrete instance $\mathcal{S}[M]$ and a concretization $\alpha : \overline{k} \rightarrow M$, a path $d = n_0 e_0 n_1 e_1 \ldots$ of $\mathcal{D}$ is a *trail* of π whenever for some extended concretization $\{\alpha_i\}$, the following holds:

$$s_i \models \alpha_i(\mu(n_i)) \qquad \text{for all} \qquad i \geq 0$$

A run π is a *computation* of $\mathcal{D}$ if there exists a trail of π that is fair and accepting. $\mathcal{L}^{[M]}(\mathcal{D})$ denotes the set of computations of $\mathcal{D}$ for parameter instance M (i.e., sequences of states of $\mathcal{S}[M]$ accepted by $\mathcal{D}$).

In the next section we will list a collection of verification conditions extracted from the diagram, and we will show that proving the validity of these verification conditions implies that

all computations of $\mathcal{S}[M]$ are in $\mathcal{L}^{[M]}(\mathcal{D})$. Given a concrete instance $\mathcal{S}[M]$ and a concretization $\alpha : \overline{k} \to [M]$, a sequence $P_0P_1\dots$ of elements from concrete elementary propositions of $\alpha(AP(\varphi))$ is a propositional model of $\mathcal{D}$ whenever there is a fair and accepting path $\pi : n_0e_0n_1\dots$ of $\mathcal{D}$ for which $P_i \vDash \alpha(f(n_i))$. We use $\mathcal{L}_p^{[M]}(\mathcal{D})$ to denote the set of propositional models of $\mathcal{D}$ (for $\mathcal{S}[M]$). Again, we will show that checking all verification conditions implies that for all $\mathcal{S}[M]$ and concretizations α, every sequence of elementary propositions of a run of $\mathcal{S}[M]$ is included in $\mathcal{L}_p^{[M]}(\mathcal{D})$. We use $\mathcal{L}^{[M]}(\varphi)$ for $\cup_{\alpha:\overline{k}\to[M]}\mathcal{L}(\alpha(\varphi))$. Finally, we will also show that every trace in $\mathcal{L}_p^{[M]}(\mathcal{D})$ for concretization α is included in $\alpha(\varphi)$, that is $\mathcal{L}_p^{[M]}(\mathcal{D}) \subseteq \mathcal{L}^{[M]}(\varphi)$.

4.3 Verification Conditions for PVD

As we mentioned, a PVD shows that all instances $\mathcal{S}[M]$ of a parametrized system $\mathcal{P}$ satisfies a temporal parametrized specification $\varphi(\overline{k})$ by checking the inclusions:

- All reachable states by the program are included into the parametrized diagram: $\mathcal{L}(\mathcal{S}[M]) \subseteq \mathcal{L}^{[M]}(\mathcal{D})$; and
- The propositional language described by the parametrized diagram satisfy the parametrized temporal specification: $\mathcal{L}_p^{[M]}(\mathcal{D}) \subseteq \mathcal{L}^{[M]}(\varphi(k))$.

Theorem 4.1 below shows that to prove $\mathcal{L}(\mathcal{S}[M]) \subseteq \mathcal{L}^{[M]}(\mathcal{D})$ it is enough to prove the verification conditions we present in this section.

The main difficulty is to define a *finite* number of verification conditions that guarantee the previous language inclusion. To do so, we rely on the definition of the vocabulary of a formula, introduced in Section 2.4.5. Note that the vocabulary represents the set of variables of type tid whose modification can potentially alter the truth value of a given formula. We now define the vocabulary of a node as $NVoc(n) = \{b_i \mid n \in \mathcal{B}_i\} \cup \overline{k}$. Given a node n, let $next(n) = \{n' \in N \mid \text{for some } n \to_e n' \in E\}$.

In the verification conditions listed below, given an edge $e \in E$ such that $in(e), out(e) \in \mathcal{B}_i$, we define $boxed(e)$ as the formula such that:

$$boxed(e) \quad = \quad \begin{cases} b_i' = b_i & \text{if } e \in \textit{within} \\ true & \text{otherwise} \end{cases}$$

Given a parametrized transition system $\mathcal{P}$, a parametrized temporal formula $\varphi(\overline{k})$ and a parametrized verification diagram $\mathcal{D}$, diagram $\mathcal{D}$ shows that for all instances $\mathcal{S}[M]$ of parametrized system $\mathcal{P}$ it holds that $\mathcal{S}[M] \vDash \varphi(\overline{k})$ whenever all these conditions hold:

Initiation: Condition (Init), known as initiation, states that at least one initial node in N_0 satisfies the initial condition of $\mathcal{P}$:

$$\Theta \to \mu(N_0) \qquad \text{(Init)}$$

Consecution: Consecution ensures that every node in the diagram has a τ-successor. Consecution is expressed by two different kind of conditions:

- Condition (SelfConsec), called self-consecution, establishes that any τ-successor of a state satisfying $\mu(n)$ satisfies the label of some successor node of n. In other words, the diagram can always move when taking any enabled transition by any thread mentioned in the property. Formally speaking, condition (SelfConsec) states that for every node $n \in N$ and for all $i \in NVoc(n)$:

$$\bigvee_{n \to_e m} \mu(n) \wedge \tau(i) \wedge boxed(e) \to \mu'(m) \qquad \text{(SelfConsec)}$$

- Condition (OtherConsec), called others-consecution, is analogous to (SelfConsec), with the difference that it considers transitions taken by an arbitrary thread not mentioned in the vocabulary of $\varphi(\overline{k})$ or used as argument of any of the boxes in the diagram. This condition is the key to guarantee that only a finite number of verification conditions is necessary, because this condition encompasses all other threads not mentioned in the formula (or in boxes). Formally, condition (OtherConsec) states that for every $n \in N$ and for a fresh $j \notin NVoc(n)$:

$$\bigvee_{n \to_e m} \mu(n) \wedge \tau(j) \wedge boxed(e) \wedge \bigwedge_{i \in \mathcal{V}} i \neq j \to \mu'(m) \qquad \text{(OtherConsec)}$$

Acceptance: Conditions (SelfAcc) and (OtherAcc), called self-acceptance and others-acceptance respectively, guarantee the acceptance condition of the diagram through the verification of ranking functions. Intuitively speaking, these verification conditions use information extracted from the data in the system to infer that certain sequences of states must be terminating. For example, this is the manner in which one checks that at most a finite number of threads can out-run a given thread when entering the critical section. These conditions guarantee that the ranking function δ_i is (strictly) decreasing in B_i edges, and non-increasing in edges $E - (G_i \cup B_i)$. We use P_i to denote edges in $E - (G_i \cup B_i)$, called *permitted edges*.

Formally, for each $(B, G, \delta) \in \mathcal{F}$ and edge $n \to_e m$, condition (SelfAcc) states that for all $i \in NVoc(n)$:

$$\begin{array}{ll} \Big(\mu(n) \wedge \tau(i) \wedge \mu'(m) \wedge boxed(e) \Big) \to \delta(n) \succ \delta(m) & \text{if } e \in B \\ \Big(\mu(n) \wedge \tau(i) \wedge \mu'(m) \wedge boxed(e) \Big) \to \delta(n) \succeq \delta(m) & \text{if } e \in E \setminus (G \cup B) \end{array}$$

Similarly, condition (OtherAcc) states that for a fresh $j \notin NVoc(n)$:

$$\begin{array}{ll} \Big(\mu(n) \wedge \tau(j) \wedge \bigwedge_{i \in NVoc(n)} i \neq j \wedge \mu'(m) \wedge boxed(e) \Big) \to \delta(n) \succ \delta(m) & \text{if } e \in B \\ \Big(\mu(n) \wedge \tau(j) \wedge \bigwedge_{i \in NVoc(n)} i \neq j \wedge \mu'(m) \wedge boxed(e) \Big) \to \delta(n) \succeq \delta(m) & \text{if } e \in E \setminus (G \cup B) \end{array}$$

If the verification conditions for δ are valid, infinite trails either traverse G_i edges infinitely often, or traverse B_i edges only finitely often. This second case holds because (1) the domain is well-founded, (2) permitted edges are non-increasing, and (3) bad edges are

decreasing.

Fairness: In order to verify the fairness of the diagram, we need to check the validity of the following two conditions. Condition (En) establishes that any transition labeling an edge coming out from a node must be enabled at every state modeled by the node. That is, for each edge $e = (n, m)$ and $\tau(i) \in \eta(e)$:

$$\mu(n) \to En(\tau(i)) \tag{En}$$

On the other hand, condition (Succ) establishes that if a transition labeling an edge is taken at the incoming node, then all edge labels cover the possible actions of the transitions. That is, for each edge $e = (n, m)$ and $\tau(i) \in \eta(e)$:

$$\mu(n) \wedge \tau(i) \to \bigvee_{\tau(i) \in \eta(n \to_e m)} \mu'(m) \tag{Succ}$$

The combination of (En) and (Succ) guarantee that a label τ is always enabled at the given nodes and that the only way to exit the nodes taking $\tau(i)$ is through the label edges. This relates fairness in any concrete system with fairness in the diagram.

Satisfaction: Finally, satisfaction ensures that the diagram satisfies the temporal parametrized specification $\varphi(\overline{k})$. Satisfaction consists of two conditions. Condition (Prop) guarantees the correctness of the propositional models of the diagram. That is, for all $n \in N$:

$$\mu(n) \to f(n) \tag{Prop}$$

Condition (ModelCheck) ensures that propositional models of the diagram are included in traces of the property $\varphi(\overline{k})$.

$$\mathcal{L}_p^{[M]}(\mathcal{D}) \subseteq \mathcal{L}^{[M]}(\varphi(\overline{k})) \tag{ModelCheck}$$

The propositional label f of the diagram allows us to use a single query to a finite state model-checker to show whether condition (ModelCheck) is satisfied. Appendix A shows how to check that the diagram $\mathcal{D}$ satisfies the temporal property $\varphi(\overline{k})$ using model checking.

For a parametrized system $\mathcal{P}$, a formula $\varphi(\overline{k})$ and a PVD $\mathcal{D}$, if all verification conditions described above hold we say that $\mathcal{D}$ is $(\mathcal{P}, \varphi)$-valid.

Note that in every case, there is finite number of verification conditions. In particular, there are $|N|(|V_{tidParams}| + 1)$ conditions for *consecution* and at most $|\mathcal{F}||E|(|V_{tidParams}| + 1)$ conditions for *acceptance*. Finally, the number of conditions needed to verify *fairness* is limited by the number of edges, program lines and thread identifiers in the vocabulary of the formulas labeling nodes in each box.

We enunciate the main result of this section, which ensures that PVDs are sound for proving that a parametrized system satisfies a parametrized temporal specification.

Theorem 4.1 (Soundness):
Let $\mathcal{P}$ be a parametrized system and $\varphi(\overline{k})$ a temporal formula. If there exists a $(\mathcal{P}, \varphi)-valid$ PVD, then $\mathcal{P} \vDash \varphi$.

Proof. We start by assuming that there is a $(\mathcal{P},\varphi)-valid$ PVD $\mathcal{D}$, and show that $\mathcal{P} \vDash \varphi$. This requires showing that for an arbitrary M and concretization $\alpha : \overline{k} \to [M]$, it holds that $\mathcal{S}[M] \vDash \alpha(\varphi(\overline{k}))$. In the proof, we will use repeatedly Lemma 2.1 from Chapter 2 which ensures that if $\psi(\overline{k})$ is a parametrized (non-temporal) formula and α is a concretization then, if $\psi(\overline{k})$ is valid, so is $\alpha(\psi(\overline{k}))$.

Let M be an arbitrary bound and α an arbitrary concretization function. We consider an arbitrary run (that is, a fair computation) of $\mathcal{S}[M]$: $\sigma : s_0\tau_0[i_0]s_1\tau_1[i_1]\ldots$ and show that $\sigma^p \vDash \alpha(\varphi)$, where σ^p is the projection of σ on the propositional alphabet of $\alpha(\varphi)$.

We first consider an extension of α such that *Img*$(\alpha) = M$ by adding one fresh thread identifier i for each $k \in M$ not mapped by the original alpha and making $\alpha(i) = k$. In this manner, all elements of M have at least one representative thread identifier (not necessarily in $\overline{k}$).

First, we show by induction that there is a path $\pi : n_0e_0n_1e_1n_2\ldots$ of σ in the diagram, and a sequence of thread identifiers $j_0j_1\ldots$ such that $\alpha(j_k) = i_k$ — that is, the identifier of the thread taking the k-th step in the execution $s_k\tau_k[i_k]s_{k+1}$ — and $s_i \vDash \alpha(\mu(n_i))$. It is enough to prove that there is a trail of nodes n_k of the diagram and a extended concretization α_k such that

1. $s_k \vDash \alpha_k(\mu(n_k))$, and
2. $\tau_k^{j_k}$ can be taken to traverse edge e_k. That is, $\neg(\mu(n_k) \wedge \tau_k^{j_k} \wedge \mathit{boxed}(e_k) \to \mu'(n_{k+1}))$ is not valid.

We build the trace by induction:

- Base case: The base case of induction follows from condition (Init). Since $\Theta \to \mu(N_0)$ is valid, then $\alpha(\Theta \to \mu(N_0))$ is valid, and $\alpha(\Theta) \to \alpha(\mu(N_0))$ is valid. Hence, since $s_0 \vDash \alpha(\Theta)$ it follows that $s_0 \vDash \alpha(\mu(N_0))$ and for some $n_0 \in N_0$, $s_0 \vDash \alpha(\mu(n_0))$ as desired.
- Induction step: Let n_k be the last node of the trail, α_k be the extended concretization, and j_k be a thread identifier for which $\alpha_k(j_k) = \alpha(j_k) = i_k$. We consider the cases for the outgoing transition $\tau_k(j_k)$ from n_k:
  - if j_k is referred to in the property, from condition (SelfConsec) we have that

$$\bigvee_{n_k \to_e n_{k+1}} \mu(n_k) \wedge \tau(j_k) \wedge \mathit{boxed}(e) \to \mu'(n_{k+1})$$

    is valid, so the following is also valid

$$\alpha_k\Big(\bigvee_{n_k \to_e n_{k+1}} \mu(n_k) \wedge \tau(j_k) \wedge \mathit{boxed}(e) \to \mu'(n_{k+1})\Big)$$

    or equivalently

$$\bigvee_{n_k \to_e n_{k+1}} \alpha_k(\mu(n_k)) \wedge \tau[i_k] \wedge \mathit{boxed}(e) \to \alpha_k(\mu'(n_{k+1})))$$

    is valid. Now, since $s_k \vDash \alpha_k(\mu(n_k))$, and (s_k, s_{k+1}) is a model of the last formula (possibly for a different value of box if $\mathit{boxed}(e)$ is true), for at least one of the conjuncts $s_{k+1} \vDash \alpha_{k+1}(\mu'(n_{k+1})$. This conjunct provides the edge e_k, the successor n_{k+1} and the value of the box for α_{k+1}.

– the case for condition (OtherConsec) follows similarly.

We now show that the trail $\pi : n_0 e_0 n_1 \ldots$ with transitions $\tau_k^{(j_k)}$ is a fair trail of the diagram. We prove it by contradiction. Assume trail π is not fair for transition τ taken by thread identifier i, which is enabled continuously but not taken. Then, there is a position j in the path π after which, for all successive states $k > j$, the node n_k of the path has an outgoing edge labeled $\tau(i)$ but $\tau_k^{(j_k)}$ in the path is not $\tau(i)$. Now, by verification conditions (En) and (Succ), there is a successor in the diagram for $\tau(i)$, and $\tau(i)$ is enabled. By taking α on these two verification conditions it follows that $\tau[\alpha(i)]$ is enabled in s_k and, s_k has a $\tau[\alpha(i)]$ successor in $\mathcal{S}[M]$ but $\tau[\alpha(i)]$ is not taken. Hence σ is not a fair run of $\mathcal{S}[M]$, which contradicts our assumption that σ is a computation.

We now check that the trail π is accepting. Again, we proceed by contradiction. Assume π is not accepting and let (B_i, G_i, δ_i) be the offending acceptance condition. This means that after some position j, for all $k > j$, only edges $e_k \notin G_i$ are visited, and some edges in B_i are seen infinitely often. This means, by conditions (SelfAcc) and (OtherAcc), that $\delta(n_k) \succeq \delta(n_{k+1})$ and for infinitely many $r > j$: $\delta(n_r) \succ \delta(n_{r+1})$. Hence, there is an infinite descending chain in a well-founded domain, which is a contradiction. This shows that $\sigma \in \mathcal{L}^{[M]}(\mathcal{D})$.

Finally, condition (Prop) ensures that $s_k \vDash \alpha_k(\mu(n_k))$ and since $\alpha_k(\mu(n_k) \to f(n_k))$ is valid, then $s_k \vDash \alpha_k(f(n_k))$. Hence, σ^p is in $\mathcal{L}_p^{[M]}(\mathcal{D})$. Finally, by (ModelCheck), $\mathcal{L}_p^{[M]}(\mathcal{D}) \subseteq \mathcal{L}(\alpha(\varphi))$. This finishes the proof. □

Example 4.3

We now illustrate the use of PVDs by presenting the full diagram for the response property eventually_critical(k), described by Equation 4.1 in page 77, for the mutual exclusion protocol SETMUTEX presented in Example 2.2.

In order to verify the protocol, we use a slightly modified version of the program. In the modified version, we keep a set of pairs instead on just the set of tickets. Pairs contain the ticket

```
   global
     Int avail := 0
     Set⟨Int, Tid⟩ bag := ∅

   procedure SetMutex()
     Int ticket := 0
   begin
1: while true do
2:     noncritical
3:     ⟨ ticket := avail++
         bag.add(ticket, me) ⟩
4:     await (bag.min == ticket)
5:     critical
6:     bag.remove(ticket, me)
7: end while
   end procedure
```

Figure 4.5: SETMUTEX modified for proving eventually_critical(k).

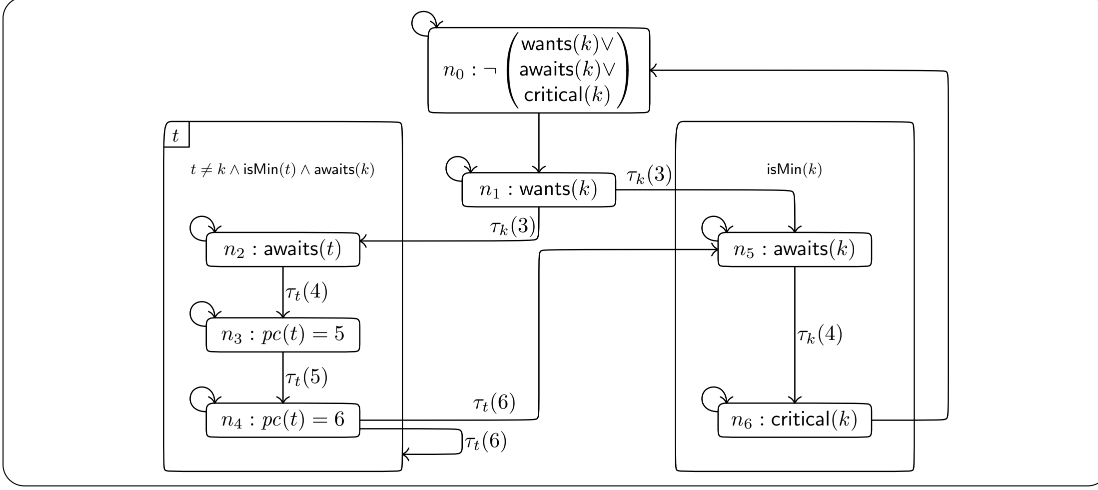


Figure 4.6: PVD for the proof that SETMUTEX satisfies eventually_critical(k).

and the thread identifier of the thread owning such ticket. Fig. 4.5 shows the modified version of the SETMUTEX program.

In order to verify this program, we rely on the theory of finite sets of pairs of integers and thread identifiers with ordered comprehension and minimum value. Given a pair p, function $\pi_{int}(p)$ returns the integer component of p while function $\pi_{tid}(p)$ returns the thread identifier component of pair p. In this theory, the function $lower(s, n)$ receives a set of pairs s and an integer n, and returns the subset of pairs whose first component is strictly lower than n. Additionally, this theory also provides a function $minPair$ that returns the lowest value in a set of pairs, using the integer component for comparison. If more than one pair satisfy the condition to be the lowest elements in a set, then one of such pairs can be arbitrarily returned.

We now present the PVD that represents the desired proof. In the diagram, we use isMin(i) for $\pi_{int}(minPair(bag)) = ticket(i)$. That is, if isMin($i$) holds, then thread i has the minimum ticket in the set bag. The diagram is depicted in Fig. 4.6. Formally, the PVD is defined by:

$$
\begin{aligned}
N &\stackrel{\text{def}}{=} \{n_i \mid 0 \le i \le 6\} \\
N_0 &\stackrel{\text{def}}{=} \{n_0\} \\
E &\stackrel{\text{def}}{=} \{n_0 \to n_1, n_2 \to n_3, n_3 \to n_4, n_5 \to n_6, n_6 \to n_0\} \\
&\quad\ \{n_i \to n_j \mid i = 1, 4 \text{ and } j = 2, 3, 4, 5\} \cup \{n_i \to n_i \mid i = 0, 1, 2, 3, 4, 5, 6\} \\
\mathit{within} &\stackrel{\text{def}}{=} \{n_2 \to n_3, n_3 \to n_4\} \\
\mathcal{B} &\stackrel{\text{def}}{=} \{(\{n_2, n_3, n_4\}, t)\} \\
\mu(n_0) &\stackrel{\text{def}}{=} \neg(\mathsf{wants}(k) \vee \mathsf{awaits}(k) \vee \mathsf{critical}(k)) \\
\mu(n_1) &\stackrel{\text{def}}{=} \mathsf{wants}(k) \\
\mu(n_2) &\stackrel{\text{def}}{=} t \neq k \wedge \mathsf{isMin}(t) \wedge \mathsf{awaits}(k) \wedge \mathsf{awaits}(t) \\
\mu(n_3) &\stackrel{\text{def}}{=} t \neq k \wedge \mathsf{isMin}(t) \wedge \mathsf{awaits}(k) \wedge pc(t) = 5 \\
\mu(n_4) &\stackrel{\text{def}}{=} t \neq k \wedge \mathsf{isMin}(t) \wedge \mathsf{awaits}(k) \wedge pc(t) = 6
\end{aligned}
$$

$$
\begin{aligned}
\mu(n_5) &\stackrel{\text{def}}{=} \mathsf{isMin}(k) \wedge \mathsf{awaits}(k) \\
\mu(n_6) &\stackrel{\text{def}}{=} \mathsf{isMin}(k) \wedge \mathsf{critical}(k) \\
\eta(e) &\stackrel{\text{def}}{=} \begin{cases} (\tau_3, k) & \text{if } e \in \{n_1 \to n_i \mid i = 2,3,4,5\} \\ (\tau_4, t) & \text{if } e \in \{n_2 \to n_3\} \\ (\tau_4, k) & \text{if } e \in \{n_5 \to n_6\} \\ (\tau_5, t) & \text{if } e \in \{n_3 \to n_4\} \\ (\tau_6, t) & \text{if } e \in \{n_4 \to n_i \mid i = 2,3,4,5\} \end{cases} \\
\mathcal{F} &\stackrel{\text{def}}{=} \langle\langle \{n_4 \to n_i \mid i = 2,3,4,5\},\ \{n_6 \to n_0\},\ \lambda n \to \mathit{lower}(\mathit{bag}, \mathit{ticket}(k))\rangle\rangle \\
f(n) &\stackrel{\text{def}}{=} \begin{cases} \neg(\mathsf{wants}(k) \vee \mathsf{critical}(k)) & \text{if } n = n_0 \\ \mathsf{wants}(k) & \text{if } n = n_1 \\ \mathsf{critical}(k) & \text{if } n = n_6 \\ \mathit{true} & \text{otherwise} \end{cases}
\end{aligned}
$$

The diagram presented above consists of 7 nodes, named n_i for $i = 0, \ldots, 6$. The initial node is n_0. Each node in the diagram contains self-loop edges for all transitions which are not labeling any other outgoing edge from such node. For example, for node n_4 there exists an (implicit) edge $n_4 \to n_4$ for all transitions other than $\tau_6(t)$. Additionally, the value of the ranking function is the subset of tickets lower than the ticket of k. This set decreases (with respect to $\subseteq$) every time the leader thread (captured by the box variable t) exits the critical section and removes its pair from the set. ⌟

In general, PVDs require to work in collaboration with *Parametrized Invariance* presented in Chapter 3, as a PVD usually relies on some parametrized invariants that can be proven using the parametrized invariance technique. Section 10.5 of Chapter 10 presents more details about the use of PVDs in collaboration with *Parametrized Invariance*. Additionally, Section 10.5 describes in more detail Example 4.3 and presents the empirical results of using PVD to prove liveness properties for other programs.

4.4 Summary

In this chapter we introduced *Parametrized Verification Diagrams*, an extension of *Generalized Verification Diagrams* which allow to prove temporal properties of concurrent systems with an unbounded number of processes. We have shown that GVD are not suitable for parametrized systems and why PVDs present as a natural candidate to tackle parametrized verification.

PVDs enable to encode in a single proof an evidence that all instances of the parametrized system satisfy a given temporal specification. We have proven a soundness theorem which indicates that the fact that a PVD encodes the proof of satisfaction of a parametrized specification by a parametrized system can be automatically checked by:

- solving a finite-state model checking problem, and
- proving a finite number of verification conditions, generated automatically from the program and the diagram.

Decision procedures for the underlying theories of the data types in the program allow to handle these verification conditions automatically as well. Later, in Chapters 6 and 7 we will present decision procedures which can be used to automatically verify the verification conditions presented in Section 4.3 when the program manipulates concurrent lists or skiplists.

PVDS has been implemented as part of the theorem prover for parametrized systems LEAP, which is presented in Chapter 9.

5

Invariant Generation for Parametrized Systems

> *Science, my lad, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.*
>
> Jules Verne

In Chapter 3 and Chapter 4 we presented techniques for the parametrized verification of safety and liveness properties. Both techniques have in common that they rely on intermediate invariants in order to successfully accomplish the verification task. In general, these intermediate invariants need to be provided by the user. However, we are interested in automatic methods for generating such intermediate invariants whenever this is possible. This chapter study the problem of automatically inferring invariants for parametrized systems.

In this chapter we define an abstract-interpretation–based framework for inferring indexed invariants of parametrized programs. The main idea here is to build what we call a *reflective abstraction* of the parametrized program. A reflective abstraction consists of a thread composed with a *mirror* abstraction that summarizes the effect of the remaining threads on the global variables. Fig. 5.1 presents the idea of mirror abstractions for deriving 2-indexed invariants. In this case, a reflective abstraction to infer a 2-indexed invariant of a parametrized system is constructed by abstracting the system into 2 materialized threads and the mirror process.

Part of the core of the method we present in this chapter is based on the idea that invariants computed at various program locations of the materialized processes can be transfered into guards of the mirror process. This way, the abstraction of other interfering threads via the mirror process varies during the course of the analysis, similarly to the way in which materialization in shape analysis enables the heap abstraction to vary for better precision. We also present in this chapter different iteration schemes.

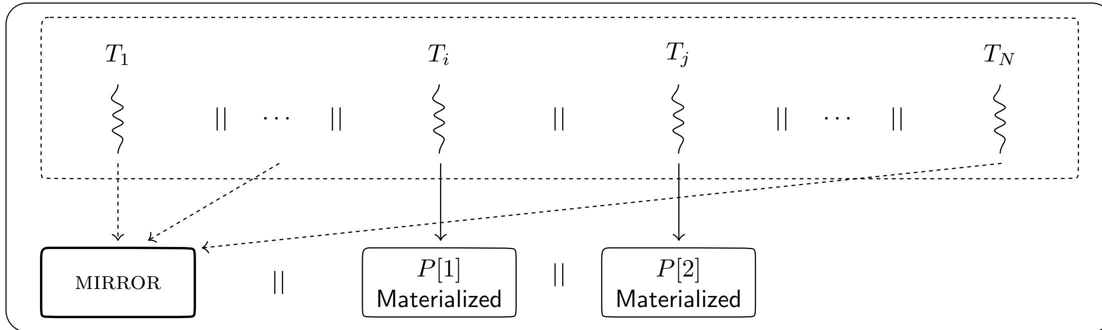


Figure 5.1: A reflective abstraction to infer 2-indexed invariants of a parametrized system.

This chapter is structured as follows. Section 5.1 introduces the basic idea of reflective abstractions for parametrized systems. Section 5.2 presents the notion of reflective abstraction of a parametrized system. Section 5.3 describes an iterative procedure to generate invariants of a parametrized system using abstract interpretation on reflective abstractions. Finally, Section 5.4 presents a summary.

5.1 Self-Reflection

In this section, we present the basic idea behind reflective abstractions of parametrized systems. As we mentioned before, parametrized programs consist of a fixed but unbounded number of thread instances $T_1, \ldots, T_M$ where $M \geq 1$. We begin with a motivating example.

Example 5.1 (Work Stealing)
Fig. 5.2 presents program WORKSTEAL. This is a parametrized program which processes a

```
   global
      Int len > 0
      Array[len] data
      Int next = 0
   procedure WORKSTEAL()
      Int c = 0
      Int end = 0
   begin
      if next + 10 ≤ len then
         ⟨ c := next
1:         end := next + 10
           next := next + 10 ⟩
      end if
2: while c < end do
3:    data[c] := process(data[c])
4:    c := c + 1
5: end while
   end procedure
```

Figure 5.2: WORKSTEAL: A parametrized array processing program.

collection of elements stored in an array. In the program, each thread processes a section of 10 consecutive elements within the array.

In this program, there is a global array $data$ of size len which contains the elements to be processed. Additionally, there is a global variable $next$ which stores the current unprocessed index. Each thread has two local variables. For a given thread, local variable c indicates the current position in the array being processed by the thread. On the other hand, local variable end holds the last position in the array that needs to process the current thread. ⌟

The goal is to prove properties about the behavior of parametrized systems that must hold regardless of the number of running thread instances M. The simplest properties involve only global variables. This kind of properties are said to be 0-index properties. Other properties may also involve local variables, as well as globals. Properties involving global variables and local variables of a single thread are said to be 1-indexed properties. Similarly, properties which refer to global variables and local variables of two different threads are called 2-index properties and so on.

Example 5.2
Consider again the WORKSTEAL program presented in Example 5.1. An example of a 0-index property would be:

$$\textsf{modten} \;=\; next \mod 10 = 0$$

Property modten says that global variable $next$ is always multiple of 10. Similarly, for line 3, a 1-indexed property that should hold is bounded, which states that access to array $data$ are always within bounds:

$$\textsf{bounded} \;=\; (\forall\, i)\; 0 \leq c[i] < len \tag{5.1}$$

Finally, an example of a 2-indexed property is race-freedom for two *distinct* threads i, j whenever one of the threads is at line 3. This is described by property racefree which says that no two different threads will process the same element in the $data$ array:

$$\textsf{racefree} \;=\; (\forall\, i, j)\;\; (i \neq j \rightarrow c[i] \neq c[j]) \tag{5.2}$$

⌟

In order to simplify the reasoning, in the rest of this chapter we will implicitly assume that two different symbols i_j, i_k involved in a given assertion ψ are used to refer to different threads.

The core of the method presented in this chapter involves the use and adaptation of existing invariant synthesis techniques to parametrized programs. The technique we present here allows us to generate invariants such as modten, bounded, and racefree presented in Example 5.2. Our approach, inspired by the idea of materialization in shape analysis [182], is based on identifying a fixed number of *materialized threads* and abstracting the remaining processes into a single, separate thread that we will call the MIRROR.

Fig. 5.1 provides a graphical idea of this approach. In this figure, a parametrized system composed by M different threads is modeled by three threads:

- 2 materialized threads, $P[1], P[2]$; and
- the mirror process MIRROR that summarizes the effects of the $M-2$ remaining threads on the shared global variables.

The number of materialized threads is fixed *a priori* and depends on the index of the desired invariant. For example, we need to materialize at least 2 threads to infer a 2-indexed invariant. The novel aspect of our *reflective* approach is that the MIRROR process is *not* fixed *a priori* but rather is derived as part of the fixed point analysis.

The composition of the materialized threads and the MIRROR results on a regular sequential transition system that can be analyzed using a standard abstract interpretation engine for sequential systems. The MIRROR process simulates the effect of the remaining non-materialized threads over the shared global variable in the system. In the case of WORKSTEAL, that is the effect of the remaining threads over the global variables $next$ and len. The MIRROR thread itself has no local variables as they are abstracted away.

In order to construct the MIRROR thread, we need to focus on the transitions that produce an effect on the global variables. These transitions are modeled as a self-loop around a single location in the MIRROR thread, and the local variable updates including program locations are quantified away. However, in order to maintain precision, it is preferable to restrict the scope of the transition within the MIRROR thread only to those states of the program that are actually reachable at the program location corresponding to the transition.

Example 5.3
Fig. 5.3 shows the basic setup for invariant synthesis for the WORKSTEAL program. In the case of WORKSTEAL, the program contains a single transition, the one going from line 1 to line 2 that actually produces an effect over the shared global variables. This transition is highlighted in Fig. 5.3. It updates the value of global variable $next$. On the other hand, global variable len is never updated by the program. This transition is then copied as a self-loop around in the MIRROR thread. As the transition modifying the global variables is the one that corresponds to program line 1, we restrict the scope of the transition in the MIRROR thread to those program states that are reachable at line 1. We will over-approximate these states by an assertion $\mathsf{Inv}@\ell_0$. In fact, the guard $\mathsf{Inv}@\ell_1$ in the MIRROR thread corresponds to the invariant computed at program location ℓ_1 in a materialized thread. ⌟

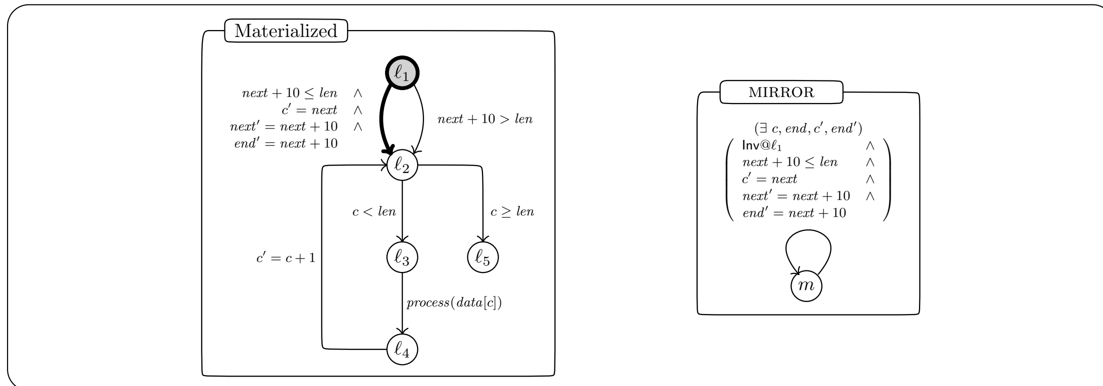


Figure 5.3: Materialized and MIRROR thread for a 1-index invariant in the WORKSTEAL program.

The main question now is to precisely determine what $\mathsf{Inv}@\ell_i$ is on a MIRROR thread for all transitions at program location ℓ_i that produce an effect on the global variables. A simple solution would be to assume $\mathsf{Inv}@\ell_i$ to be $true$. In fact, doing so leads to a valid over-approximation of all states that are reachable whenever some process resides at location ℓ_i However, $true$ can be often a very coarse over-approximation. Our key observation is that $\mathsf{Inv}@\ell_i$, and hence the corresponding MIRROR thread, does not need to be fixed *a priori*. Instead, we build a more precise abstraction by incrementally constructing MIRROR as follows:

1. The first iteration sets $\mathsf{Inv}@\ell_i$ to $false$. By doing this, we are in fact disabling the MIRROR thread. Consequently, this first iteration approximates only those states reachable by the materialized threads running in isolation.

2. We run an abstract interpreter and compute invariants of the composition of the current MIRROR thread and the materialized threads.

3. The MIRROR process for the next iteration is updated with $\mathsf{Inv}@\ell_i$ set to the candidate invariants computed in the previous iteration at location ℓ_i in the materialized threads with the local variables projected out. This *candidate invariant reflection* allows the MIRROR to run from a larger portion of the reachable state space.

4. Convergence is achieved whenever the invariants obtained at some iteration are subsumed by those at the previous iteration. At this point, the effect of the mirror and the materialized processes in the invariants and the guards is stable.

Upon convergence, the result are k-indexed invariants that relate the local variables of the k materialized threads to the global variables.

5.2 Reflective Abstractions and Inductive Invariants

We now define the notion of a *reflective abstraction* of a parametrized system. Then we present the main result of this section, Theorem 5.1, which proves the soundness of reflective abstractions.

Recall the definition of non-parametrized fair transition system we gave in Section 2.4.2. To simplify the presentation, in this chapter we assume that all program variables are of type integer. Additionally, through this chapter, we will represent a transition as a tuple $\tau : \langle \ell_{\mathrm{src}}, \ell_{\mathrm{tgt}}, \rho \rangle$ consisting of a pre-location ℓ_{src}, a post-location ℓ_{tgt} and a transition relation ρ that relates the values of the variables (global and local) before the transition with the values after the transition is taken.

Example 5.4 (Parametrized Transition System for WORKSTEAL)
Consider once again the WORKSTEAL program presented in Example 5.1. The parametrized transition system associated to this program consists of:

- Global variables $V_{global} = \{len, next\}$.
- Local variables $V_{local} = \{c, end\}$.
- Program locations $Locs = \{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5\}$, with the initial location being ℓ_1.

- Transition relations are depicted as labels in Fig. 5.3, ignoring the transition relations for the MIRROR process.

Not modeling the array $data$ means the transition relation ρ between ℓ_3 and ℓ_4 is a no-op, that is, $\rho = pres(V_{global} \cup V_{local})$. ⌟

Given a formula φ, where helpful for clarity in presentation, we write $\varphi[G, X]$ to indicate the variables $G \subseteq V_{global}$ and $X \subseteq V_{local}$ over which the formula φ is defined.

Definition 5.1 (Post-Condition).
Let φ be an assertion over the set of variables X and τ be a transition with transition relation ρ_τ. The (strongest) post-condition (also known as image, transfer function, or transformer) of φ across τ is given by the assertion

$$\text{post}(\varphi, \tau):\ (\exists\, X_0)\ (\ \varphi[X_0]\ \wedge\ \rho_\tau[X_0, X]\ )$$

The post-condition describes all the valuations of variables X reachable in one step by executing transition τ starting from a state that satisfies φ. ⌟

Let $\Gamma(X)$ be some fixed first-order language of assertions, such as the theory of integers, involving free variables X. We overload $\models$ to denote the semantic entailment relation between these formulas. An *assertion map* $\eta : Locs \rightharpoonup_{\text{fin}} \Gamma(X)$ maps each location to an assertion in $\Gamma(X)$ and it is inductive whenever the assertion at the initial location subsumes the initial condition and the assertion map respects consecution. Formally speaking, an inductive assertion map is described by the following definition.

Definition 5.2 (Inductive Assertion Map).
An assertion map $\eta : Loc \rightharpoonup_{\text{fin}} \Gamma[X]$ maps each location ℓ to an assertion. An assertion map η is inductive *whenever the following conditions hold:*

– **Initiation:** *The assertion at ℓ_0 subsumes the initial condition. That is:*

$$\Theta\ \models\ \eta(\ell_0)$$

– **Consecution:** *For any transition τ between ℓ_{src} and ℓ_{tgt}, the post-condition transformer for τ applied to $\eta(\ell_{\mathrm{src}})$ implies $\eta(\ell_{\mathrm{tgt}})$. Formally, for each transition $\tau\colon \langle \ell_{\mathrm{src}}, \ell_{\mathrm{tgt}}, \rho\rangle$:*

$$\text{post}(\eta(\ell_{\mathrm{src}}), \tau)\ \models\ \eta(\ell_{\mathrm{tgt}})$$ ⌟

5.2.1 Reflective Abstractions

In essence, a reflective abstraction is an over-approximation of a parametrized transition system by a sequential one. What makes an abstraction *reflective* is that the over-approximation is computed under an assertion map. As usual, to specifically denote a sequential transition system, we use $\mathcal{S}$.

In this chapter we will use a slightly different definition of state from the one presented in Chapter 2. Here, we consider a state ξ to be a tuple $\langle L, \mathbb{V} \rangle$ consisting of a map $L : [M] \rightharpoonup_{\text{fin}} \mathit{Locs}$ that associates a location $L(i)$ for each thread instance i in M and a valuation map $\mathbb{V}$ which assigns to each global variable and each local variable instance to its integer value. We write $\mathbb{V}(V_{global})$ to denote the valuations to all global variables and $\mathbb{V}(V_{local}[i])$ to denote the valuations of local thread i. We write $\mathbb{V} \models \varphi$ for a valuation $\mathbb{V}$ satisfying a formula φ. Given a state ξ, we write $\xi \models \varphi$ when state ξ satisfies formula φ.

In Chapter 2 we introduced the idea of i-indexed invariants. We now present a formal definition of 1-index invariants that adapts to the invariant generation framework we are developing in this chapter.

Definition 5.3 (1-Indexed Invariant).
A pair $\langle \ell, \varphi \rangle$ consisting of a location ℓ and an assertion $\varphi[V_{param}]$ is a 1-index invariant of a parametrized program P if and only if for every reachable state $\xi \colon (L, \mathbb{V})$ with $M > 0$ thread instances, and for every $i \in [M]$

$$\textit{if } L(i) = \ell \textit{ then } (\mathbb{V}(V_{global}), \mathbb{V}(V_{local}[i])) \models \varphi$$

In other words, the valuations of the local variables $V_{local}[i]$ and global variables V_{global} for any thread instance i reaching the location ℓ satisfies φ. ┘

We can now generalize Definition 5.3 to k-indexed invariants.

Definition 5.4 (k-Indexed Invariant).
A tuple $\langle \ell_1, \ell_2, \ldots, \ell_k, \varphi \rangle$ consisting of locations $\ell_1, \ldots, \ell_k$ and assertion $\varphi[V_{global}, X_1, \ldots, X_k]$, where $X_1, \ldots, X_k$ are k disjoint copies of the local variables in V_{local}, is a k-indexed invariant of a parametrized transition system $\mathcal{P}$ whenever for every reachable state $\xi \colon (L, \mathbb{V})$ with $M \geq k$ thread instances, and for every $i_1, \ldots, i_k \in [M]$ where $i_a \neq i_b$ if $a \neq b$:

$$\begin{array}{ll} \textit{if} & L(i_1) = \ell_1, L(i_2) = \ell_2, \cdots, L(i_k) = \ell_k \\ \textit{then} & (\mathbb{V}(V_{global}), \mathbb{V}(X[i_1]), \cdots, \mathbb{V}(X[i_k])) \models \varphi(V_{global}, X_1, \ldots, X_k) \end{array}$$ ┘

Inductive invariants are fundamental to the process of verifying safety properties of programs. In order to prove an assertion φ over all reachable states at a location ℓ, we seek an inductive assertion map η over the entire program such that $\eta(\ell) \models \varphi$.

We now formally define the notion of a reflective abstraction, which abstracts a parametrized system by:

- a system with $k > 0$ materialized threads; and

- a MIRROR thread that models the interference of the remaining threads on the shared global variables V_{global}.

The key result is that an invariant of a reflective abstraction is that of a parametrized system. Despite reflective abstraction can be used to infer k-indexed invariants, with $k > 1$, in the rest of the chapter we will describe reflective abstractions with a single thread in order to simplify the presentation.

Now, let η be an assertion map over the locations of a parametrized system $\mathcal{P}$. Our goal is to define a sequential system $\text{REFLECT}_{\mathcal{P}}(\eta)$. This sequential system contains transitions to model one specific thread instance, named the materialized thread, and the MIRROR thread, which models the influence of the other threads on the global variables.

Definition 5.5 (Reflective Abstraction).
The reflective abstraction of a parametrized system $\mathcal{P} : \langle \Sigma_{param}, V_{param}, \Theta_{param}, \mathcal{T}_{param} \rangle$ *associated to program* P *with respect to an assertion map* η *is a sequential transition system, written* $\text{REFLECT}_{\mathcal{P}}(\eta)$*, over variables* V_{param}*, with locations given by Locs and transitions given by:*

$$\mathcal{T}_{param} \cup \{ \text{MIRROR}(\tau, \eta, \ell) \mid \tau \in \mathcal{T}_{param} \text{ and } \ell \in Locs \}$$

Here, the original transitions T_{param} *model the materialized thread, while the* MIRROR *transitions model the visible effects of the remaining threads.*

For a transition $\tau : \langle \ell_{\mathrm{src}}, \ell_{\mathrm{tgt}}, \rho \rangle$ *and some location* $\ell \in Locs$*, the corresponding* MIRROR *transition* $\text{MIRROR}(\tau, \eta, \ell)$ *is defined as follows:*

$$\left\langle \ell,\ \ell,\ pres(V_{local}) \ \wedge\ (\exists\, Y, Y') \left(\eta(\ell_{\mathrm{src}})[V_{global}, Y] \ \wedge\ \rho[V_{global}, Y, V'_{global}, Y'] \right) \right\rangle$$

The initial location of the reflective abstraction is ℓ_0 *and the initial condition is* Θ_{param}*, as it comes directly from the parametrized system.* ⌟

Note that each MIRROR transition is a self-loop at location ℓ of the materialized thread. Equivalently, MIRROR can be seen as a process with a single location and self-looping transitions that is composed concurrently with the materialized thread. Note that each MIRROR transition preserves the local variables of the materialized thread, as they describe only the effect of each transition over the global shared variables.

Also, observe that the guard $\eta(\ell_{\mathrm{src}})[V_{global}, Y]$ of the MIRROR transition includes the invariant $\eta(\ell_{\mathrm{src}})$ of the interfering thread at the pre-location, which can be seen as reflecting the invariant of the materialized thread at ℓ_{src} on to the interfering thread.

Finally, the local variables are projected away from the transition relation using existential quantification to model the effect of the materialized transition on the shared variables. In Definition 5.5 this is done by existentially quantifying Y and Y'.

Example 5.5 (Reflective Abstraction)
We now present as an example part of an assertion map η for program WORKSTEAL introduced in Example 5.1. In this example, we use $\rho(\tau)$ to describe the transition relation of transition τ

in the original parametrized system $\mathcal{P}$ and we use $\rho(m)$ for the transition relation of a MIRROR transition in the reflective abstraction. Note that the assertion map η is not necessarily inductive.

Let us consider first program line 1 for which the assertion map is $\eta(\ell_1) : next = 0 \wedge c = 0 \wedge end = 10$. Now, program line 1 is associated to two possible transition relations. One when the guard of the conditional holds and another one in the case it does not hold. In the first case we have that the transition relation associated to program line 1 is:

$$\rho(\tau_1 : \langle \ell_1, \ell_2, \rho_1 \rangle) \quad : \quad next + 10 \leq len \wedge \begin{pmatrix} c' = next & \wedge \\ next' = next + 10 & \wedge \\ end' = next + 10 & \end{pmatrix} \wedge pres(\{len\})$$

Then, using this assertion map and the transition relations stated above, we have that the transition relation of the MIRROR transition m_1 is derived from the original transition τ_1 by computing the MIRROR self-loop:

$$\rho(m_1 : \text{MIRROR}(\tau_1, \eta, _)) \colon pres(\{c, end\}) \wedge (\exists c, end, c', end')\; \eta(\ell_1) \wedge \rho_1$$

Which, after replacing $\eta(\ell_1)$ and ρ_1, we obtain:

$$pres(\{c, end\}) \ \wedge (\exists\ c, end, c', end') \begin{bmatrix} next = 0 \ \wedge \ c = 0 \ \wedge \ end = 10 \wedge \\ next + 10 \leq len \ \wedge \ c' = next \wedge \\ next' = end' = next + 10 \ \wedge \ pres(\{len\}) \end{bmatrix} \tag{5.3}$$

Since transition $\rho(m_1 : \text{MIRROR}(\tau_1, \eta, _))$ expresses solely the effect of the transition over the global variables, we have that (5.3) is finally reduced to:

$$\rho(\tau_0 : \langle \ell_1, \ell_2, \rho_1 \rangle) \colon next + 10 \leq len \ \wedge \ c' = next \ \wedge \ next' = end' = next + 10 \ \wedge \ pres(\{len\})$$

Now, let's consider the second transition relation associated to program line 1. That is:

$$\rho(\tau_1' : \langle \ell_1, \ell_2, \rho_1' \rangle) \quad : \quad next + 10 > len \wedge pres(\{next, len, c, end\})$$

The MIRROR transition relation associated to this program transition is:

$$\rho(m_0' : \text{MIRROR}(\tau_0', \eta, _)) : 10 > len \ \wedge \ pres(\{next, len, c, end\})$$

As another example, consider the assertion map η associated to program location 3:

$$\eta(\ell_3) next \geq 0 \ \wedge \ c \geq 0 \ \wedge \ c < end$$

In this case, as the transition relation associated to program location 4 is:

$$\rho(\tau_4 : \langle \ell_4, \ell_5, \rho_4 \rangle) \quad : \quad c' = c + 1 \ \wedge \ pres(\{next, len, end\})$$

We have that the transition relation associated to the MIRROR process is:

$$\rho(m_4 : \text{MIRROR}(\tau_4, \eta, _)) : next \geq 0 \ \wedge \ pres(\{next, len, c, end\})$$

Note that the only MIRROR transition which modifies global variables is the one associated to m_1 (that is the transition highlighted in Fig. 5.3). All other MIRROR transitions preserve the value of the global variables $next$ and len and thus they can be omitted. ⌟

We now present the main result involving reflective abstractions which establishes that if η is an inductive invariant of the reflective abstraction $\text{Reflect}_{\mathcal{P}}(\eta)$ for a parametrized program P, then for every location ℓ, the assertion $\eta(\ell)$ is a 1-indexed invariant according to Definition 5.3.

Theorem 5.1 (Reflection Soundness):
Let η be an assertion map such that η is inductive for the system $\text{Reflect}_{\mathcal{P}}(\eta)$. It follows that for each location ℓ of $\mathcal{P}$, $\eta(\ell)$ is a 1-index invariant of $\mathcal{P}$. ⌟

Proof. The proof proceeds by induction, considering all the states reachable in m steps.

- **Base case:** For the base case, we consider all states reachable in 0 steps. Let $M > 0$ represent the number of thread instances. Each initial state is of the form $(L_0, \mathbb{V}_0)$ where $L_0(i) = \ell_0$ and $\mathbb{V}_0(V_{global}), \mathbb{V}_0(V_{local}[i]) \vDash \Theta(V_{global}, V_{local})$. Since, $\Theta \vDash \eta(\ell_0)$, we have $L_0(i) = \ell_0$ and $\mathbb{V}_0(V_{global}), \mathbb{V}_0(V_{local}[i]) \vDash \eta(\ell_0)$. This implies the base case.

- **Inductive case:** Let us assume that for every state $\xi_m : (L_m, \mathbb{V}_m)$ reachable in some $m \geq 0$ steps by $M > 0$ thread instances:

  $$\mathbb{V}_m(V_{global}), \mathbb{V}_m(V_{local}[i]) \ \vDash \ \eta(L_m(i)), \quad \text{for all } i \in [M]$$

  Consider an arbitrary state $\xi_{m+1} : (L_{m+1}, \mathbb{V}_{m+1})$ reachable in one step from ξ_m by the execution of some transition $\tau[i_m]$ for $i_m \in [M]$. That is, $\xi_m \xrightarrow{\tau[i_m]} \xi_{m+1}$. We wish to prove that

  $$\mathbb{V}_{m+1}(V_{global}), \mathbb{V}_{m+1}(V_{local}[j]) \ \vDash \ \eta(L_{m+1}(j)), \quad \text{for all } j \in [M]$$

  We need now to consider two possible cases.

  1. Case $j = i_m$. By our induction hypothesis we have $\mathbb{V}_m(V_{global}), \mathbb{V}_m(V_{local}[i_m]) \vDash \eta(L_m(i_m))$. Moreover:

     $$\mathbb{V}_m(V_{global}), \mathbb{V}_m(V_{local}[i_m]), \mathbb{V}_{m+1}(V_{global}), \mathbb{V}_{m+1}(V_{local}[i_m]) \ \vDash \ \rho_\tau$$

     Therefore, $\mathbb{V}_{m+1}(V_{global}), \mathbb{V}_{m+1}(V_{local}[i_m]) \vDash \text{post}(\eta(L_m(i_m)), \tau)$. Applying consecution of η for τ, we get:

     $$\mathbb{V}_{m+1}(V_{global}), \mathbb{V}_{m+1}(V_{local}[i_m]) \ \vDash \ \text{post}(\eta(L_m(i_m)), \tau) \ \vDash \ \eta(L_{m+1}(i_m))$$

2. Case $j \neq i_m$. Let $j \neq i_m$ refer to some thread instance whose local state remains unchanged due to the execution of $\tau[i_m]$:

$$\mathbb{V}_m(V_{local}[j]) = \mathbb{V}_{m+1}(V_{local}[j])$$

The execution of $\tau[i_m]$ can be treated as a transition of the MIRROR thread. In particular, the local variables of the thread instance i_m simulated by the MIRROR thread satisfy $\eta(L_m(i_m))$ by the induction hypothesis. Therefore, we have:

$$(\mathbb{V}_m(V_{global}), \mathbb{V}_{m+1}(V_{global})) \models (\exists\, X, X')\ \eta(L_m(i_m)) \ \wedge\ \rho_\tau(V_{global}, X, V'_{global}, X')$$

Therefore,

$$\begin{array}{l}\mathbb{V}_m(V_{global}), \mathbb{V}_m(V_{local}[j]), \mathbb{V}_{m+1}(V_{global}), \mathbb{V}_{m+1}(V_{local}[j]) \models \\ \qquad \underbrace{pres(V_{local}) \ \wedge\ (\exists\, Y, Y')\ \eta(L_m(i_m)) \ \wedge\ \rho_\tau(V_{global}, Y, V'_{global}, Y')}_{\text{MIRROR}(\tau, \eta, L_m(j))}\end{array} \tag{5.4}$$

We have $\mathbb{V}_m(V_{global}), \mathbb{V}_m(V_{local}[j]) \models \eta(L_m(j))$ by the inductive hypothesis. Since η is an inductive assertion for the reflective abstraction $\text{REFLECT}_{\mathcal{P}}(\eta)$, it satisfies consecution with respect to the transition $\text{MIRROR}(\tau, \eta, L_m(j))$:

$$\text{post}(\eta(L_m(j)), \text{MIRROR}(\tau, \eta, L_m(j))) \ \models\ \eta(L_m(j)) \tag{5.5}$$

Combining (5.4) and (5.5) we obtain:

$$\begin{array}{rl}\mathbb{V}_{m+1}(V_{global}), \mathbb{V}_{m+1}(V_{local}[j]) & \models \text{post}(\eta(L_m(j)), \text{MIRROR}(\tau, \eta, L_m(j))) \\ & \models \eta(L_m(j)) \ \ (= \eta(L_{m+1}(j)))\end{array}$$

This completes our induction proof. □

To generalize reflective abstraction to $k > 1$ materialized threads, we first construct a transition system that is the product of k-copies of the parametrized program P. This transition system uses k-copies of the locals and a single instance of the globals from P. Then, given an assertion map η, we add MIRROR transitions to construct the reflective abstraction following Definition 5.5 on this product system. Finally, each transition τ is projected onto the global shared variables guarded by the assertion given by η in the transition's pre-location. This way, an inductive assertion derived on the reflective abstraction of the product system is a k-indexed invariant for the original parametrized system.

5.3 Reflective Abstract Interpretation

In this section, we present an iterative procedure to generate invariants of a parametrized system by applying abstract interpretation on reflective abstractions. The basic idea is to iteratively alternate between constructing a reflective abstraction $\text{REFLECT}_{\mathcal{P}}(\eta_i)$ using a candidate invariant map η_i and generating the next candidate invariant map η_{i+1} using abstract interpretation on $\text{REFLECT}_{\mathcal{P}}(\eta_i)$. We also consider various approaches to reflective abstract interpretation and we

compare reflective abstraction with *interference abstraction*, a commonly-used approach when analyzing multi-thread programs [155].

First, we briefly need to recall the theory of abstract interpretation [29, 57, 58] for finding inductive assertion maps as the fixed point of a monotone operator over an abstract domain.

Abstract interpretation is based on the observation that invariants of a program are over-approximations of the concrete collecting semantics η^*, an assertion map that associates each location ℓ with a first-order assertion $\eta^*(\ell)$ characterizing all reachable states at the location ℓ.

Formally, we write:

$$\eta^* = \mathsf{lfp}\ \mathcal{F}_{\mathcal{S}}(\mathit{false})$$

In the definition above, $\mathcal{F}_{\mathcal{S}}(\eta)$ is a "single-step" semantics. That is, a monotone operator over the lattice of assertion maps that collects all the states reachable in at most one step of the system $\mathcal{S}$, and false maps every location to false. Here, we will make the transition system $\mathcal{S}$ an explicit argument of $\mathcal{F}$, rather than fixed:

$$\eta^* = \mathsf{lfp}\ \mathcal{F}(\mathit{false}, \mathcal{S}) \tag{5.6}$$

We can also define a structural pre-order on sequential transition systems. We say a sequential system $\mathcal{S}$ structurally refines other sequential system $\mathcal{S}'$, written $\mathcal{S} \preceq \mathcal{S}'$, as simply saying that $\mathcal{S}$ and $\mathcal{S}'$ have the same structure—in terms of their variables, locations, and transitions—and where the initial conditions and the corresponding transition relations are ordered by $\models$. Formally speaking, structural pre-order between sequential transition systems is described by the following definition.

Definition 5.6 (Structural Pre-Order).
A sequential transition system

$$\mathcal{S} \colon \langle V, \Theta, \mathcal{T} \rangle \quad \textit{structurally refines} \quad \mathcal{S}' \colon \langle V', \Theta', \mathcal{T}' \rangle$$

written $\mathcal{S} \preceq \mathcal{S}'$, if and only if the following conditions hold:

1. *the program variables of $\mathcal{S}$ and $\mathcal{S}'$ are the same. That is, $V = V'$;*
2. *the initial condition of $\mathcal{S}'$ over-approximates the initial condition of $\mathcal{S}$. That is, $\Theta \models \Theta'$; and*
3. *for any transition $\tau' \colon \langle \ell'_{\mathrm{src}}, \ell'_{\mathrm{tgt}}, \rho' \rangle \in \mathcal{T}'$, there is a transition*

$$\tau \colon \langle \ell_{\mathrm{src}}, \ell_{\mathrm{tgt}}, \rho \rangle \in \mathcal{T}$$

such that the pre-locations and post-locations are the same and the transition relation of τ' over-approximates the transition relation of τ. That is, $\ell_{\mathrm{src}} = \ell'_{\mathrm{src}}$, $\ell_{\mathrm{tgt}} = \ell'_{\mathrm{tgt}}$, and $\rho \models \rho'$. ⌟

It is clear that if $\mathcal{S} \preceq \mathcal{S}'$, then the behaviors of $\mathcal{S}'$ over-approximate the behaviors of $\mathcal{S}$. Now, we can see that the concrete collecting semantics functional $\mathcal{F}(\eta, \mathcal{S})$ is monotone over both arguments:

1. over concrete assertion maps ordered by $\models$ location-wise; and
2. over sequential transition systems using the structural pre-order.

The abstract interpretation framework allows to approximate the *collecting semantics* of programs in an abstract domain $\mathcal{A}\colon \langle A, \sqsubseteq, \bot, \top, \sqcup, \sqcap\rangle$ defined by a lattice. The abstract lattice is related to the concrete lattice of first-order assertions $\Gamma[X]$ through a *Galois connection* described by:

- an abstraction function $\alpha : \Gamma[X] \to A$ that maps assertions in the concrete domain to abstract objects; and
- $\gamma : A \to \Gamma[X]$ that interprets abstract objects as concrete assertions representing sets of states.

In the abstract interpretation framework, we lift the operator $\mathcal{F}$ defined over the concrete domain to the corresponding monotone operator $\widehat{\mathcal{F}}$ over the abstract domain $\mathcal{A}$. Analogously, we write $\widehat{\eta} : Locs \rightharpoonup_{\mathsf{fin}} A$ for an abstract assertion map. A fixed point computation in Equation (5.6) is then expressed in terms of the abstract domain $\mathcal{A}$ as follows:

$$\widehat{\eta}^* = \mathsf{lfp}\ \widehat{\mathcal{F}}(\bot, \mathcal{S})$$

Here, $\bot$ is the abstract assertion map that maps every location to the bottom element $\bot$ of the abstract domain. If $\mathcal{A}$ is an abstract domain, then it follows that $\gamma \circ \widehat{\eta}^*$ yields an inductive assertion map over the concrete domain.

If the domain $\mathcal{A}$ is finite or has the ascending chain condition, the least-fixed point lfp operator may be computed iteratively. On the other hand, many domains of interest fail to satisfy these conditions. Herein, abstract interpretation provides us a framework using the *widening* operator that can be repeatedly applied to guarantee convergence to a *post-fixed point* that over-approximates the least-fixed point. Concretizing this post-fixed point leads to a valid but weaker inductive assertion map.

5.3.1 Abstract Interpretation using Reflection

The overall idea behind our invariant generation technique is to alternate between constructing a sequential reflective abstraction of the given parametrized system $\mathcal{P}$ and applying abstract interpretation for sequential systems on the computed reflective abstraction. We distinguish two abstract interpretation schemes: *lazy* and *eager*.

Lazy Reflective Abstract Interpretation: Lazy reflective abstract interpretation for a parametrized system $\mathcal{P}$ proceeds as follows:

1. First, begin with an initial abstract candidate invariant map $\widehat{\eta}_0$ that maps each location to the least abstract element $\bot$.
2. Then, iterate the following steps until convergence:
   (a) compute the reflective abstraction $\mathcal{S}_j$ using $\widehat{\eta}_j$;
   (b) on the reflective abstraction $\mathcal{S}_j$, apply an abstract interpreter for sequential systems to obtain the next candidate invariant map $\widehat{\eta}_{j+1}$;

(c) terminate the iteration whenever $\widehat{\eta}_{j+1}(\ell) \sqsubseteq \widehat{\eta}_j(\ell)$ for all $\ell \in Locs$.

We now formally derive the lazy abstract interpretation scheme above. Let $\widehat{\mathcal{G}}_{\text{LAZY},\mathcal{P}}$ be the following operator defined over the abstract lattice:

$$\widehat{\mathcal{G}}_{\text{LAZY},\mathcal{P}}(\widehat{\eta}) \stackrel{\text{def}}{=} \mathsf{lfp}\ \widehat{\mathcal{F}}(\bot, \textsc{Reflect}_{\mathcal{P}}(\gamma \circ \widehat{\eta})) \quad (5.7)$$

Given a map $\widehat{\eta}$ associating locations with abstract objects, the operator $\widehat{\mathcal{G}}_{\text{LAZY},\mathcal{P}}$ is implemented by:

1. concretizing $\widehat{\eta}$ to compute $\textsc{Reflect}_{\mathcal{P}}(\gamma \circ \widehat{\eta})$, the reflective abstraction; and
2. applying the least fixed point of $\widehat{\mathcal{F}}$ over the reflection.

We note that the monotonicity of $\widehat{\mathcal{G}}_{\text{LAZY}}$ holds where lfp is computable. In particular, note that $\textsc{Reflect}_{\mathcal{P}}(\eta)$ is a monotone operator. In conclusion, the overall scheme for inferring invariants of the original system $\mathcal{P}$ consists of computing the following:

$$\widehat{\eta}^* = \mathsf{lfp}\ \widehat{\mathcal{G}}_{\text{LAZY},\mathcal{P}}(\bot) \qquad \text{and let} \qquad \eta_{\text{inv}} \stackrel{\text{def}}{=} \gamma \circ \widehat{\eta}^* \quad (5.8)$$

Then, for each location ℓ, $\eta_{\text{inv}}(\ell)$ is a 1-index invariant of the system $\mathcal{P}$.

Soundness follows from the soundness of abstract interpretation and reflection soundness proved in Theorem 5.1. In practice, we implement the operator $\widehat{\mathcal{G}}_{\text{LAZY}}$ by constructing a reflective abstraction and calling an abstract interpreter as a black-box. Note that if the abstract interpreter uses widening to enforce convergence, $\widehat{\mathcal{G}}_{\text{LAZY}}$ is not necessarily monotone since the post-fixed point computation cannot be guaranteed to be monotone. We revisit these considerations in Section 5.3.3.

Eager Reflective Abstract Interpretation: In contrast with the lazy scheme, it is possible to construct an eager scheme that weaves the computation of a least-fixed point and the reflective abstractions in a single iteration. This scheme can be thought of as abstract interpretation on the Cartesian product of the abstract domain $\mathcal{A}$ and the space of reflective abstractions $\textsc{Reflect}_{\mathcal{P}}(\gamma \circ \widehat{\eta})$ for $\widehat{\eta} \in (Locs \rightharpoonup_{\text{fin}} A)$ ordered by the structural pre-order relation $\preceq$.

The eager scheme consists of using an eager operator and an eager reflective abstract interpretation as a least-fixed point computation with that operation starting at $\bot$:

$$\widehat{\mathcal{G}}_{\text{EAGER},\mathcal{P}}(\widehat{\eta}) \stackrel{\text{def}}{=} \widehat{\mathcal{F}}(\widehat{\eta}, \textsc{Reflect}_{\mathcal{P}}(\gamma \circ \widehat{\eta})) \qquad \text{and} \qquad \widehat{\eta}^* = \mathsf{lfp}\ \widehat{\mathcal{G}}_{\text{EAGER},\mathcal{P}}(\bot) \quad (5.9)$$

In other words, we apply a single step of the abstract operator $\widehat{\mathcal{F}}$ starting from the map $\widehat{\eta}$ over the reflective abstraction from $\gamma \circ \widehat{\eta}$.

5.3.2 Interference Abstraction versus Reflective Abstraction

We now compare the eager and lazy reflective abstraction approaches with the commonly used interference abstraction. The goal of interference abstraction [155] is to capture the effect of interfering transitions flow-insensitively much like a reflective abstraction. The *interference*

semantics can be expressed concisely in the formalism developed in this section by the following operator:

$$\widehat{\eta}^* = \mathsf{lfp}\ \widehat{\mathcal{F}}(\bot, \mathcal{S}_\top) \quad \text{where} \quad \mathcal{S}_\top \stackrel{\text{def}}{=} \textsc{Reflect}_{\mathcal{P}}(\mathit{true}) \tag{5.10}$$

Here $\top$ represents the abstract assertion map that associates each location with $\top \in A$. In particular, the mirror process is fixed to say that any transition in $\mathcal{P}$ of an interfering thread can fire at any point.

Example 5.6
As a concrete example, consider the parametrized system constructed with program COUNTINCRGLOBAL presented in Fig. 5.4.

This program declares a global variable g and a local variable x. A thread executing program COUNTINCRGLOBAL waits at line 1 until the value of the global variable g is positive. As part of the same operation, it saves that value into the local variable x while setting g to 0. Then, it increments the locally saved value and writes it back to the global variable g.

Our first goal is to establish that $g \geq 0$ at every program location. Consider the transition associated to program location ℓ_3. Following the framework described in this chapter, the ideal transition relation for the corresponding MIRROR transition is:

$$(\exists\, x)\ \eta^*(\ell_3) \wedge g' = x$$

Our approach builds η incrementally starting from $\bot$, updating the MIRROR transitions in an eager or a lazy fashion and obtaining $x \geq 0$ everywhere. The interference semantics, on the other hand, is obtained by setting η to $\top$, essentially dropping the guards related to the invariant candidates we have computed. Such a semantics will be too coarse to prove an assertion involving x. Specifically, in the absence of any information about $\eta(\ell_3)$, the interference due to the transition associated to line 3 is written as a non-deterministic update of x. The incremental construction of the reflective abstraction presented here prevents this situation by building invariants on g that can then be used to prove $x \geq 0$.

However, the reflective abstraction approach is not complete either. For instance, reflective abstractions cannot be used to establish the invariant $g = 0$ when all threads are at line 2 or line 3 without the use of additional auxiliary variables. ⌟

```
global
   Int g ≥ 0
procedure COUNTINCRGLOBAL()
   Int x = 0
begin
     ⟨ await g > 0
1:     x := g
       g := 0 ⟩
2: x := x + 1
3: g := x
end procedure
```

Figure 5.4: COUNTINCRGLOBAL: a simple global counter example.

5.3.3 The Effect of Widening

So far, we have defined all iterations via least-fixed points of monotone operators, implicitly assuming abstract domains for which the least-fixed point is computable. However, in practice, abstract interpretation is used with abstract domains that do not enjoy this property. In particular, we want to be able to use abstract domains that rely on widening to enforce convergence to a post-fixed point that over-approximates the least-fixed point.

Applying abstract interpretation with widening instead of lfp in the previous definitions of this section raises a number of issues in an implementation. First, the lazy reflective operator $\widehat{\mathcal{G}}_{\text{LAZY}}$ defined in Equation (5.7) is not necessarily monotonic. To solve this, in our implementation we enforce monotonicity by applying an *outer join* that joins the assertion maps from the previous iteration with the one from the current iteration. To enforce convergence of this iteration, we must apply an *outer widening*.

Another consequence is that the relative precision of the reflective abstract interpretation schemes are unclear. Perhaps counter-intuitively, the interference abstraction approach described in Section 5.3.2 is not necessarily less precise than the reflective abstract interpretation with $\widehat{\mathcal{G}}_{\text{EAGER}}$ as defined in Equation (5.9). To see this possibility, let $\widehat{\eta}^*_{\text{EAGER}}$ be the fixed point abstract assertion map computed by iterating $\widehat{\mathcal{G}}_{\text{EAGER}}$. While the final reflective abstraction $\mathcal{S}_{\text{EAGER}}: \text{REFLECT}_{\mathcal{P}}(\gamma \circ \widehat{\eta}^*_{\text{EAGER}})$ using $\widehat{\mathcal{G}}_{\text{EAGER}}$ is trivially no less precise than the interference abstraction $\mathcal{S}_{\text{INTERFERE}}: \text{REFLECT}_{\mathcal{P}}(true)$, the abstract interpretation with widening is not guaranteed to be monotonic. Instead, this observation suggests another scheme, which we call eager+. The eager+ scheme runs $\widehat{\mathcal{G}}_{\text{EAGER}}$ to completion to get $\mathcal{S}_{\text{EAGER}}$ and then applies standard abstract interpretation over this sequential transition system. In other words, the eager+ scheme is defined as follows:

$$\widehat{\eta}^*_{\text{EAGER}} = \mathsf{lfp}\ \widehat{\mathcal{G}}_{\text{EAGER},\mathcal{P}}(\bot) \qquad \widehat{\eta}^*_{\text{EAGER+}} = \mathsf{lfp}\ \widehat{\mathcal{F}}(\bot, \text{REFLECT}_{\mathcal{P}}(\gamma \circ \widehat{\eta}^*_{\text{EAGER}})) \tag{5.11}$$

Empirical evaluation using our parametrized invariant generation approach is presented in Section 10.6 in Chapter 10. There, we present further examples, we evaluate our parametrized invariant generation method and we analyze the advantages and disadvantages of the lazy, eager and eager+ schemes applied to various examples.

5.4 Summary

In this chapter we have described the *reflective abstraction* approach for automatically inferring k-indexed invariants of parametrized systems. This method can be seen as an effective way to automatically infer invariants for supporting the verification methods described in Chapter 3 and Chapter 4.

A novel aspect of the technique presented in this chapter is that it enables the use of off-the-shelf abstract interpretation-based sequential invariant generators for the generation of parametrized invariants. The central idea of our approach is that inferences made on materialized threads can be transferred or reflected on to the summarized threads represented by the MIRROR thread. In order to study the behaviour of our method, we introduced three variants of reflective abstraction named lazy, eager and eager+. Later, in Chapter 10 we present some results we have obtained from the empirical evaluation of these schemes on a collection of parametrized programs.

Part II

Decision Procedures

6

TL3: A Decidable Theory of Concurrent Lists

> *Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.*
>
> Isaac Asimov

In Chapter 3 and Chapter 4 we introduced deductive techniques to verify safety and liveness properties of parametrized systems. Such deductive methods discharge a collection of verification conditions that needs to be validated. With the assistance of specialized decision procedures for complex data types, such as linked lists, it is possible to automatically verify the validity of the generated verification conditions.

In this chapter we present TL3—the *Theory of Linked Lists with Locks*— a decidable theory for single-linked lists, which is also equipped with features to reason about concurrency. We use TL3 to study the verification problem of concurrent data types that manipulate dynamic memory in the heap in the shape of single-linked lists. Later, in Chapter 10 we show how to use an implementation of our TL3 decision procedure in order to verify list shape preservation and functional correctness of implementations of many single-linked list data structures, including lock-coupling lists, lock-based unbounded queues and lock-free queues and stacks.

We show that TL3 enjoys the so called *bounded model property*. This property states that if a TL3 formula has a model, then it must also have a model within a bounded domain. By showing that models of TL3 formulas can be bounded just by analyzing the literals in the formula, we can conclude that the satisfiability problem for quantifier-free TL3 formulas is decidable. This

decidability result comes from the fact that if there exist bounds for the domains, then enumerate all possible models within the bounds.

The core of TL3 is a decidable theory for sequential lists introduced in [178]. The theory TL3 incorporates native built-in predicates that make its quantifier-free fragment powerful enough to describe rich properties of concurrent programs that manipulate single-linked lists. In TL3 it is possible to express list-like data structures, pointers, explicit heap regions with region separation and lock ownership. This expressive power allows TL3 to verify many concurrent data types lists such as concurrent queues and stacks. As we will see in future chapters, TL3 serves as an essential building block in the construction of richer theories for more sophisticated memory shapes. For example, in Chapter 7 we show how to extend TL3 into TSL_K, a family of theories capable of reasoning about concurrent skiplists.

The rest of the chapter is structured as follows. Section 6.1 presents some concurrent data types that manipulate single-linked lists shapes. Section 6.2 formally presents the theory TL3. Section 6.3 shows that TL3 is decidable by stating and proving a bounded model theorem. Finally, Section 6.4 presents a summary of what have been discussed in this chapter. Later, in Chapter 10 we report the results of our empirical evaluation using an implementation of a TL3 decision procedure based on the bounded model theorem of this chapter.

6.1 Concurrent Data Types that Manipulate Lists

In this section we introduce concurrent data types that manipulate lists in the heap. We present implementations of four different concurrent data structures based on single-linked lists:

(a) An implementation of a concurrent lock-coupling single-linked list.

(b) An implementation of an unbounded concurrent queue which uses a single lock.

(c) An implementation of a lock-free stack.

(d) An implementation of a lock-free non-blocking queue.

6.1.1 A Concurrent Lock-coupling List

A lock-coupling concurrent list [107, 207] is a concurrent data type that implements a set by maintaining in the heap an ordered single-linked list with non-repeating elements. Each node in the list is protected by a lock which guarantees that a single thread can access a list node at the same time. When a thread traverses the list, it acquires the lock of the node that it visits, and only releases this lock after the lock of the successor node has been successfully acquired. This technique of protecting cells with locks (instead of protecting the whole data-structure with a single coarse-grain lock) is known as *fine-grained locking*.

Nodes of a concurrent lock-coupling list are instances of the following $ListNode$ class:

$$\begin{array}{llllll} \textbf{class} & ListNode & \{ & Elem & data; & \\ & & & Addr & next; & \\ & & & Lock & lock; & \} \end{array}$$

An object of class $ListNode$ contains the following fields:

- $data$, the value stored in the node, which is also used to keep the list ordered;
- $next$, a pointer that stores the address of the next node in the list; and
- $lock$, which contains the lock protecting the node.

We assume that the operating system provides the atomic operations $lock$ and $unlock$ to lock and unlock the lock of a node. It is easy to adapt this framework with additional fields in $ListNode$. For instance, we could use different fields for the element stored (for example a value) and for the data used to keep the list ordered (a key).

Our implementation of concurrent lock-coupling lists maintains two global addresses $head$ and $tail$, and two ghost global variables reg and $elems$:

```
global
  Addr head
  Addr tail
  Set⟨Addr⟩ reg
  Set⟨Elem⟩ elems
```

The declared global program variables are:

(a) A variable $head$, of type address (pointer), which points to the first node of the list.

(b) A variable $tail$, of type address (pointer), which points to the last node of the list.

(c) A ghost variable reg, of type set of addresses, which is used to keep track of the portion of the heap whose cells form the list.

(d) A ghost variable $elems$, of type set of elements, which represents the collection of elements stored in the list.

In our implementation, $head$ and $tail$ point to the nodes with the lowest and highest possible values, $-\infty$ and $+\infty$ respectively. We consider $head$ and $tail$ sentinel nodes which are neither removed nor modified and we assume that the list is initialized with $head$ and $tail$ already set. The set reg is initialized containing solely the addresses of $head$ and $tail$. Similarly, the set $elems$ is initialized containing only the elements initially stored at the nodes pointed by $head$ and $tail$.

We now present the code of an implementation of concurrent lock-coupling lists. This implementation contains three main operations named SEARCH, INSERT and REMOVE:

SEARCH: described in Fig. 6.1, receives an element e and traverses the list in order to determine whether e is stored in the list. The procedure uses two local auxiliary pointers ($prev$ and $curr$) to traverse the list.

Initially, $prev$ points to the head of the list (line 1) and $curr$ to the node immediately after $head$ (line 3). When a thread begins executing procedure SEARCH, it gets the lock of both $prev$ and $curr$ (line 2 and 4) in order to prevent other threads from modifying these nodes concurrently.

The loop (lines 5 to 11) performs the search of a node containing e. In the loop, an auxiliary pointer aux is used while $prev$ is reassigned to point to $curr$ (line 7). This is the way a thread traverses the list. Once $prev$ points to the same node as $curr$ and the thread has released the lock at the node previously pointed by $prev$ (line 8), $curr$ can be modified in

```
     procedure SEARCH(Elem e)
        Addr prev, curr, aux
        Bool found
     begin
 1:  prev := head
 2:  lock(prev→lock)
 3:  curr := prev→next
 4:  lock(curr→lock)
 5:  while curr→data < e do
 6:      aux := prev
 7:      prev := curr
 8:      unlock(aux→lock)
 9:      curr := curr→next
10:      lock(curr→lock)
11:  end while
12:  found := (curr→data = e)
13:  unlock(prev→lock)
14:  unlock(curr→lock)
15:  return found
     end procedure
```

Figure 6.1: SEARCH procedure for concurrent lock-coupling lists.

order to point to the next node in the list (line 9). Once $prev$ is set, the thread grabs the lock at the node currently pointed by $curr$ (line 10). At the end of the loop, the following holds:

$$prev{\rightarrow}data < e \wedge curr{\rightarrow}data \geq e$$

Hence, if the list is ordered, it can be determined whether e was present in the list or not (line 12) just by checking the value stored at the node pointed by $curr$. If element e was in the list then $found$ is set to $true$, otherwise $found$ is set to $false$. Finally, the locks at the nodes pointed by $prev$ and $curr$ are released (line 13 and 14) and $found$ is returned. Procedure SEARCH returns $true$ if e was found and $false$ otherwise.

INSERT: depicted in Fig. 6.2, receives an element e and attempts to insert it into the list. INSERT first determines the position at which e should be inserted and then manipulates the pointers of the neighbor nodes appropriately.

This procedure initially behaves as SEARCH, looking for the position at which element e should be inserted into the list. In fact, from line 1 to 11 procedure SEARCH and INSERT are identical. Note that, again, after the initial loop of INSERT (line 12) the following holds:

$$prev{\rightarrow}data < e \wedge curr{\rightarrow}data \geq e$$

Hence, as $curr{\rightarrow}data \geq e$, the conditional at line 12 checks whether the element stored at the node pointed by $curr$ is e or not. If the test at line 12 is false, then $curr{\rightarrow}data = e$ and the element e is already present in the list. In this case we do not need proceed with the insertion element e and we just require to release the locks on $prev$ and $curr$ (lines 16 and 18) before exiting. On the contrary, if the test at 12 holds, e is not in the list and we need to insert it.

```
        procedure INSERT(Elem e)
          Addr  prev, curr, aux, newnode
        begin
 1:     prev := head
 2:     lock(prev→lock)
 3:     curr := prev→next
 4:     lock(curr→lock)
 5:     while curr ≠ null ∧ curr→data < e do
 6:         aux := prev
 7:         prev := curr
 8:         unlock(aux→lock)
 9:         curr := curr→next
10:         lock(curr→lock)
11:     end while
12:     if curr ≠ null ∧ curr→data > e then
13:         newnode := malloc(e)
14:         newnode→next := curr
15:         prev→next := newnode
            ┌─────────────────────────┐
            │reg := reg ∪ {newnode}   │
            │elems := elems ∪ {e}     │
            └─────────────────────────┘
16:     end if
17:     unlock(prev→lock)
18:     unlock(curr→lock)
19:     return
        end procedure
```

Figure 6.2: INSERT procedure for concurrent lock-coupling lists.

To insert e in the list, a new node is created (line 13) and assigned value e. We assume that the node created by a call to $malloc$ is initialized with $next = null$ and $lock = \#$. At this point:

$$prev \rightarrow data < e \wedge curr \rightarrow data > e$$

Hence node $newnode$, which contains element e, needs to be inserted between the nodes pointed by $prev$ and $curr$. At line 14 we modify $newnode$'s $next$ pointer to point to $curr$ and at line 15 we make $prev$'s $next$ pointer point to $newnode$, finally connecting $newnode$ to the rest of the list. Line 15 is annotated with ghost code which updates reg and $elems$ to keep track of the new cell and element inserted in the list. After INSERT has been executed, e is guaranteed to be present in the list.

REMOVE: described in Fig. 6.3, receives as argument an element e and removes the node containing element e from the list by redirecting the next pointer of the previous node appropriately.

The first section of REMOVE (line 1 to 11) behaves again similar to SEARCH, looking for the position in the list at which element e should be. After the loop (line 12):

$$prev \rightarrow data < e \wedge curr \rightarrow data \geq e$$

Again the test at line 12 captures whether the node pointed by $curr$ contains e and whether it is not $tail$. This check can be omitted if we assume that element e cannot be neither $-\infty$

```
    procedure REMOVE(Elem e)
       Addr  prev, curr, aux
    begin
1:  prev := head
2:  lock(prev→lock)
3:  curr := prev→next
4:  lock(curr→lock)
5:  while curr ≠ tail ∧ curr→data < e do
6:      aux := prev
7:      prev := curr
8:      unlock(aux→lock)
9:      curr := curr→next
10:     lock(curr→lock)
11: end while
12: if curr ≠ tail ∧ curr→data = e then
13:     aux := curr→next
14:     prev→next := aux
        reg := reg − {curr}
        elems := elems − {e}
15: end if
16: unlock(prev→lock)
17: unlock(curr→lock)
18: return
    end procedure
```

Figure 6.3: REMOVE procedure for concurrent lock-coupling lists.

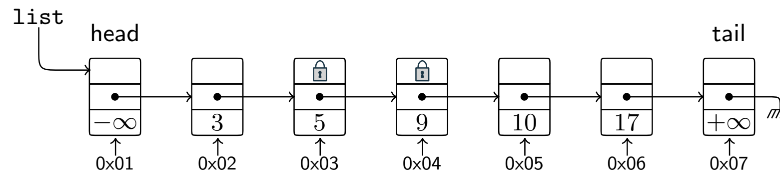
nor $+\infty$. If the node pointed by $curr$ contains a value different from e or $curr$ is pointing to $tail$, then REMOVE does nothing except releasing the locks of $prev$ and $curr$ (line 16 and 17). On the contrary, if $curr$ points to a node different form $tail$ which stores element e, then we use pointer aux to overrun the node pointed by $curr$. To do so, first, aux is assigned the node immediately after $curr$ (line 13). Then, the $next$ pointer of the node pointed by $prev$ is modified in order to point to aux (line 14), making $curr$ unreachable from the head of the list. Line 14 also contains the ghost code which updates the values of reg and $elems$, tracking the effect of the actual removal of the node.

```
    procedure MGCLIST()
       Elem e
    begin
1:  while true do
2:      e := havocListElem()
3:      nondet choice
4:              call SEARCH(e)
5:          or  call INSERT(e)
6:          or  call REMOVE(e)
7:      end choice
8:  end while
    end procedure
```

Figure 6.4: Most general client procedure MGCLIST for concurrent lock-coupling lists.

It is easy to see that, in fact, line 15 of procedure INSERT and line 14 of procedure REMOVE correspond to the linearization points of the concurrent data type. Finally, in order to verify the data type against all possible clients, we create the most general client of the concurrent lock-coupling list, called MGCLIST. The most general client is presented in Fig. 6.4. This program consists of an infinite loop which simply invokes non-deterministically any of the procedures SEARCH, INSERT and REMOVE that implements the data type, with an arbitrary parameter.

Example 6.1
The following figure shows an example of a memory snapshot of the heap maintained by a concurrent lock-coupling list.



This list stores the set of elements $\{3, 5, 9, 10, 17\}$. The elements $-\infty$ and $+\infty$ correspond to the sentinel values stored at the nodes pointed the $head$ and $tail$. In this example, we use 0x01 . . . 0x07 to denote memory addresses. This particular memory snapshot is obtained as a result of an execution in which a thread has acquired the locks protecting the nodes stored in memory addresses 0x03 and 0x04. ⌟

6.1.2 An Unbounded Queue

We now present other data types based on single-linked lists memory shapes. The data type we present now is a concurrent unbounded queue [107]. This queue is unbounded because it can potentially store any number of elements.

A queue is a data structure capable of storing elements and later retrieving them in the same order as they were inserted. Because of this, it is also called *first-in, first-out* queue.

Our implementation of an unbounded concurrent queue is depicted in Fig. 6.5. The implementation contains three global program variables:

(a) A variable $head$, of type address (pointer), which points to the first node in the queue, that is, the oldest element residing in the queue.

(b) A variable $tail$, of type address (pointer), which points to the last node in the queue, that is, the latest element inserted into the queue.

(c) A single global lock, named $queueLock$, to protect the access to the queue allowing that a single thread at a time modifies the queue.

This technique of protecting the whole data structure with a single global locks is known as *coarse-grained locking*. Nodes of an unbounded concurrent queue are implemented by the $UnboundedQueueNode$ class:

$$\textbf{class} \quad UnboundedQueueNode \quad \{ \quad \begin{array}{ll} Elem & data; \\ Addr & next; \quad \} \end{array}$$

Instances of *UnboundedQueueNode* contains only two fields. A field *data*, to represent values stored in the node, and a field *next* which contains a pointer to the following node in the queue. In this case, the list need not be ordered.

The implementation offers two procedures, one for inserting elements and another for removing elements from the queue:

ENQUEUE: This procedure receives an element e and inserts it into the queue. To do so, first the thread gets the global lock (line 1). Then, a new node *newnode* is created (line 2) storing element e. As new nodes are connected to the tail of the queue, we require to assign *null* to the *next* pointer of the new node (line 3) so that once connected, the queue will remain *null* terminated. At this point, we can modify the current last node of the queue, pointed by *tail*, to make it point to the new node (line 4). This way, after this step, *newnode* comes to be the last node in the list that represents the queue. As global variable *tail* should always point to the last node in the queue, at line 5 we modify *tail* to make it point to the new node. Finally, the thread releases the global lock (line 6) and the procedure finishes.

DEQUEUE: This procedure is in charge of removing elements from the queue. A thread running this procedure begins by performing a wait on the global lock *lock* (line 8). Then it checks whether the queue is empty (line 9). If $head \rightarrow next$ happens to be *null*, then this means that there are no elements to be removed from the queue. This happens because, in fact, the queue's actual head is not the node referenced by *head*. Instead, the actual head is the successor of the node referenced by *head*. The node pointed by *head* is just used as a sentinel node. So, in case $head \rightarrow next = null$, it means that there are no elements to remove from the queue, in which case the global lock is released and an *EmptyException* is raised. On the contrary, if the queue is not empty, then the element at the queue's head is stored in the local program variable *result* (line 13). Then, *head* is modified to point to the successor node of the node previously pointed by *head* (line 14). This is equivalent to making the old node pointed by *head* unreachable from the new *head*. Finally, the thread

```
global
  Addr head, tail
  Lock queueLock

  procedure ENQUEUE(Elem e)
    Addr n
  begin
1:  lock(queueLock)
2:  newnode := malloc(e)
3:  newnode→next := null
4:  tail→next := newnode
5:  tail := newnode
6:  unlock(queueLock)
7:  return()
  end procedure

  procedure DEQUEUE()
    Elem result
  begin
8:  lock(queueLock)
9:  if head→next = null then
10:     unlock(queueLock)
11:     raise(EmptyException)
12: end if
13: result := head→next→data
14: head := head→next
15: unlock(queueLock)
16: return(result)
  end procedure
```

Figure 6.5: A coarse-grain locking unbounded queue implementation.

releases the global lock (line 15) and returns the element stored at $result$ (i.e., the one that used to be at queue's head) at line 16.

6.1.3 An Unbounded Lock-free Stack

The queue implementation presented in Section 6.1.2 uses a single global lock which provides a coarse synchronization method. This can be a concurrency bottleneck, because the access to the data structure is limited to a single thread at a time.

A more efficient approach consists of using lock-free algorithms. A lock-free algorithms does not rely on locks for synchronization, but uses *compare-and-swap* (CAS) operations instead.

Here we present a lock-free implementation of a stack from [107]. A stack, in contrast with a queue, is a *last-in, first-out* data type. This means that the latest element being inserted will be the first one to be removed from the stack. The implementation we show here is based on a single-linked list. Nodes of the list implementing a stack are instances of the $LockfreeStackNode$ class, which is identical to the $UnboundedQueueNode$ class presented in Section 6.1.2. An object of class $LockfreeStackNode$ contains two fields, $data$ which stores the value and $next$ which references to the next node in the stack.

Fig. 6.6 presents an implementation of a lock-free stack. This implementation uses a single global program variable top, of type address, which is used to keep a reference to the node at the top of the stack. The implementation provides two procedures, one for inserting elements and the other for removing them from the stack.

PUSH: receives an element e as argument and inserts it at the top of the stack. To do so, it first creates a new node, called $newnode$, which stores the value e (line 1). Then, the loop (line 2 to 8) inserts the element e into the stack. The loop uses the local program variable $oldTop$ to point to the same node as top (line 3). Then it modifies the $next$ pointer of $newnode$ to make it reference the addresses pointed by $oldTop$ (line 4). When line 5 is reached, the CAS

```
global
  Addr top

    procedure Push(Elem e)
      Addr oldTop, newTop, n
    begin
 1: newnode := malloc(e)
 2: while true do
 3:     oldTop := top
 4:     newnode→next := oldTop
 5:     if CAS(top, oldTop, n) then
 6:         return()
 7:     end if
 8: end while
    end procedure

    procedure Pop()
      Addr oldTop, newTop
    begin
 9: while true do
10:     oldTop := top
11:     if oldTop = null then
12:         raise(EmptyException)
13:     end if
14:     newTop := oldTop→next
15:     if CAS(top, oldTop, newTop) then
16:         return(oldTop→data)
17:     end if
18: end while
    end procedure
```

Figure 6.6: An unbounded lock-free stack implementation.

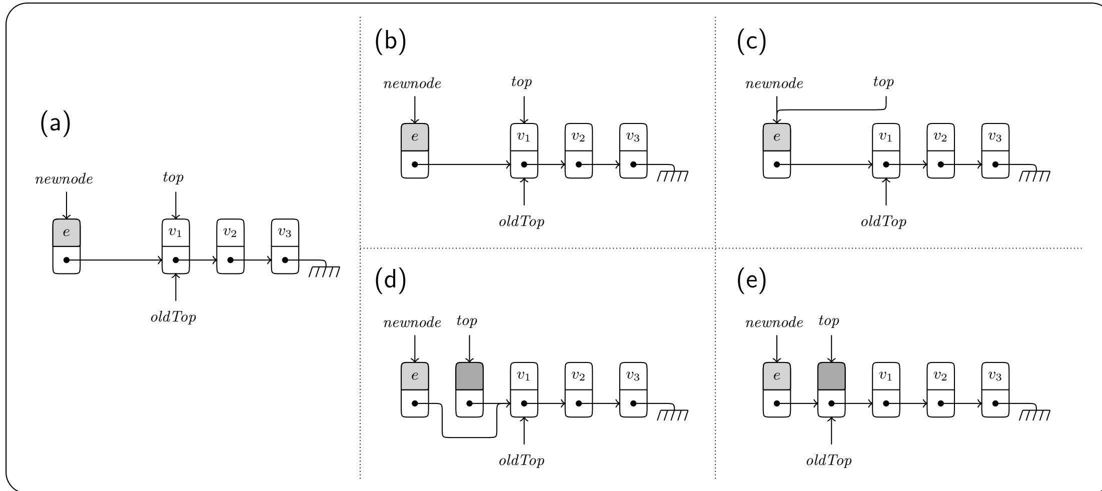


Figure 6.7: Sequence of states reached changes suffered by a lock-free stack during the execution of CAS operation at procedure PUSH: (a) the state of the stack when reaching line 5; (b) the condition of the CAS operation holds; (c) after successful execution of CAS operation; (d) the condition of the CAS operation does not hold; (e) after unsuccessful execution of CAS operation.

operation compares the address referenced by top with the address referenced by $oldTop$. At this point, two things may happen (Fig.6.7(a)):

(1) If no other thread has tried in parallel to insert a new element, the value of top has not been modified by other thread (Fig. 6.7(b)), then the operation CAS succeeds, and top points to $newnode$ (Fig. 6.7(c)). In this situation, element e has been successfully inserted into the stack.

(2) If another thread has inserted a new element in the node while the original thread was still executing the statements at line 3 or 4, then top will not match $oldTop$ any longer (Fig. 6.7(d)). In this case, when the procedure compares the addresses pointed by top and $oldTop$ (line 5), the CAS operation fails and the loop is executed once again. In the next round of the loop, $oldTop$ will be assigned the new current value of top and the comparison will be performed again until it succeeds (Fig. 6.7(e)).

POP: returns the value stored in the node at the top of the stack and removes such node from the data structure. The procedure begins by storing in the local program variable $oldTop$ the address pointed by top (line 10). Then checks whether $oldTop$ points to $null$ (line 11). This test implies that the stack is empty and no node can be removed from the stack. So, in this case the procedure returns an $EmptyException$. If the stack is not empty, then POP uses its local program variable $newTop$ to store the address of the node that follows $oldTop$ (line 14). Then, the procedure checks whether the global program variable top and the local program variable $oldTop$ point to the same node. If they both point to the same address, it means that no node has been removed and that no new node has been added to the stack while the thread was between line 10 and 15. As happened with procedure PUSH, if at line 15 top and $oldTop$ point to the same address, then top is modified to point to $newTop$, that is, the node following the top of the stack. If this CAS operation succeeds, then the procedure ends returning the value stored at the old top node of the stack. If the CAS

operation fails, then the procedure tries to perform the removal again in the next iteration of the loop.

6.1.4 An Unbounded Lock-free Queue

In Section 6.1.2 we presented an unbounded queue implemented on top of a single-linked list which uses a single global lock to prevent the simultaneous access of multiple threads to the queue. Now we present a lock-free non-blocking implementation of an unbounded queue [153] known as Michael-Scott queue.

As before, the queue data structure is implemented as a list of nodes of class $LockfreeQueueNode$. Instances of $LockfreeQueueNode$, as instances of $LockfreeStackNode$, maintain two fields:

(a) A $data$ field, of type $Elem$, for the node's value.

(b) A $next$ field, of type $Addr$ (pointer), which keeps the reference of the successor node.

The implementation of the lock-free queue is presented in Fig. 6.8. This implementation uses two global program variables of type $Addr$:

(a) Global variable $head$, pointing to the first node in the queue.

(b) Global variable $tail$, pointing to the last node in the queue.

As in the unbounded queue presented in Section 6.1.2, here the node pointed by $head$ is a sentinel node and its value is not used. The implementation provides two procedures for manipulating the queue:

ENQUEUE: receives an element e and proceeds to insert it at the tail of the queue. An interesting aspect of ENQUEUE is that it is lazy, in the sense that the complete insertion of a new element takes place in two steps. First, a CAS operation (line 8) connects the new node to the queue. Then, if no new nodes have been added between $last$ and $tail$, then pointer $tail$ is updated by the same thread through the CAS operation at line 16. On the contrary, if the new node could not have been connected to the queue because another thread inserted a different node before the CAS operation at line 8 took place, then the thread running ENQUEUE just updates the reference of $tail$ (line 12). In this case, a new attempt to insert $newnode$ will be made in the next iteration of the loop. This means that, in order to make the procedure lock-free, threads may need to help each other in order to fully insert a new element.

Now, if we follow the execution of ENQUEUE step by step, the procedure begins by creating the new node to be inserted (line 1), assigning to it the value e and making its $next$ field to point to $null$ (line 2). The graph in Fig. 6.9 shows a representation of a scenario in which a thread tries to insert a new element into the lock-free queue.

At line 4, a thread running procedure ENQUEUE stores in its local program variable $last$ what at that moment appears to be the address of the last node in the queue (Fig. 6.9(a)). Then, it assigns to local program variable $nextptr$ the address of the successor node of $last$ (line 5). When this assignment is done, as the data type is not protected by a lock, two things may have happened:

- some other thread has inserted a new node in the queue before line 5 is executed, so $nextptr$ may not point to $null$ (Fig. 6.9(b)); or

```
global
  Addr head, tail

  procedure Enqueue(Elem e)
    Addr last, nextptr, n
  begin
 1: newnode := malloc(e)
 2: newnode→next := null
 3: while true do
 4:     last := tail
 5:     nextptr := last→next
 6:     if last = tail then
 7:         if nextptr = null then
 8:             if CAS(last→next, nextptr, newnode) then
 9:                 break
10:             end if
11:         else
12:             CAS(tail, last, nextptr)
13:         end if
14:     end if
15: end while
16: CAS(tail, last, newnode)
17: return()
  end procedure

  procedure Dequeue()
    Addr first, last, nextptr
    Elem value
  begin
18: while true do
19:     first := head
20:     last := tail
21:     nextptr := first→next
22:     if first = head then
23:         if first = last then
24:             if nextptr = null then
25:                 raise(EmptyException)
26:             end if
27:             CAS(tail, last, nextptr)
28:         else
29:             value := nextptr→data
30:             if CAS(head, first, nextptr) then
31:                 break
32:             end if
33:         end if
34:     end if
35: end while
36: return(value)
  end procedure
```

Figure 6.8: An unbounded lock-free queue implementation.

- no other node has been inserted after the node pointed by $last$, and hence $nextptr$ indeed points to $null$ (Fig. 6.9(c)).

Line 6 checks whether $last$ and $tail$ do still reference to the same address. Let's first consider what happens in the case of the state shown in Fig. 6.9(b). In this case, if $last$ is not pointing to the same address as $tail$ (Fig. 6.9(e)), then the condition of line 6 fails and the algorithm needs to try again the insertion, reassigning again $last$ to the current $tail$ of the queue and returning to the state shown in Fig. 6.9(a). On the contrary, if $last$ and $tail$ do point to the same address (Fig. 6.9(f)), then condition at line 7 fails as we are considering the scenario at which $nextptr$ is not $null$. This situation happens when, while we were inserting $newnode$, another thread inserted a new node after $last$ but it could not finished with the update of $tail$, and hence, pointer $tail$ is not currently pointing to the last node in the queue. In order to assist in the update of $tail$, the CAS operation at line 12 is executed on behalf of the other thread. If the CAS operation succeeds then $tail$ is modified in order to point

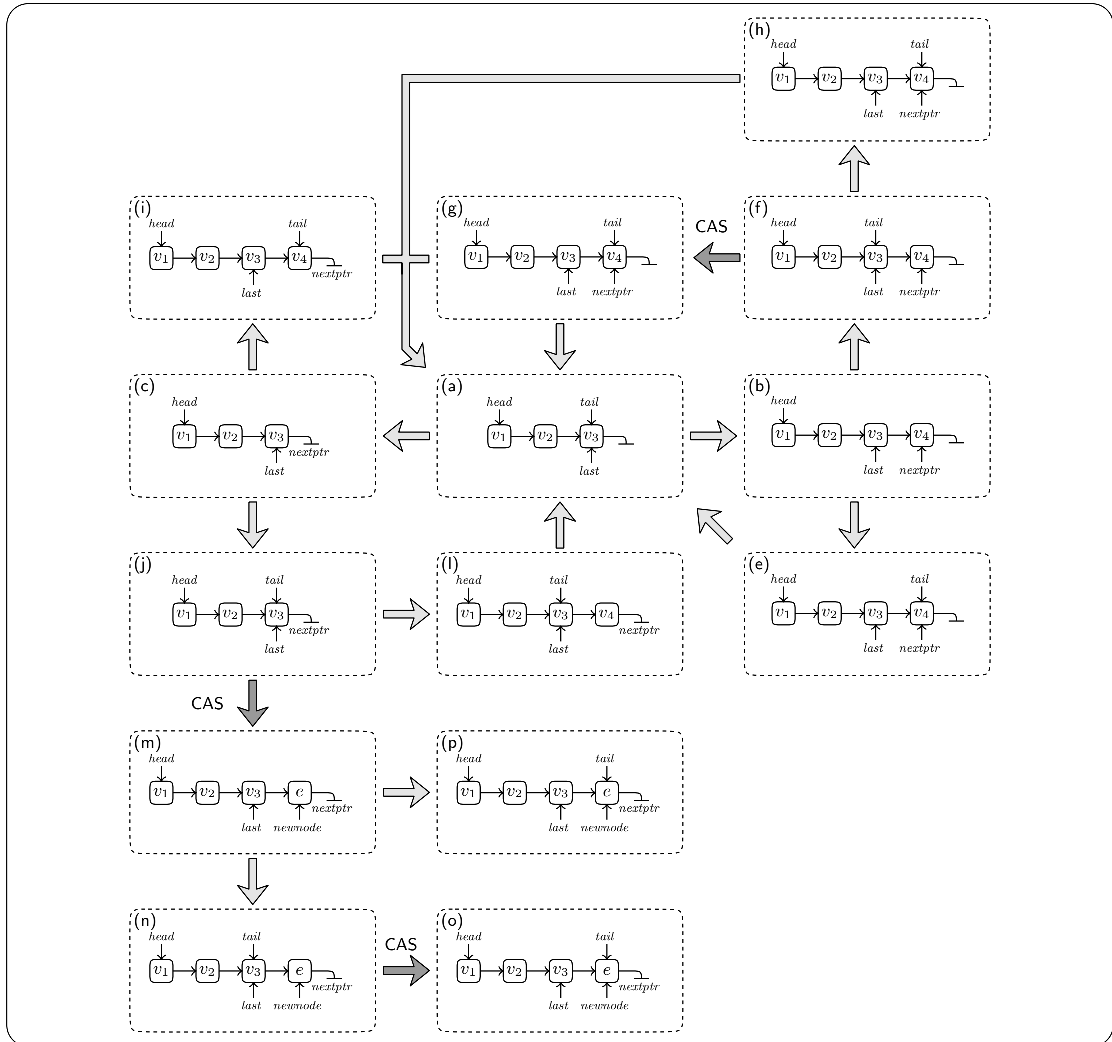


Figure 6.9: Sequence of steps for inserting a new element into a lock-free queue implementation.

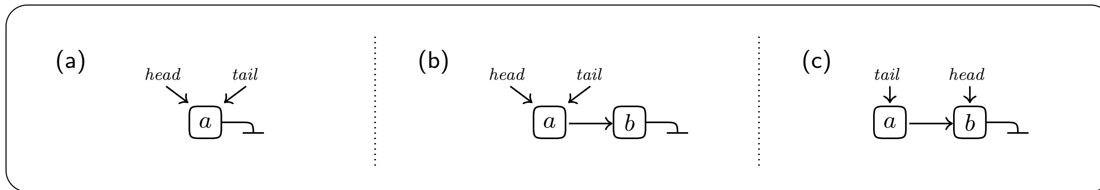


Figure 6.10: Potential problem in the lock-free queue implementation if DEQUEUE does not assist ENQUEUE in advancing *tail* pointer.

to *nextptr* (Fig. 6.9(g)). In this scenario the new element e has not been inserted into the queue, but at least we contributed updating the value of *tail*. As e was not successfully inserted, we restart the process and try once again to insert e into the queue (Fig. 6.9(a)). Now, it is possible that in the interleaving between the moment in which *last* is compared to be equal to *tail* (line 6) and the execution of the CAS operation (line 12) the value of *tail* is updated. In this case, the CAS operation fails (Fig. 6.9(h)) and the procedure will require to try to insert the element in a new iteration of the loop (Fig. 6.9(a)).

Let's consider now the case shown in Fig. 6.9(c) right before executing line 6. Again, if *tail* has been updated by another thread, then the condition in line 6 will fail (Fig. 6.9(i)) and the procedure will require a new iteration of the loop in order to insert element e (Fig. 6.9(a)). On the contrary, if the conditional at line 6 succeeds, then condition at line 7 will clearly also succeed. Then, the CAS operation at line 8 is executed. If the CAS operation fails, it means that new nodes have been inserted by other threads between *last* and *null* (Fig. 6.9(l)) and hence a new iteration of the loop is required in order to insert element e in the queue. If the CAS operation succeeds, this means that the new node *newnode* with value e has been successfully inserted into the queue (Fig. 6.9(m)). Then, we just require to update *tail* to reflect that a new node has been inserted at the tail of the queue. The process of updating *tail* is done by the last CAS operation at line 16. If at line 16 *tail* and *last* points to the same address (Fig. 6.9(n)) then CAS operation is successful and *tail* is updated pointing to the new inserted node (Fig. 6.9(o)). On the contrary, if at line 16 *tail* does not point to the same address as *last*, this means that *tail* has been updated by another thread through the CAS operation at line 12 and consequently procedure ENQUEUE does not need to perform any other update (Fig. 6.9(p)).

DEQUEUE: removes the element that is at the head of the queue and returns the value that was stored in such node. The removal is made by changing *head* from the sentinel node to its successor, making the successor node the new sentinel.

The procedure begins by assigning to local program variables *first* and *last* the addresses of the nodes pointed at that moment by *head* and *tail* respectively (line 19 and 20). Then, the procedure makes a local copy at program variable *nextptr* of the address that follows *head* in the queue. Line 22 checks whether *head* has been modified since the moment its value was assigned to *first*. If so, then the procedure starts again in the next iteration of the loop.

There may be a problem when ENQUEUE and DEQUEUE interact. Consider the situation of a queue with a single node, where *head* and *tail* point to the same node. This situation is depicted in Fig. 6.10(a). Here *head* and *tail* point to the same node a. Now, imagine that

one thread enqueuing b has redirected a's $next$ pointer to b but has not yet redirected $tail$ form a to b (Fig. 6.10(b)). At this point, if another thread starts dequeuing, it will read b's value and redirect $head$ from a to b, removing a while $tail$ is still pointing to it (Fig. 6.10(c)). To prevent this problem from happening, procedure DEQUEUE must assist in advancing $tail$ from a to b before redirecting $head$. Line 23 checks whether $first$ and $tail$ point to the same address. If they do point to the same address and its successor is $null$, then an exception is raised indicating that the queue is empty (line 25). If the queue is not empty, then the $tail$ pointer is advanced until $nextptr$ (line 27) and the DEQUEUE procedure proceeds to try the dequeuing in the next iteration of the loop. On the other hand, if $first$ and $last$ do not match, then the value stored in the node following $last$ is retrieved (line 29) and a CAS operation is performed (line 30). The CAS operation checks whether no other thread has performed a DEQUEUE operation modifying the reference of $head$. If $head$ still matches $first$, then the sentinel node $head$ is removed, $nextptr$ becomes the new sentinel node and the procedure returns the value retrieved in line 29. If $head$ does not match $first$, then the procedure for removing a node starts again in the next iteration of the loop.

6.2 TL3: A Theory of Concurrent Single-Linked Lists

In this section we present the *Theory of Linked Lists with Locks* TL3, a theory we have developed for describing linked-list heap memory layouts. TL3 is powerful enough as to cope with the examples we have presented in the previous sections, as we will see later.

The theory TL3 is a multi-sorted first-order theory. We formally define the *Theory of Linked Lists with Locks* TL3 as a combination of theories $\mathsf{TL3} = (\Sigma_{\mathsf{TL3}}, \mathbf{TL3})$, where

$$\Sigma_{\mathsf{TL3}} := \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{setaddr}} \cup \Sigma_{\mathsf{settid}} \cup \Sigma_{\mathsf{setelem}} \cup \Sigma_{\mathsf{reach}} \cup \Sigma_{\mathsf{bridge}}$$

and **TL3** is a class of models of interest presented later in Section 6.2.3.

Informally, Σ_{cell} models *cells*, structures containing an element (data), an address (pointer) and a lock owner, which represents a node in a linked list. Σ_{mem} models the memory. $\Sigma_{\mathsf{setaddr}}$ models sets of addresses. Σ_{settid} models sets of thread identifiers. $\Sigma_{\mathsf{setelem}}$ models sets of elements. Σ_{reach} models finite sequences of non-repeating addresses, to represent acyclic paths in memory. Finally, Σ_{bridge} is a *bridge theory* containing auxiliary functions, for example, that allow to map paths of addresses to set of addresses, or to obtain the set of addresses reachable from a given address following a chain of $next$ fields.

We describe now the sorts, the signature and restrictions on the interpretation for each of the theories in TL3.

6.2.1 Sorts

The sorts shared among these theories are cell, elem, addr, tid, mem, path, setaddr, settid and setelem. The intended meaning of these sorts is:

- cell: instances of the class $ListNode$ representing a node in the list.
- elem: elements stored in these cells.
- addr: memory addresses.

- tid: thread identifiers.
- mem: heaps, as maps from addresses to cells.
- path: paths, as finite sequences of non-repeating addresses.
- setaddr: sets of addresses.
- settid: sets of thread identifiers.
- setelem: sets of elements.

Interpretations $\mathcal{A}$ restrict the domains of sorts to satisfy the following:

(a) $\mathcal{A}_{\mathsf{elem}}$, $\mathcal{A}_{\mathsf{tid}}$ and $\mathcal{A}_{\mathsf{addr}}$ are discrete sets.

(b) $\mathcal{A}_{\mathsf{cell}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{addr}} \times \mathcal{A}_{\mathsf{tid}}$.

(c) $\mathcal{A}_{\mathsf{mem}} = \mathcal{A}_{\mathsf{cell}}^{\mathcal{A}_{\mathsf{addr}}}$.

(d) $\mathcal{A}_{\mathsf{path}}$ is the set of all finite sequences of (pairwise) distinct elements of $\mathcal{A}_{\mathsf{addr}}$.

(e) $\mathcal{A}_{\mathsf{setaddr}}$ is the power-set of $\mathcal{A}_{\mathsf{addr}}$.

(f) $\mathcal{A}_{\mathsf{settid}}$ is the power-set of $\mathcal{A}_{\mathsf{tid}}$.

(g) $\mathcal{A}_{\mathsf{setelem}}$ is the power-set of $\mathcal{A}_{\mathsf{elem}}$.

6.2.2 Signature

We describe next the signature of each theory, listing the sorts used and each of the functions and predicates with their signatures:

a) Σ_{cell}: The sorts used are cell, elem, addr and tid. The function symbols are:

$$\begin{array}{rcl} error & : & \mathsf{cell} \\ mkcell & : & \mathsf{elem} \times \mathsf{addr} \times \mathsf{tid} \to \mathsf{cell} \\ _.data & : & \mathsf{cell} \to \mathsf{elem} \\ _.next & : & \mathsf{cell} \to \mathsf{addr} \\ _.lockid & : & \mathsf{cell} \to \mathsf{tid} \\ _.lock & : & \mathsf{cell} \to \mathsf{tid} \to \mathsf{cell} \\ _.unlock & : & \mathsf{cell} \to \mathsf{cell} \end{array}$$

The cell $error$ is used to model the return of an incorrect memory dereference. The function $mkcell$ is the constructor of cells. The corresponding selectors are the functions $data$, $next$ and $lockid$. Finally, the functions $lock$ and $unlock$ model the effect of acquisition and release of locks. There are no predicate symbols in Σ_{cell} except from cell equality.

b) Σ_{mem}: The sorts used are mem, addr and cell. The function symbols are:

$$\begin{array}{rcl} null & : & \mathsf{addr} \\ _[_] & : & \mathsf{mem} \times \mathsf{addr} \to \mathsf{cell} \\ upd & : & \mathsf{mem} \times \mathsf{addr} \times \mathsf{cell} \to \mathsf{mem} \end{array}$$

The function $null$ models the null address. The function $_[_]$ models a memory dereference which returns a cell given a memory and an address. Finally, the function upd is used to create a modified heap given a heap, an address and the cell to be stored in that address. There are no predicate symbols in Σ_{cell} except from memory equality.

c) $\mathbf{\Sigma_{setaddr}}$: The sorts used are addr and setaddr. The function symbols are:

$$\begin{array}{rcl} \emptyset & : & \mathsf{setaddr} \\ \{_\} & : & \mathsf{addr} \rightarrow \mathsf{setaddr} \\ \cup, \cap, \setminus & : & \mathsf{setaddr} \times \mathsf{setaddr} \rightarrow \mathsf{setaddr} \end{array}$$

The predicate symbols in $\Sigma_{\mathsf{setaddr}}$, in addition to set equality, are:

$$\begin{array}{rcl} \in & : & \mathsf{addr} \times \mathsf{setaddr} \\ \subseteq & : & \mathsf{setaddr} \times \mathsf{setaddr} \end{array}$$

The intended interpretation of these symbols is their usual meaning in set theory, for sets of addresses.

d) $\mathbf{\Sigma_{settid}}$: The sorts used are tid and settid. The function symbols are:

$$\begin{array}{rcl} \emptyset_{\mathrm{T}} & : & \mathsf{settid} \\ \{_\}_{\mathrm{T}} & : & \mathsf{tid} \rightarrow \mathsf{settid} \\ \cup_{\mathrm{T}}, \cap_{\mathrm{T}}, \setminus_{\mathrm{T}} & : & \mathsf{settid} \times \mathsf{settid} \rightarrow \mathsf{settid} \end{array}$$

The predicate symbols in Σ_{settid}, in addition to set equality, are:

$$\begin{array}{rcl} \in_{\mathrm{T}} & : & \mathsf{tid} \times \mathsf{settid} \\ \subseteq_{\mathrm{T}} & : & \mathsf{settid} \times \mathsf{settid} \end{array}$$

Again, the intended interpretation of these symbols is their usual meaning in set theory, for sets of thread identifiers.

e) $\mathbf{\Sigma_{setelem}}$: The sorts used are elem and setelem. The function symbols are:

$$\begin{array}{rcl} \emptyset_{\mathrm{E}} & : & \mathsf{setelem} \\ \{_\}_{\mathrm{E}} & : & \mathsf{elem} \rightarrow \mathsf{setelem} \\ \cup_{\mathrm{E}}, \cap_{\mathrm{E}}, \setminus_{\mathrm{E}} & : & \mathsf{setelem} \times \mathsf{setelem} \rightarrow \mathsf{setelem} \end{array}$$

The predicate symbols in $\Sigma_{\mathsf{setelem}}$, in addition to set equality, are:

$$\begin{array}{rcl} \in_{\mathrm{E}} & : & \mathsf{elem} \times \mathsf{setelem} \\ \subseteq_{\mathrm{E}} & : & \mathsf{setelem} \times \mathsf{setelem} \end{array}$$

Once again, the intended interpretation of these symbols is their usual meaning in set theory, for sets of elements.

f) Σ_{reach}: The sorts used are mem, addr and path. The function symbols are:

$$\begin{array}{rcl} \epsilon & : & \mathsf{path} \\ [_] & : & \mathsf{addr} \rightarrow \mathsf{path} \end{array}$$

The constant ϵ models the empty path, and the function $[_]$ allows to build a singleton path with one address in it. The predicate symbols in Σ_{reach}, in addition to path equality, are:

$$\begin{array}{rcl} append & : & \mathsf{path} \times \mathsf{path} \times \mathsf{path} \\ reach & : & \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{path} \end{array}$$

The predicate $append$ relates two paths with its concatenation. A path must be a sequence of non-repeated elements, so some pairs of paths cannot be concatenated, if they contain common elements. The predicate $reach$ relates two addresses with the path that connects them in a given memory.

g) Σ_{bridge}: The sorts used are mem, addr, setaddr and path. The function symbols are:

$$\begin{array}{rcl} path2set & : & \mathsf{path} \rightarrow \mathsf{setaddr} \\ addr2set & : & \mathsf{mem} \times \mathsf{addr} \rightarrow \mathsf{setaddr} \\ getp & : & \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \rightarrow \mathsf{path} \\ fstlock & : & \mathsf{mem} \times \mathsf{path} \rightarrow \mathsf{addr} \end{array}$$

The function $path2set$ returns the set of addresses present in a given path. The function $addr2set$ returns the set of addresses reachable from a given address by following the $next$ pointers. The function $getp$ returns the path that connects two addresses in a given heap, if there is one (or the empty path otherwise). Finally, the function $fstlock$ returns the first address in a given path that is locked by some thread. There are no predicate symbols in Σ_{bridge}.

6.2.3 Interpretations

We restrict the class of models to **TL3**, a class of Σ_{TL3}-structures that satisfy the following conditions:

a) Σ_{cell}: Every interpretation $\mathcal{A}$ of Σ_{cell} must satisfy that for every element $e \in \mathcal{A}_{\mathsf{elem}}$, every address $a \in \mathcal{A}_{\mathsf{addr}}$ and every thread identifiers $t, t_1 \in \mathcal{A}_{\mathsf{tid}}$:

- $mkcell^{\mathcal{A}}(e, a, t) = \langle e, a, t \rangle$
- $\langle e, a, t \rangle.data^{\mathcal{A}} = e$
- $\langle e, a, t \rangle.next^{\mathcal{A}} = a$
- $\langle e, a, t \rangle.lockid^{\mathcal{A}} = t$
- $\langle e, a, t \rangle.lock^{\mathcal{A}}(t_1) = \langle e, a, t_1 \rangle$
- $\langle e, a, t \rangle.unlock^{\mathcal{A}} = \langle e, a, \oslash \rangle$
- $error^{\mathcal{A}}.next^{\mathcal{A}} = null^{\mathcal{A}}$

Essentially, the models in **TL3** restrict cells to be records consisting of an element, an address and a thread identifier. The functions $lock$ and $unlock$ allow to assign and query the thread identifier.

b) Σ_{mem}**:** For each $m \in \mathcal{A}_{\mathsf{mem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $c \in \mathcal{A}_{\mathsf{cell}}$:

- $m[a]^{\mathcal{A}} \quad = \quad m(a)$
- $upd^{\mathcal{A}}(m, a, c) \quad = \quad m_{a \mapsto c}$
- $m^{\mathcal{A}}(null^{\mathcal{A}}) \quad = \quad error^{\mathcal{A}}$

In models in **TL3**, the memory dereference function returns the cell associated with a given address. The memory update simply transforms the memory into a memory that differs only in the modified address, which points to the given cell. Above, $m_{a \mapsto c}$ denotes a memory map that matches in every address with m, except for address a, which is mapped to cell c. Finally, dereferencing $null$ must return the $error$ cell.

c) $\Sigma_{\mathsf{setaddr}}$, $\Sigma_{\mathsf{setelem}}$, Σ_{settid}**:** The symbols $\emptyset$, $\{_\}$, $\cup$, $\cap$, $\setminus$, $\in$ and $\subseteq$ are interpreted according to their standard interpretation over sets of addresses. Similarly, the symbols $\emptyset_{\mathrm{E}}$, $\{_\}_{\mathrm{E}}$, $\cup_{\mathrm{E}}$, $\cap_{\mathrm{E}}$, $\setminus_{\mathrm{E}}$, $\in_{\mathrm{E}}$ and $\subseteq_{\mathrm{E}}$, and the symbols $\emptyset_{\mathrm{T}}$, $\{_\}_{\mathrm{T}}$, $\cup_{\mathrm{T}}$, $\cap_{\mathrm{T}}$, $\setminus_{\mathrm{T}}$, $\in_{\mathrm{T}}$ and $\subseteq_{\mathrm{T}}$ are interpreted according to their standard interpretations over sets of elements and threads respectively.

d) Σ_{reach}**:** The symbol ϵ is interpreted as the empty sequence, and $[i]^{\mathcal{A}}$ must be the singleton sequence containing $i \in \mathcal{A}_{\mathsf{addr}}$ as its only element.

- In the case of $append$:

$$([i_1, \ldots, i_n], [j_1, \ldots, j_m], [i_1, \ldots, i_n, j_1, \ldots, j_m]) \in append^{\mathcal{A}}$$

  if and only if all $i_1, \ldots, i_n, j_1, \ldots, j_m$ are all pairwise distinct.

- In the case of $reach$:

$$(m, i, j, p) \in reach^{\mathcal{A}}$$

  if and only if one of the following conditions hold:

  (a) $i = j$ and $p = \epsilon$; or

  (b) there exist addresses $i_1, \ldots, i_n \in \mathcal{A}_{\mathsf{addr}}$ such that:

  (1) $p = [i_1, \ldots, i_n]$ (3) $m(i_r).next^{\mathcal{A}} = i_{r+1}$, for $1 \leq r < n$

  (2) $i_1 = i$ (4) $m(i_n).next^{\mathcal{A}} = j$

  Note that according to the definition of $reach$, the path p that goes from address i to j must include all addresses reachable through $next$ from i to j, except from j which is not included in p.

e) Σ_{bridge}**:** The interpretation of $addr2set$, $path2set$, $getp$ and $fstlock$ are restricted as follows:

- In the case of $addr2set$:

$$addr2set^{\mathcal{A}}(m, i) \quad = \quad \left\{ j \in \mathcal{A}_{\mathsf{addr}} \mid \exists p \in \mathcal{A}_{\mathsf{path}} \text{ s.t. } (m, i, j, p) \in reach \right\}$$

- In the case of $path2set$, for every $i_1, \ldots, i_n \in \mathcal{A}_{\mathsf{addr}}$, if $p = [i_1, \ldots, i_n] \in \mathcal{A}_{\mathsf{path}}$, then

$$path2set^{\mathcal{A}}(p) = \{i_1, \ldots, i_n\}$$

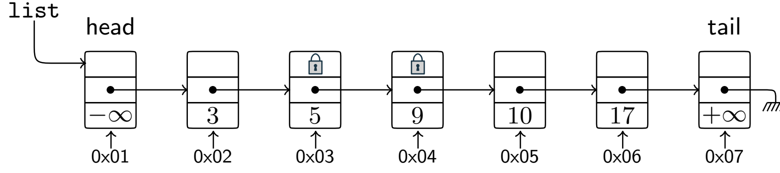
- In the case of $getp$, for each $m \in \mathcal{A}_{\mathsf{mem}}$, $p \in \mathcal{A}_{\mathsf{path}}$ and $i, j \in \mathcal{A}_{\mathsf{addr}}$:

$$getp^{\mathcal{A}}(m, i, j) \quad = \quad \begin{cases} p & \text{if } (m, i, j, p) \in reach^{\mathcal{A}} \\ \epsilon & \text{otherwise} \end{cases}$$

- In the case of $fstlock$, for each $m \in \mathcal{A}_{\mathsf{mem}}$ and $i_1, \ldots i_n \in \mathcal{A}_{\mathsf{addr}}$:

$$fstlock^{\mathcal{A}}(m, [i_1, \ldots, i_n]) \quad = \quad \begin{cases} i_k & \text{if there is } 1 \leq k \leq n \text{ such that} \\ & \text{for all } 1 \leq j < k, m[i_j].lockid = \oslash \\ & \text{and } m[i_k].lockid \neq \oslash \\ null & \text{otherwise} \end{cases}$$

Example 6.2
Consider again the following list, which was presented in Example 6.1:



Assuming that nodes are locked by thread T_1 the following interpretation $\mathcal{A}$ is in the class **TL3**:

$$\begin{aligned} \mathcal{A}_{\mathsf{addr}} &= \{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07\} \\ \mathcal{A}_{\mathsf{elem}} &= \{-\infty, 3, 5, 9, 10, 17, +\infty\} \\ \mathcal{A}_{\mathsf{tid}} &= \{T_1, \oslash\} \\ \mathcal{A}_{\mathsf{mem}} &= \{m : \mathcal{A}_{\mathsf{addr}} \to \mathcal{A}_{\mathsf{cell}}\} \end{aligned}$$

where

$$\begin{aligned} null^{\mathcal{A}} &= 0x00 \\ error^{\mathcal{A}} &= \langle -\infty, 0x00, \oslash \rangle \\ m(0x00) &= \langle -\infty, 0x00, \oslash \rangle \\ m(0x01) &= \langle -\infty, 0x02, \oslash \rangle \\ m(0x02) &= \langle 3, 0x03, \oslash \rangle \\ m(0x03) &= \langle 5, 0x04, T_1 \rangle \\ m(0x04) &= \langle 9, 0x05, T_1 \rangle \\ m(0x05) &= \langle 10, 0x06, \oslash \rangle \\ m(0x06) &= \langle 17, 0x07, \oslash \rangle \\ m(0x07) &= \langle +\infty, 0x00, \oslash \rangle \end{aligned}$$

It is easy to check that all restrictions in the class **TL3** are met.
For predicates $append$ and $reach$ we have, for instance, that:

$$\begin{array}{lcl} ([\text{0x01}, \text{0x02}], [\text{0x03}], [\text{0x01}, \text{0x02}, \text{0x03}]) & \in & append^{\mathcal{A}} \\ ([\text{0x01}], [\text{0x01}], [\text{0x01}, \text{0x01}]) & \notin & append^{\mathcal{A}} \\ ([\text{0x01}, \text{0x02}], [\text{0x03}, \text{0x04}], [\text{0x01}, \text{0x02}, \text{0x03}]) & \notin & append^{\mathcal{A}} \end{array}$$

$$\begin{array}{lcl} (m, \text{0x01}, \text{0x01}, \epsilon) & \in & reach^{\mathcal{A}} \\ (m, \text{0x01}, \text{0x01}, [\text{0x01}]) & \notin & reach^{\mathcal{A}} \\ (m, \text{0x01}, \text{0x04}, [\text{0x01}, \text{0x02}, \text{0x03}]) & \in & reach^{\mathcal{A}} \end{array}$$

For functions $addr2set$, $path2set$, $getp$ and $fstlock$ we have, for example:

$$addr2set^{\mathcal{A}}(m, \text{0x01}) = \{\text{0x01}, \text{0x02}, \text{0x03}, \text{0x04}, \text{0x05}, \text{0x06}, \text{0x07}\}$$

$$path2set^{\mathcal{A}}([\text{0x01}, \text{0x02}]) = \{\text{0x01}, \text{0x02}\}$$

$$\begin{aligned} getp^{\mathcal{A}}(m, \text{0x01}, \text{0x01}) &= \epsilon \\ getp^{\mathcal{A}}(m, \text{0x01}, \text{0x04}) &= [\text{0x01}, \text{0x02}, \text{0x03}] \\ getp^{\mathcal{A}}(m, \text{0x03}, \text{0x01}) &= \epsilon \end{aligned}$$

$$\begin{aligned} fstlock^{\mathcal{A}}(m, [\text{0x02}, \text{0x03}, \text{0x04}]) &= \text{0x03} \\ fstlock^{\mathcal{A}}(m, [\text{0x02}, \text{0x04}, \text{0x03}]) &= \text{0x04} \\ fstlock^{\mathcal{A}}(m, [\text{0x01}, \text{0x02}]) &= null \end{aligned}$$

⌟

6.2.4 Satisfiability of TL3

We now show that the satisfiability problem of quantifier-free first order formulas in TL3 is decidable. Following this result, Later we can use a TL3 decision procedure in the automatic verification of verification conditions as long as these verification conditions are quantifier-free, which is usually the case. Given a formula φ, we first transform φ into its disjunctive normal form $\varphi_1 \vee \cdots \vee \varphi_n$, where each φ_i is a conjunction of flat TL3 literals. We then consider a subset of TL3-literals, called normalized literals. All other literals can be rewritten into equivalent formulas that use only normalized literals. Considering only normalized literals aids in simplifying the theoretical developments later.

Definition 6.1 (TL3-normalized Literals).
A TL3-literal is normalized if it is a flat literal of the form:

$e_1 \neq e_2$	$a_1 \neq a_2$	
$a = null$	$c = error$	
$c = mkcell(e, a, t)$	$c = m[a]$	$m_2 = upd(m_1, a, c)$
$s = \{a\}$	$s_1 = s_2 \cup s_3$	$s_1 = s_2 \setminus s_3$
$r = \{t\}_{\mathrm{T}}$	$r_1 = r_2 \cup_{\mathrm{T}} r_3$	$r_1 = r_2 \setminus_{\mathrm{T}} r_3$
$x = \{e\}_{\mathrm{E}}$	$x_1 = x_2 \cup_{\mathrm{E}} x_3$	$x_1 = x_2 \setminus_{\mathrm{E}} x_3$
$p_1 \neq p_2$	$p = [a]$	
$s = path2set(p)$	$append(p_1, p_2, p_3)$	$\neg append(p_1, p_2, p_3)$
$s = addr2set(m, a)$	$p = getp(m, a_1, a_2)$	
$t_1 \neq t_2$	$a = fstlock\,(m, p)$	

where e, e_1 and e_2 are elem*-variables, a, a_1 and a_2 are* addr*-variables, c is a* cell*-variable, m, m_1 and m_2 are* mem*-variables, p, p_1, p_2 and p_3 are* path*-variables, s, s_1, s_2 and s_3 are* setaddr*-variables, r, r_1, r_2 and r_3 are* settid*-variables, x, x_1, x_2 and x_3 are* setelem*-variables, and t, t_1 and t_2 are* tid*-variables.* ⌟

Lemma 6.1 (TL3 Normalization):
Every non-normalized TL3-literal can be rewritten into an equivalent formula that contains only TL3-normalized literals. ⌟

Proof. We use the following equivalences to convert each non-normalized TL3-literal into formulas containing only TL3-normalized literals:

$$e = c.data \quad\leftrightarrow\quad (\exists_{\mathsf{addr}} a\ \exists_{\mathsf{tid}} t)\,[c = mkcell\,(e, a, t)] \tag{6.1}$$

$$a = c.next \quad\leftrightarrow\quad (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{tid}} t)\,[c = mkcell\,(e, a, t)] \tag{6.2}$$

$$t = c.lockid \quad\leftrightarrow\quad (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{addr}} a)\,[c = mkcell\,(e, a, t)] \tag{6.3}$$

$$c_1 = c_2.lock\,(t) \quad\leftrightarrow\quad \left(\begin{array}{l} (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{addr}} a\ \exists_{\mathsf{tid}} t_1) \\ \quad c_1 = mkcell(e, a, t) \wedge c_2 = mkcell(e, a, t_1) \end{array}\right) \tag{6.4}$$

$$c_1 = c_2.unlock \quad\leftrightarrow\quad \left(\begin{array}{l} (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{addr}} a\ \exists_{\mathsf{tid}} t_1) \\ \quad c_1 = mkcell(e, a, \oslash) \wedge c_2 = mkcell(e, a, t_1) \end{array}\right) \tag{6.5}$$

$$c_1 \neq_{\mathsf{cell}} c_2 \quad\leftrightarrow\quad \left(\begin{array}{l} (\exists_{\mathsf{elem}} e_1, e_2\ \exists_{\mathsf{addr}} a_1, a_2\ \exists_{\mathsf{tid}} t_1, t_2) \\ \quad c_1 = mkcell(e_1, a_1, t_1) \wedge \\ \quad c_2 = mkcell(e_2, a_2, t_2) \wedge \\ \quad (e_1 \neq e_2 \vee a_1 \neq a_2 \vee t_1 \neq t_2) \end{array}\right) \tag{6.6}$$

$$m_1 \neq_{\mathsf{mem}} m_2 \quad\leftrightarrow\quad (\exists_{\mathsf{addr}} a)\,(m_1[a] \neq m_2[a]) \tag{6.7}$$

$$s_1 \neq_{\mathsf{setaddr}} s_2 \quad\leftrightarrow\quad (\exists_{\mathsf{addr}} a)\,[a \in (s_1 \setminus s_2) \cup (s_2 \setminus s_1)] \tag{6.8}$$

$$s = \emptyset \quad\leftrightarrow\quad s = s \setminus s \tag{6.9}$$

$$s_3 = s_1 \cap s_2 \quad\leftrightarrow\quad s_3 = (s_1 \cup s_2) \setminus ((s_1 \setminus s_2) \cup (s_2 \setminus s_1)) \tag{6.10}$$

$$a \in s \quad\leftrightarrow\quad \{a\} \subseteq s \tag{6.11}$$

$$s_1 \subseteq s_2 \quad\leftrightarrow\quad s_2 = s_1 \cup s_2 \tag{6.12}$$

$$r_1 \neq_{\mathsf{settid}} r_2 \quad\leftrightarrow\quad (\exists_{\mathsf{tid}} t)\,[t \in_{\mathrm{T}} (r_1 \setminus_{\mathrm{T}} r_2) \cup_{\mathrm{T}} (r_2 \setminus_{\mathrm{T}} r_1)] \tag{6.13}$$

$$r = \emptyset_{\mathrm{T}} \quad\leftrightarrow\quad r = r \setminus_{\mathrm{T}} r \tag{6.14}$$

$$r_3 = r_1 \cap_{\mathrm{T}} r_2 \quad\leftrightarrow\quad r_3 = (r_1 \cup_{\mathrm{T}} r_2) \setminus_{\mathrm{T}} ((r_1 \setminus_{\mathrm{T}} r_2) \cup_{\mathrm{T}} (r_2 \setminus_{\mathrm{T}} r_1)) \tag{6.15}$$

$$t \in_{\mathrm{T}} r \quad\leftrightarrow\quad \{t\}_{\mathrm{T}} \subseteq_{\mathrm{T}} r \tag{6.16}$$

$$r_1 \subseteq_{\mathrm{T}} r_2 \quad\leftrightarrow\quad r_2 = r_1 \cup_{\mathrm{T}} r_2 \tag{6.17}$$

$$x_1 \neq_{\mathsf{setelem}} x_2 \quad\leftrightarrow\quad (\exists_{\mathsf{elem}} e)\,[e \in_{\mathrm{E}} (x_1 \setminus_{\mathrm{E}} x_2) \cup_{\mathrm{E}} (x_2 \setminus_{\mathrm{E}} x_1)] \tag{6.18}$$

$$x = \emptyset_{\mathrm{E}} \quad\leftrightarrow\quad x = x \setminus_{\mathrm{E}} x \tag{6.19}$$

$$x_3 = x_1 \cap_{\mathrm{E}} x_2 \quad\leftrightarrow\quad x_3 = (x_1 \cup_{\mathrm{E}} x_2) \setminus_{\mathrm{E}} ((x_1 \setminus_{\mathrm{E}} x_2) \cup_{\mathrm{E}} (x_2 \setminus_{\mathrm{E}} x_1)) \tag{6.20}$$

$$e \in_{\mathrm{E}} x \quad\leftrightarrow\quad \{e\}_{\mathrm{E}} \subseteq_{\mathrm{E}} x \tag{6.21}$$

$$x_1 \subseteq_{\mathrm{E}} x_2 \quad\leftrightarrow\quad x_2 = x_1 \cup_{\mathrm{E}} x_2 \tag{6.22}$$

$$p = \epsilon \quad\leftrightarrow\quad append(p, p, p) \tag{6.23}$$

$$reach(m, a_1, a_2, p) \quad\leftrightarrow\quad a_2 \in addr2set(m, a_1) \wedge p = getp(m, a_1, a_2) \tag{6.24}$$

We prove each equivalence separately. In each case we assume a model $\mathcal{A}$ of the non-normalized literal on the left hand side of the equivalence and show that $\mathcal{A}$ is a model of the corresponding formula on the right hand side of the equivalence. Similarly, we assume a model $\mathcal{B}$ of the formula on the right and prove that $\mathcal{B}$ is a model of the literal on the left.

- Equivalence (6.1). Given $\mathcal{A}$, $c^{\mathcal{A}}$ is an element of $\mathcal{A}_{\mathsf{cell}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{addr}} \times \mathcal{A}_{\mathsf{tid}}$, so there are n in $\mathcal{A}_{\mathsf{elem}}$, x in $\mathcal{A}_{\mathsf{addr}}$ and j in $\mathcal{A}_{\mathsf{tid}}$ with $c^{\mathcal{A}} = (n, x, j)$. By the restriction on the class **TL3** of models (in particular for $data$) $\mathcal{A}$ must satisfy that $e^{\mathcal{A}} = n$, and $mkcell^{\mathcal{A}}(n, x, j) = c^{\mathcal{A}}$. Hence, by taking $a^{\mathcal{A}} = x$ and $t^{\mathcal{A}} = j$ it follows that $(\exists_{\mathsf{addr}} a\ \exists_{\mathsf{tid}} t)\,[c = mkcell\,(e, a, t)]$ holds in $\mathcal{A}$.

  For the other direction, we assume $\mathcal{B}$ is a model of $(\exists_{\mathsf{addr}} a\ \exists_{\mathsf{tid}} t)\,[c = mkcell\,(e, a, t)]$ It follows, by the interpretation in $\mathcal{B}$ of $data$, that $c^{\mathcal{B}}.data^{\mathcal{B}}$ is $e^{\mathcal{B}}$ as desired.

- Equivalences (6.2) and (6.3). Analogous to equivalence (6.1).

- Equivalence (6.4). Let $\mathcal{A}$ be a model of $c_1 = c_2.lock\,(t)$. It follows that $c_1^{\mathcal{A}} = \langle e, a, t^{\mathcal{A}} \rangle$ for some $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and that $c_2^{\mathcal{A}} = \langle e, a, j \rangle$ for some j in $\mathcal{A}_{\mathsf{tid}}$. Hence, $c_1 = mkcell(e, a, t)$ and $c_2 = mkcell(e, a, j)$ hold in $\mathcal{A}$. For the other direction, every model $\mathcal{B}$ of:

  $$c_1 = mkcell(e, a, t) \ \wedge\ c_2 = mkcell(e, a, t_1)$$

  is such that $c_1^{\mathcal{B}} = \langle e^{\mathcal{B}}, a^{\mathcal{B}}, t^{\mathcal{B}} \rangle$ and $c_2^{\mathcal{B}} = \langle e^{\mathcal{B}}, a^{\mathcal{B}}, j \rangle$, for some j. If follows, by the interpretation of $lock$ that both $c_1^{\mathcal{B}}$ and $c_2^{\mathcal{B}}.lock^{\mathcal{B}}(t^{\mathcal{B}})$ are the cell $\langle e^{\mathcal{B}}, a^{\mathcal{B}}, t^{\mathcal{B}} \rangle$.

- Equivalence (6.5). Analogous to equivalence (6.4).

- Equivalence (6.6). Consider a model $\mathcal{A}$ of $c_1 \neq_{\mathsf{cell}} c_2$ and let $c_1^{\mathcal{A}} = \langle n, a, i \rangle$ and $c_2^{\mathcal{A}} = \langle m, b, j \rangle$. Since $(c_1 \neq_{\mathsf{cell}} c_2)^{\mathcal{A}}$ holds, it follows that either:

  - $n \neq m$, or
  - $a \neq b$, or
  - $i \neq j$.

  For the other direction, consider a model $\mathcal{B}$ of:

  $$c_1 = mkcell(e_1, a_1, t_1) \ \wedge \ c_2 = mkcell(e_2, a_2, t_2) \ \wedge \ (e_1 \neq e_2 \vee a_1 \neq a_2 \vee t_1 \neq t_2)$$

  In this case, $c_1^{\mathcal{B}} = \langle e_1^{\mathcal{B}}, a_1^{\mathcal{B}}, t_1^{\mathcal{B}} \rangle$ and $c_2^{\mathcal{B}} = \langle e_2^{\mathcal{B}}, a_2^{\mathcal{B}}, t_2^{\mathcal{B}} \rangle$. Then, if $e_1^{\mathcal{B}} \neq e_2^{\mathcal{B}}$, $a_1^{\mathcal{B}} \neq a_2^{\mathcal{B}}$ or $t_1^{\mathcal{B}} \neq t_2^{\mathcal{B}}$ it follows that $c_1^{\mathcal{B}} \neq_{\mathsf{cell}} c_2^{\mathcal{B}}$.

- Equivalence (6.7). This equivalence follows easily from the restriction in the class of models **TL3** of $\mathcal{A}_{\mathsf{mem}}$ to be functions from addresses to cells. The extensionality principle then implies that two functions are different precisely when they differ for some input.

- Equivalences (6.8), (6.9), (6.10), (6.11) and (6.12) are simple tautologies in the theory of sets, applied to sets of addresses (restricted in the class of models **TL3**).

- Equivalences (6.13), (6.14), (6.15), (6.16) and (6.17) are the corresponding tautologies for the theory of sets of thread identifiers.

- Equivalences (6.18), (6.19), (6.20), (6.21) and (6.22) are the corresponding tautologies for the theory of sets of elements.

- Equivalence (6.23). A model $\mathcal{A}$ of $p = \epsilon$ makes $p^{\mathcal{A}}$ be the empty path. Hence, $append(p, p, p)$ holds in $\mathcal{A}$ because **TL3** restricts the class of models to those in which $append$ concatenates the empty path with itself to resulting in the empty path. Similarly, if $append(p, p, p)$ holds in $\mathcal{B}$ it follows that $p^{\mathcal{B}}$ is the empty path (otherwise the resulting p would have repeated elements). Hence $p = \epsilon$ holds in $\mathcal{B}$.

- Equivalence (6.24). Assume that in model $\mathcal{A}$, the literal $reach(m, a_1, a_2, p)$ holds. The interpretation in **TL3** of $reach$ forces:

  $$addr2set^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}) = \{ j \in \mathcal{A}_{\mathsf{addr}} \mid \exists_{\mathsf{path}} p \text{ s.t. } (m, i, j, p) \in reach^{\mathcal{A}} \}$$

  Since $(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, p^{\mathcal{A}}) \in reach^{\mathcal{A}}$, it follows that $a_1^{\mathcal{A}} \in addr2set^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}})$ as desired. Similarly, $getp^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}) = p^{\mathcal{A}}$. For the other direction, let $\mathcal{B}$ be a model of

  $$a_2 \in addr2set(m, a_1) \wedge p = getp(m, a_1, a_2)$$

  The second conjunct, $(p = getp(m, a_1, a_2))$, implies that $(m^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, p^{\mathcal{B}}) \in reach^{\mathcal{B}}$ and then $reach(m, a_1, a_2, p)$ holds in $\mathcal{B}$.

All (6.1)–(6.24) are valid TL3 equivalences. □

Orienting the (6.1)–(6.24) equivalences from left to right allows to eliminate non-flat literals from a given formula, resulting in a formula that only contains normalized literals. Note also, that all quantifiers introduced in these equivalences are existential quantifiers, which can be pushed (with renaming to avoid capturing if necessary) to the front of the formula. Hence, a quantifier-free formula (which is implicitly existentially quantified) results into a quantifier-free formula after the rewriting step.

Example 6.3
Consider once again the list presented in Example 6.1 and consider the non-normalized formula:

$$\varphi_{\mathsf{nonNorm}} \stackrel{\text{def}}{=} reach(m, head, tail, getp(m, head, tail)) \ \wedge \ m[tail].next = null$$

This formula states that in the list, the cell pointed by $tail$ is reachable from $head$ following the path obtained by $getp$. Additionally, this formula states that the cell pointed by $tail$ is followed by $null$. If we rewrite this formula using only normalized literals, we have:

$$\begin{array}{rcll} \{tail\} & \in & s & \wedge \\ s & = & addr2set(m, head) & \wedge \\ p & = & getp(m, head, tail) & \wedge \\ c & = & m[tail] & \wedge \\ c & = & mkcell(e, a, t) & \wedge \\ a & = & null & \end{array}$$ ⌟

The formula obtained after normalization can be converted into its disjunctive normal form, obtaining the following result.

Lemma 6.2 (Normalized Literals):
Every TL3-formula is equivalent to a disjunction of conjunctions of normalized TL3-literals. ⌟

Proof. Take the TL3-formula and normalize all its literals, transforming all non-normalized literals using Lemma 6.1. Finally, from the resulting formula containing only normalized literals, compute the disjunctive normal. □

6.3 Decidability of TL3

The theory TLL enjoys of the finite model property, as shown in [178]. We now prove that TL3 enjoys the bounded model property presented in Definition 2.2 with respect to domains elem, addr and tid. Moreover, we will show how to compute for a given formula φ, a (polynomial) bound on the size of $\mathcal{A}_{\mathsf{elem}}$, $\mathcal{A}_{\mathsf{addr}}$ and $\mathcal{A}$tid of a sufficiently large model $\mathcal{A}$. In other words, if there is no model within the bounds, then φ is unsatisfiable. Note that a bound on the domain of the sorts elem, addr and tid is enough to also obtain bounds on the domains of the remaining sorts (cell, mem, path, setaddr, setelem and settid) because the domains of these latter sorts are constructed

from the domains of elem, addr and tid due to the restrictions imposed in the class of models **TL3**. These bounds imply that TL3 is decidable because one can enumerate all Σ_{TL3}-structures up to the cardinality given by the bound of the finite model theorem, and check whether each of the structures is indeed a model of the given formula.

Consider an arbitrary TL3-interpretation $\mathcal{A}$ satisfying a conjunction of normalized TL3-literals Γ. We will construct domains $\mathcal{B}_{\mathsf{elem}}$, $\mathcal{B}_{\mathsf{addr}}$ and $\mathcal{B}_{\mathsf{tid}}$ with bounded cardinalities (with the bound depending on Γ), and then construct a finite interpretation $\mathcal{B}$ which also satisfies Γ.

6.3.1 Auxiliary Functions for Model Transformation

Before proceeding with the proof that TL3 enjoys the bounded model property, we first define some auxiliary functions.

Definition 6.2 (TL3 Many Jumps).
Given a memory m and an address a, we use the following notation for $s \geq 1$:

$$m^{[s]}(a).next \;=\; \begin{cases} m(a).next & \text{if } s = 1 \\ m(m^{[s-1]}(a).next).next & \text{if } s > 1 \end{cases}$$

⌟

We start by defining the function *first*, whose intended meaning is to give the first *relevant* element of the domain of addresses that is necessary to preserve the valuation of all functions and predicates. Later, all irrelevant elements will be removed from the large domain to obtain the bounded domain. Let $X \subseteq \mathcal{A}_{\mathsf{addr}}$, $m : \mathcal{A}_{\mathsf{addr}} \to \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{addr}} \times \mathcal{A}_{\mathsf{tid}}$ and $a \in X$ be an address. The function $\mathit{first}(m, a, X)$ is defined by

$$\mathit{first}(m,a,X) = \begin{cases} \mathit{null} & \text{if for all } r \geq 1,\ m^{[r]}(a).next(l) \notin X \\ \\ m^{[s]}\left(a\right).next(l) & \text{if for some } s \geq 1,\ m^{[s]}(a).next(l) \in X \\ & \quad \text{and for all } r < s,\ m^{[r]}(a).next(l) \notin X \end{cases}$$

Basically, given a set of addresses X, function *first* chooses the next address in X that can be reached from a given address by repeatedly following the $next$ pointer. We will later filter out unnecessary intermediate nodes and use *first* to bypass properly the removed nodes, preserving the important connectivity properties.

Lemma 6.3 (Function *first*):
Let $X \subseteq \mathcal{A}_{\mathsf{addr}}$, m_1 and $m_2 : \mathcal{A}_{\mathsf{addr}} \to \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{addr}} \times \mathcal{A}_{\mathsf{tid}}$, a_1 and $a_2 \in X$, $c \in \mathcal{A}_{\mathsf{elem}} \times X \times \mathcal{A}_{\mathsf{tid}}$ and $a_1 \neq a_2$. Then:

(a) If $m_1(a_1).next \in X$ then $\mathit{first}(m_1, a_1, X) = m_1(a_1).next$

(b) If $m_1 = \mathit{upd}(m_2, a_1, c)$ then $\mathit{first}(m_1, a_2, X) = \mathit{first}(m_2, a_2, X)$ ⌟

Proof. We proof (a) and (b) separately.

(a) Immediate from the definition of first.

(b) Let $m_1 = \mathit{upd}(m_2, a_1, c)$. We consider two possible cases:

(1) $m_1^{[r]}(a_2).\mathit{next} \notin X$ for all $r \geq 1$. By induction we can show that $m_1^{[r]}(a_2) = m_2^{[r]}(a_2)$ for each $r \geq 1$. It follows that $\mathit{first}(m_1, a_2, X) = \mathit{null} = \mathit{first}(m_2, a_2, X)$.

(2) $m_1^{[s]}(a_2).\mathit{next} \in X$ for some $s \geq 1$. In this case, assume without loss of generality that $\mathit{first}(m_1, a_2, X) = m_1^{[s]}(a_2).\mathit{next}$. By induction it can be shown that $m_1^{[r]}(a_2) = m_2^{[r]}(a_2)$ for each $1 \leq r < s$. It follows that $\mathit{first}(m_1, a_2, X) = m_1^{[s]}(a_2).\mathit{next} = m_2^{[s]}(a_2).\mathit{next} = \mathit{first}(m_2, a_2, X)$. □

Next, we define the $\mathit{compress}$ function which, given a path p and a set X of addresses, returns the path obtained from p by removing all the addresses that do not belong to X.

$$\mathit{compress}([i_1, \dots, i_n], X) = \begin{cases} \epsilon & \text{if } n = 0 \\ [i_1] \circ \mathit{compress}\left([i_2, \dots, i_n], X\right) & \text{if } n > 0 \text{ and } i_1 \in X \\ \mathit{compress}\left([i_2, \dots, i_n], X\right) & \text{otherwise} \end{cases}$$

In the definition above, we use $\circ$ to denote the concatenation of paths. That is, $[i_1, \dots, i_k] \circ [j_1, \dots, j_l] = [i_1, \dots, i_k, j_1, \dots, j_l]$.

Lemma 6.4 (Function $\mathit{compress}$):
Let a be an address, let X be a set of addresses and let p_1 and p_2 be paths. Then:

(a) if $a \in X$ then $\mathit{compress}([a], X) = [a]$

(b) $\mathit{path2set}(p_1) \cap X = \mathit{path2set}(\mathit{compress}(p_1, X))$

(c) If $\mathit{path2set}(p_1) \cap \mathit{path2set}(p_2) = \emptyset$, then

$$\mathit{compress}(p_1 \circ p_2, X) = \mathit{compress}(p_1, X) \circ \mathit{compress}(p_2, X)$$ ⌟

Proof. Immediate from definition of function $\mathit{compress}$. □

The third auxiliary function we need is diseq, already introduced in [178], which provides a set of addresses that witness the inequality of two given paths:

$$\mathit{diseq}([i_1, \dots, i_n], [j_1, \dots, j_m]) = \begin{cases} \emptyset & \text{if } n = m = 0 \\ \{i_1\} & \text{if } n > 0 \text{ and } m = 0 \\ \{j_1\} & \text{if } n = 0 \text{ and } m > 0 \\ \{i_1, j_1\} & \text{if } n, m > 0 \text{ and } i_1 \neq j_1 \\ \mathit{diseq}([i_2, \dots, i_n], [j_2, \dots, j_m]) & \text{otherwise} \end{cases}$$

Note that the set that diseq returns has at most two elements.

Lemma 6.5 (Function $diseq$):
For all paths p_1 and p_2 and for any set of addresses X, if $p_1 \neq p_2$ and $diseq(p_1, p_2) \subseteq X$, then $compress(p_1, X) \neq compress(p_2, X)$ ⌟

Proof. Immediate from definition of $diseq$ and $compress$. □

Finally, the last function is $common$, which outputs an element common to two paths, that is, an element that witnesses that $(path2set(p) \cap path2set(q) \neq \emptyset)$:

$$common([i_1, \dots, i_n], p) = \begin{cases} \emptyset & \text{if } n = 0 \\ \{i_1\} & \text{if } n > 0 \text{ and } i_1 \in path2set(p) \\ common([i_2, \dots, i_n], p) & \text{otherwise} \end{cases}$$

6.3.2 A Bounded Model Theorem for TL3

We are now ready to establish the main result of this chapter. Before doing so, we will proof an auxiliary lemma.

Lemma 6.6 (TL3 Reachability Preservation):
Let $\mathcal{A}$ and $\mathcal{B}$ be two TL3 models such that:

- *For each $e \in V_{\mathsf{elem}}$, $e^{\mathcal{A}} = e^{\mathcal{B}}$.*
- *For each $a \in V_{\mathsf{addr}}$, $a^{\mathcal{A}} = a^{\mathcal{B}}$.*
- *For each $t \in V_{\mathsf{tid}}$, $t^{\mathcal{A}} = t^{\mathcal{B}}$.*
- *For each $m \in V_{\mathsf{mem}}$ and $a \in \mathcal{B}_{\mathsf{addr}}$, $m^{\mathcal{B}}(a) = \left(m^{\mathcal{A}}(a).data^{\mathcal{A}}, first(m^{\mathcal{A}}, a, \mathcal{B}_{\mathsf{addr}}), m^{\mathcal{A}}(a).lockid^{\mathcal{A}}\right)$.*

Then, for any $a, b \in V_{\mathsf{addr}}$ and $m \in V_{\mathsf{mem}}$:

$$\text{if} \quad m^{\mathcal{B}}(a^{\mathcal{B}}).next^{\mathcal{B}} = b^{\mathcal{B}} \quad \text{then} \quad m^{\mathcal{A}[s]}(a^{\mathcal{A}}).next^{\mathcal{A}} = b^{\mathcal{A}}, \text{ for some } s \geq 1$$ ⌟

Proof. According to the lemma assumptions we have that $a^{\mathcal{A}} = a^{\mathcal{B}}$, $b^{\mathcal{A}} = b^{\mathcal{B}}$ and $m^{\mathcal{B}}(a^{\mathcal{B}}).next^{\mathcal{B}} = first(m^{\mathcal{A}}, a^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$. Following these equalities we have that

$$\begin{aligned} b^{\mathcal{A}} &= b^{\mathcal{B}} \\ &= m^{\mathcal{B}}(a^{\mathcal{B}}).next^{\mathcal{B}} \\ &= first(m^{\mathcal{A}}, a^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) \end{aligned}$$

Assume that for all $r \geq 1$, $m^{\mathcal{A}}(a^{\mathcal{A}}).next^{\mathcal{A}} \notin \mathcal{B}_{\mathsf{addr}}$. This is a contradiction, as $first(m^{\mathcal{A}}, a^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) = b^{\mathcal{A}} = b^{\mathcal{B}} \in \mathcal{B}_{\mathsf{addr}}$. Then is must be that, $m^{\mathcal{A}}(a^{\mathcal{A}}).next^{\mathcal{A}} \in \mathcal{B}_{\mathsf{addr}}$ and moreover, because of the definition of $first$, $first(m^{\mathcal{A}}, a^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) = m^{\mathcal{A}[s]}(a^{\mathcal{A}}).next^{\mathcal{A}}$ for some $s \geq 1$. □

Now, we are ready for the main result of this chapter. The following theorem establishes the existence of a small model with bounded domains whenever a TL3 formula is satisfiable.

Theorem 6.1 (TL3 Bounded Model Property):
Let Γ be a conjunction of normalized TL3-literals. Let $\mathsf{E} = |V_{\mathsf{elem}}(\Gamma)|$, $\mathsf{A} = |V_{\mathsf{addr}}(\Gamma)|$, $\mathsf{M} = |V_{\mathsf{mem}}(\Gamma)|$, $\mathsf{P} = |V_{\mathsf{path}}(\Gamma)|$ and $\mathsf{T} = |V_{\mathsf{tid}}(\Gamma)|$. Then the following are equivalent:

1. *Γ is TL3-satisfiable (i.e., Γ is true in some TL3 interpretation $\mathcal{A}$).*

2. *Γ is true in some TL3 interpretation $\mathcal{B}$ such that*

$$
\begin{aligned}
|\mathcal{B}_{\mathsf{addr}}| &\leq \mathsf{A} + 1 + \mathsf{M} \times \mathsf{A} + 2 \times \mathsf{P}^2 + 2 \times \mathsf{P}^3 \\
|\mathcal{B}_{\mathsf{tid}}| &\leq \mathsf{T} + \mathsf{M} \times |\mathcal{B}_{\mathsf{addr}}| + 1 \\
|\mathcal{B}_{\mathsf{elem}}| &\leq \mathsf{E} + \mathsf{M} \times |\mathcal{B}_{\mathsf{addr}}|
\end{aligned}
$$

⌟

Proof. ($2 \to 1$) is immediate, since $\mathcal{B}$ is a model of Γ. We now show the other direction: ($1 \to 2$). We first construct the small model candidate $\mathcal{B}$ from $\mathcal{A}$ and then show that all literals hold in $\mathcal{B}$ precisely when they hold in $\mathcal{A}$. Let $\mathcal{A}$ be a TL3-interpretation satisfying Γ. The domain $\mathcal{B}_{\mathsf{addr}}$ is defined as follows:

$$
\begin{aligned}
\mathcal{B}_{\mathsf{addr}} = \; & V^{\mathcal{A}}_{\mathsf{addr}} \cup \left\{ null^{\mathcal{A}} \right\} \cup && (6.25) \\
& \left\{ m^{\mathcal{A}}(v^{\mathcal{A}}).next^{\mathcal{A}} \mid m \in V_{\mathsf{mem}} \text{ and } v \in V_{\mathsf{addr}} \right\} \cup && (6.26) \\
& \left\{ diseq(p^{\mathcal{A}}, q^{\mathcal{A}}) \mid \text{ the literal } p \neq q \text{ is in } \Gamma \right\} \cup && (6.27) \\
& \left\{ common(p_1{}^{\mathcal{A}}, p_2{}^{\mathcal{A}}) \mid \text{ the literal } \neg append(p_1, p_2, p_3) \text{ is in } \Gamma \text{ and } \right. \\
& \qquad \left. path2set^{\mathcal{A}}(p_1{}^{\mathcal{A}}) \cap path2set^{\mathcal{A}}(p_2{}^{\mathcal{A}}) \neq \emptyset \right\} \cup && (6.28) \\
& \left\{ common(p_1{}^{\mathcal{A}} \circ p_2{}^{\mathcal{A}}, p_3{}^{\mathcal{A}}) \mid \text{ the literal } \neg append(p_1, p_2, p_3) \text{ is in } \Gamma \text{ and } \right. \\
& \qquad \left. path2set^{\mathcal{A}}(p_1{}^{\mathcal{A}}) \cap path2set^{\mathcal{A}}(p_2{}^{\mathcal{A}}) = \emptyset \right\} && (6.29)
\end{aligned}
$$

Essentially, $\mathcal{B}_{\mathsf{addr}}$ is a subset of $\mathcal{A}_{\mathsf{addr}}$ in which the following addresses are preserved:

- Because of (6.25), all the address in the domain $\mathcal{A}_{\mathsf{addr}}$ that correspond to variables are preserved in $\mathcal{B}_{\mathsf{addr}}$. There are at most A of these addresses. The element modeling $null$ in $\mathcal{A}_{\mathsf{addr}}$ is also kept.
- In (6.26) we ensure that for each variable of sort address and each variable of sort memory, the next address is also preserved. There are at most ($\mathsf{M} \times \mathsf{A}$) of these addresses.
- Then, (6.27) ensures that one or two addresses (as returned by $diseq$) are kept for every literal $p \neq q$. Since there are at most P paths, there are at most (P^2) literals $p \neq q$, which results in preserving at most ($2 \times \mathsf{P}^2$) addresses.
- Finally, there are two reasons that explain why a literal ($\neg append(p_1, p_2, p_3)$) holds in $\mathcal{A}$. The first case is when paths p_1 and p_2 share addresses. In this case, their concatenation is not a legal path, because legal paths cannot contain repeated addresses, so (6.28) keeps a

common address present in both paths to witness this sharing. In the second case, paths p_1 and p_2 do not share any address, so their concatenation is a legal path, but this resulting path is not equal to p_3. In this case, (6.29) keeps one or two addresses as return by $diseq$ to witness this difference. Since there are P paths in Γ there are at most (P^3) literals of the form $\neg append(p_1, p_2, p_3)$ which result in at most $(2 \times \mathsf{P}^3)$ variables preserved.

The domains for sorts tid and elem are as follows:

$$\mathcal{B}_{\mathsf{tid}} = V^{\mathcal{A}}_{\mathsf{tid}} \cup \left\{ m^{\mathcal{A}}(a).lockid^{\mathcal{A}} \mid m \in V_{\mathsf{mem}} \text{ and } a \in \mathcal{B}_{\mathsf{addr}} \right\} \cup \left\{ \oslash \right\}$$

$$\mathcal{B}_{\mathsf{elem}} = V^{\mathcal{A}}_{\mathsf{elem}} \cup \left\{ m^{\mathcal{A}}(a).data^{\mathcal{A}} \mid m \in V_{\mathsf{mem}} \text{ and } a \in \mathcal{B}_{\mathsf{addr}} \right\}$$

Again, the domain $\mathcal{B}_{\mathsf{tid}}$ is built by pruning $\mathcal{A}_{\mathsf{tid}}$. The interpretations of all variables of sort tid in the formula are kept, and the invalid thread identifier $\oslash$ is also kept. Finally, the domain $\mathcal{B}_{\mathsf{tid}}$ also contains the thread identifiers that appear in fields of cells that are obtained from preserved address kept in $\mathcal{B}_{\mathsf{addr}}$ by memory variables. Similarly, $\mathcal{B}_{\mathsf{elem}}$ is built from $\mathcal{A}_{\mathsf{elem}}$ by keeping interpretations of variables of sort elem and fields of preserved cells.

The domains $\mathcal{B}_{\mathsf{addr}}$, $\mathcal{B}_{\mathsf{tid}}$ and $\mathcal{B}_{\mathsf{elem}}$ satisfy the cardinality constraints expressed in the statement of Theorem 6.1. The interpretations of the rest of domains are obtained using the restrictions of **TL3**, shown in Section 6.2:

- $\mathcal{B}_{\mathsf{cell}} = \mathcal{B}_{\mathsf{elem}} \times \mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{\mathsf{tid}}$.
- $\mathcal{B}_{\mathsf{mem}} = \mathcal{B}_{\mathsf{cell}}^{\mathcal{B}_{\mathsf{addr}}}$.
- $\mathcal{B}_{\mathsf{path}}$ is the set of all finite sequences of (pairwise) distinct elements of $\mathcal{B}_{\mathsf{addr}}$.
- $\mathcal{B}_{\mathsf{setaddr}}$ is the power-set of $\mathcal{B}_{\mathsf{addr}}$.
- $\mathcal{B}_{\mathsf{settid}}$ is the power-set of $\mathcal{B}_{\mathsf{tid}}$.
- $\mathcal{B}_{\mathsf{setelem}}$ is the power-set of $\mathcal{B}_{\mathsf{elem}}$.

All these domains are finite, because $\mathcal{B}_{\mathsf{addr}}$, $\mathcal{B}_{\mathsf{tid}}$ and $\mathcal{B}_{\mathsf{elem}}$ are finite.

We are left to show the interpretation of all variables and function symbols, and to prove that

$\mathcal{B}$ is a model of Γ. The interpretation of variables and symbols in $\mathcal{B}$ is:

$$\begin{array}{rcll}
error^{\mathcal{B}} & = & error^{\mathcal{A}} & \\
null^{\mathcal{B}} & = & null^{\mathcal{A}} & \\
e^{\mathcal{B}} & = & e^{\mathcal{A}} & \text{for each } e \in V_{\mathsf{elem}} \\
v^{\mathcal{B}} & = & v^{\mathcal{A}} & \text{for each } v \in V_{\mathsf{addr}} \\
c^{\mathcal{B}} & = & c^{\mathcal{A}} & \text{for each } c \in V_{\mathsf{cell}} \\
t^{\mathcal{B}} & = & t^{\mathcal{A}} & \text{for each } t \in V_{\mathsf{tid}} \\
m^{\mathcal{B}}(a) & = & \left\langle m^{\mathcal{A}}(a).data^{\mathcal{A}}, \mathit{first}(m^{\mathcal{A}}, a, \mathcal{B}_{\mathsf{addr}}), m^{\mathcal{A}}(a).lockid^{\mathcal{A}} \right\rangle & \text{for each } m \in V_{\mathsf{mem}},\, a \in \mathcal{B}_{\mathsf{addr}} \\
s^{\mathcal{B}} & = & s^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}} & \text{for each } s \in V_{\mathsf{setaddr}} \\
r^{\mathcal{B}} & = & r^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{tid}} & \text{for each } r \in V_{\mathsf{settid}} \\
x^{\mathcal{B}} & = & x^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{elem}} & \text{for each } x \in V_{\mathsf{setelem}} \\
p^{\mathcal{B}} & = & compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) & \text{for each } p \in V_{\mathsf{path}}
\end{array}$$

The interpretation of $error$ and $null$ and of all variables is preserved from $\mathcal{A}$. Sets are pruned to contain only elements kept in $\mathcal{B}_{\mathsf{addr}}$, $\mathcal{B}_{\mathsf{tid}}$ and $\mathcal{B}_{\mathsf{elem}}$. The interpretation of path variables is the path reduced to contain only elements in $\mathcal{B}_{\mathsf{addr}}$, using the $compress$ function. Finally, memory variables are interpreted as maps from address in $\mathcal{B}_{\mathsf{addr}}$ into cells, where the next field is modified to be in $\mathcal{B}_{\mathsf{addr}}$, using the function first. It is easy to check that $\mathcal{B}$ is an interpretation of Γ and hence a candidate model of Γ

It remains to be seen that $\mathcal{B}$ satisfies all literals in Γ assuming that $\mathcal{A}$ does, concluding that $\mathcal{B}$ is indeed a model of Γ. We reason by cases considering all possible literals:

Literals of the form $e_1 \neq e_2$, $a_1 \neq a_2$ and $t_1 \neq t_2$:

Immediate, because these are preserved from $\mathcal{A}$ into $\mathcal{B}$. For instance, consider the case of the literal $t_1 \neq t_2$. Since the valuation for variables is preserved, $t_1^{\mathcal{B}} = t_1^{\mathcal{A}}$ and $t_2^{\mathcal{B}} = t_2^{\mathcal{A}}$. Hence $t_1 \neq t_2$ holds in $\mathcal{B}$ because it holds in $\mathcal{A}$.

Literals of the form $a = null$ and $c = error$:

Immediate, following a reasoning similar to the previous case.

Literals of the form $c = mkcell(e, a, t)$:

In this case

$$\begin{array}{rcl}
c^{\mathcal{B}} & = & c^{\mathcal{A}} \\
 & = & mkcell^{\mathcal{A}}(e, a, t) \\
 & = & \langle e^{\mathcal{A}}, a^{\mathcal{A}}, t^{\mathcal{A}} \rangle \\
 & = & \langle e^{\mathcal{B}}, a^{\mathcal{B}}, t^{\mathcal{B}} \rangle \\
 & = & mkcell^{\mathcal{B}}(e, a, t)
\end{array}$$

Literals of the form $c = m[a]$:

In this case we have that:

$$
\begin{aligned}
\Big(m[a]\Big)^{\mathcal{B}} &= m^{\mathcal{B}}(a^{\mathcal{B}}) \\
&= m^{\mathcal{B}}(a^{\mathcal{A}}) \\
&= \Big(m^{\mathcal{A}}(a^{\mathcal{A}}).data^{\mathcal{A}}, \mathit{first}(m^{\mathcal{A}}, a^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}), m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}\Big) \\
&= \Big(m^{\mathcal{A}}(a^{\mathcal{A}}).data^{\mathcal{A}}, m^{\mathcal{A}}(a^{\mathcal{A}}).next^{\mathcal{A}}, m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}\Big) \qquad (6.30) \\
&= m^{\mathcal{A}}(a^{\mathcal{A}}) \\
&= c^{\mathcal{A}} \\
&= c^{\mathcal{B}}
\end{aligned}
$$

where (6.30) is justified by Lemma 6.3(a) and (6.26).

Literals of the form $m = upd(\tilde{m}, a, c)$**:**
We want to prove that $m^{\mathcal{B}} = \tilde{m}^{\mathcal{B}}_{a^{\mathcal{B}} \mapsto c^{\mathcal{B}}}$. Consider two cases:

- $m(a)$: In this case,

$$
m^{\mathcal{B}}(a^{\mathcal{B}}) \;=\; m^{\mathcal{A}}(a^{\mathcal{A}}) \;=\; c^{\mathcal{A}} \;=\; c^{\mathcal{B}}
$$

- $m(d)$ for $d \neq a$: In this case,

$$
\begin{aligned}
m^{\mathcal{B}}(d) &= \Big(m^{\mathcal{A}}(d).data^{\mathcal{A}}, \mathit{first}(m^{\mathcal{A}}, d, \mathcal{B}_{\mathsf{addr}}), m^{\mathcal{A}}(d).lockid^{\mathcal{A}}\Big) \\
&= \Big(\tilde{m}^{\mathcal{A}}(d).data^{\mathcal{A}}, \mathit{first}(\tilde{m}^{\mathcal{A}}, d, \mathcal{B}_{\mathsf{addr}}), \tilde{m}^{\mathcal{A}}(d).lockid^{\mathcal{A}}\Big) \qquad (6.31) \\
&= \tilde{m}^{\mathcal{B}}(d)
\end{aligned}
$$

Where (6.31) is justified by Lemma 6.3(b)

Literals of the form $s = \{a\}$**:**

$$
s^{\mathcal{B}} \;=\; s^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}} \;=\; \{a^{\mathcal{A}}\} \cap \mathcal{B}_{\mathsf{addr}} \;=\; \{a^{\mathcal{B}}\} \cap \mathcal{B}_{\mathsf{addr}} \;=\; \{a^{\mathcal{B}}\}
$$

Literals of the form $r = \{t\}_{\mathtt{T}}$ **and** $x = \{e\}_{\mathtt{E}}$**:**
Analogous to the previous case.

Literals of the form $s_1 = s_2 \cup s_3$**:**

$$
\begin{aligned}
s_1^{\mathcal{B}} &= s_1^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}} \\
&= \big(s_2^{\mathcal{A}} \cup s_3^{\mathcal{A}}\big) \cap \mathcal{B}_{\mathsf{addr}} \\
&= \big(s_2^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}}\big) \cup \big(s_3^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}}\big) \\
&= s_2^{\mathcal{B}} \cup s_3^{\mathcal{B}}
\end{aligned}
$$

Literals of the form $r_1 = r_2 \cup_{\mathtt{T}} r_3$ **and** $x_1 = x_2 \cup_{\mathtt{E}} x_3$**:**
Analogous to the previous case.

Literals of the form $s_1 = s_2 \setminus s_3$:

$$
\begin{aligned}
s_1^{\mathcal{B}} &= s_1^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}} \\
&= \left(s_2^{\mathcal{A}} \setminus s_3^{\mathcal{A}}\right) \cap \mathcal{B}_{\mathsf{addr}} \\
&= \left(s_2^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}}\right) \setminus \left(s_3^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}}\right) \\
&= s_2^{\mathcal{B}} \setminus s_3^{\mathcal{B}}
\end{aligned}
$$

Literals of the form $r_1 = r_2 \setminus_{\mathrm{T}} r_3$ and $x_1 = x_2 \setminus_{\mathrm{E}} x_3$:

Analogous to the previous case.

Literals of the form $p_1 \neq p_2$:

Assume that literal $p_1 \neq p_2$ holds in $\mathcal{A}$. Hence, due to the way we construct $\mathcal{B}_{\mathsf{addr}}$ we have that $diseq(p_1{}^{\mathcal{A}}) \in \mathcal{B}_{\mathsf{addr}}$. Then, by Lemma 6.5, we have that $compress(p_1{}^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) \neq compress(p_2{}^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$. Finally, because of the interpretation of paths in $\mathcal{B}$, we know that for all variables of sort path, $p^{\mathcal{B}} = compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$. Consequently, $p_1{}^{\mathcal{B}} \neq p_2{}^{\mathcal{B}}$.

Literals of the form $p = [a]$:

By Lemma 6.4(a), we have that

$$
p^{\mathcal{B}} = compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) = compress([a^{\mathcal{A}}], \mathcal{B}_{\mathsf{addr}}) = [a^{\mathcal{B}}]
$$

Literals of the form $s = path2set(p)$:

By Lemma 6.4(b), we have that

$$
\begin{aligned}
s^{\mathcal{B}} &= s^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}} \\
&= path2set^{\mathcal{A}}(p^{\mathcal{A}}) \cap \mathcal{B}_{\mathsf{addr}} \\
&= path2set^{\mathcal{A}}(compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) \\
&= path2set^{\mathcal{A}}(p^{\mathcal{B}})
\end{aligned}
$$

Literals of the form $append(p_1, p_2, p_3)$:

Assume that $(p_1^{\mathcal{A}}, p_2^{\mathcal{A}}, p_3^{\mathcal{A}}) \in append^{\mathcal{A}}$. Then $p_1^{\mathcal{A}} \cap p_2^{\mathcal{A}} = \emptyset$ and $p_1^{\mathcal{A}} \circ p_2^{\mathcal{A}} = p_3^{\mathcal{A}}$. By Lemma 6.4(c) this implies that:

- $path2set(compress(p_1^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) \cap path2set(compress(p_2^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) = \emptyset$, and
- $compress(p_1^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) \circ compress(p_2^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) = compress(p_3^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$

This means that $path2set(p_1^{\mathcal{B}}) \cap path2set(p_2^{\mathcal{B}}) = \emptyset$ and $p_1^{\mathcal{B}} \circ p_2^{\mathcal{B}} = p_3^{\mathcal{B}}$. Consequently $(p_1^{\mathcal{B}}, p_2^{\mathcal{B}}, p_3^{\mathcal{B}}) \in append^{\mathcal{B}}$.

Literals of the form $\neg append(p_1, p_2, p_3)$:

Assume that $(p_1^{\mathcal{A}}, p_2^{\mathcal{A}}, p_3^{\mathcal{A}}) \notin append^{\mathcal{A}}$. If $path2set(p_1^{\mathcal{A}}) \cap path2set(p_2^{\mathcal{A}}) \neq \emptyset$ then we have that $common(p_1^{\mathcal{A}}, p_2^{\mathcal{A}}) \neq \emptyset$ which implies that $path2set(p_1^{\mathcal{B}}) \cap path2set(p_2^{\mathcal{B}}) \neq \emptyset$ and thus $(p_1^{\mathcal{B}}, p_2^{\mathcal{B}}, p_3^{\mathcal{B}}) \notin append^{\mathcal{B}}$. On the other hand, if $path2set(p_1^{\mathcal{A}}) \cap path2set(p_2^{\mathcal{A}}) = \emptyset$, then $diseq(p_1^{\mathcal{A}} \circ p_2^{\mathcal{A}}, p_3^{\mathcal{A}}) \neq \emptyset$, and thus $diseq(p_1^{\mathcal{B}} \circ p_2^{\mathcal{B}}, p_3^{\mathcal{B}}) \neq \emptyset$ and then $(p_1^{\mathcal{B}}, p_2^{\mathcal{B}}, p_3^{\mathcal{B}}) \notin append^{\mathcal{B}}$.

Literals of the form $s = addr2set(m, a)$**:**

Let $x = a^{\mathcal{B}} = a^{\mathcal{A}}$. Then,

$$\begin{aligned} s^{\mathcal{B}} &= s^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}} \\ &= \left\{ y \in \mathcal{A}_{\mathsf{addr}} \mid \text{for some } p \in \mathcal{A}_{\mathsf{path}},\ (m^{\mathcal{A}}, x, y, p) \in reach^{\mathcal{A}} \right\} \cap \mathcal{B}_{\mathsf{addr}} \\ &= \left\{ y \in \mathcal{B}_{\mathsf{addr}} \mid \text{for some } p \in \mathcal{A}_{\mathsf{path}},\ (m^{\mathcal{A}}, x, y, p) \in reach^{\mathcal{A}} \right\} \\ &= \left\{ y \in \mathcal{B}_{\mathsf{addr}} \mid \text{for some } p \in \mathcal{B}_{\mathsf{path}},\ (m^{\mathcal{B}}, x, y, p) \in reach^{\mathcal{B}} \right\} \end{aligned}$$

It just remains to see that the last equality holds. Let,

$$\begin{aligned} S_{\mathcal{B}} &= \left\{ y \in \mathcal{B}_{\mathsf{addr}} \mid \text{for some } p \in \mathcal{B}_{\mathsf{path}},\ (m^{\mathcal{B}}, x, y, p) \in reach^{\mathcal{B}} \right\}, \text{ and} \\ S_{\mathcal{A}} &= \left\{ y \in \mathcal{B}_{\mathsf{addr}} \mid \text{for some } p \in \mathcal{A}_{\mathsf{path}},\ (m^{\mathcal{A}}, x, y, p) \in reach^{\mathcal{A}} \right\} \end{aligned}$$

We now show that $S_{\mathcal{A}} = S_{\mathcal{B}}$ by showing that $S_{\mathcal{A}} \subseteq S_{\mathcal{B}}$ and $S_{\mathcal{B}} \subseteq S_{\mathcal{A}}$:

(1) We first show that $S_{\mathcal{A}} \subseteq S_{\mathcal{B}}$. Let $y \in S_{\mathcal{A}}$. Then, there exists $p \in \mathcal{A}_{\mathsf{path}}$ such that $(m^{\mathcal{A}}, x, y, p) \in reach^{\mathcal{A}}$ and by definition of $reach$ there are two possible cases.

- If $p = \epsilon$ and $x = y$, then $(m^{\mathcal{B}}, x, y, \epsilon^{\mathcal{B}}) \in reach^{\mathcal{B}}$ and therefore $y \in S_{\mathcal{B}}$.
- Otherwise, there exist $a_1, \ldots, a_n \in \mathcal{A}_{\mathsf{addr}}$ s.t.,

$$\begin{array}{llll} i) & p = [a_1, \ldots, a_n] & iii) & m^{\mathcal{A}}(a_r).next^{\mathcal{A}} = a_{r+1}, \text{for } 1 \leq r < n \\ ii) & x = a_1 & iv) & m^{\mathcal{A}}(a_n).next^{\mathcal{A}} = y \end{array}$$

  We proceed by induction on n.

  - If $n = 1$, then $p = [a_1]$ and since $a_1 = x$, $m^{\mathcal{A}}(a_1).next^{\mathcal{A}} = y$ and $y \in \mathcal{B}_{\mathsf{addr}}$, we have that $y \in S_{\mathcal{B}}$.
  - If $n > 1$, then let $a_i = first(m^{\mathcal{A}}, x, \mathcal{B}_{\mathsf{addr}})$. Then, since $p = a_1, \ldots, a_i, a_{i+1}, \ldots, a_n$ we have that considering path $\tilde{p} = [a_i, \ldots, a_n]$ it holds that $(m^{\mathcal{A}}, a_i, y, \mathcal{B}_{\mathsf{addr}}) \in reach^{\mathcal{A}}$. Then, by inductive hypothesis

$$y \in \left\{ y \in \mathcal{B}_{\mathsf{addr}} \mid \text{for some } p \in \mathcal{B}_{\mathsf{path}},\ (m^{\mathcal{B}}, a_i, y, p) \in reach^{\mathcal{B}} \right\}$$

    Moreover, as $a_i = first(m^{\mathcal{A}}, x, \mathcal{B}_{\mathsf{addr}}) = m^{\mathcal{B}}(x).next^{\mathcal{B}}$ we have that $y \in S_{\mathcal{B}}$.

(2) We show now that $S_{\mathcal{B}} \subseteq S_{\mathcal{A}}$. Let $y \in S_{\mathcal{B}}$. Then, there exists $q \in \mathcal{B}_{\mathsf{path}}$ such that $(m^{\mathcal{B}}, x, y, q) \in reach^{\mathcal{B}}$. By definition of $reach$ there are two possible cases.

- If $q = \epsilon$ and $x = y$, then $(m^{\mathcal{A}}, x, y, \epsilon^{\mathcal{A}}) \in reach^{\mathcal{A}}$ and therefore $y \in S_{\mathcal{A}}$.
- Otherwise, there exist $\tilde{a}_1, \ldots, \tilde{a}_s \in \mathcal{B}_{\mathsf{addr}}$ such that:

$$\begin{array}{llll} i) & q = [\tilde{a}_1, \ldots, \tilde{a}_s] & iii) & m^{\mathcal{A}}(\tilde{a}_r).next^{\mathcal{A}} = \tilde{a}_{r+1}, \text{for } 1 \leq r < s \\ ii) & x = \tilde{a}_1 & iv) & m^{\mathcal{A}}(\tilde{a}_s).next^{\mathcal{A}} = y \end{array}$$

  We proceed by induction on s.

- If $s = 1$, then $q = [\tilde{a}_1]$ and since $\tilde{a}_1 = x$, $m^{\mathcal{B}}(\tilde{a}_1).next^{\mathcal{B}} = y$ and $y \in \mathcal{B}_{\mathsf{addr}}$, we have that $y \in S_{\mathcal{A}}$.
- If $s > 1$, then we have that $\tilde{a}_1 = x$. Let $a_i = \mathit{first}(m^{\mathcal{A}}, x, \mathcal{B}_{\mathsf{addr}})$. As because of $iii)$ we have that $m^{\mathcal{B}}(\tilde{a}_1).next^{\mathcal{B}} = \tilde{a}_2$ and

$$\begin{aligned} \tilde{a}_2 &= m^{\mathcal{B}}(\tilde{a}_1).next^{\mathcal{B}} && (6.32) \\ &= \mathit{first}(m^{\mathcal{A}}, x, \mathcal{B}_{\mathsf{addr}}) && (6.33) \\ &= a_i \end{aligned}$$

Hence, $a_i = \tilde{a}_2$. As

$$y \in \left\{ y \in \mathcal{B}_{\mathsf{addr}} \mid \text{for some } q \in \mathcal{B}_{\mathsf{path}},\ (m^{\mathcal{B}}, \tilde{a}_2, y, p) \in reach_{\mathsf{K}}{}^{\mathcal{B}} \right\}$$

by inductive hypothesis we have that

$$y \in \left\{ y \in \mathcal{B}_{\mathsf{addr}} \mid \text{for some } p \in \mathcal{A}_{\mathsf{path}},\ (m^{\mathcal{A}}, \tilde{a}_2, y, p) \in reach^{\mathcal{A}} \right\}$$

Finally, because of (6.32), (6.33) and Lemma 6.6 we have that $y \in S_{\mathcal{A}}$.

Literals of the form $p = getp(m, a, b)$**:**
We consider two possible cases.

(1) Case $b^{\mathcal{A}} \in addr2set(m^{\mathcal{A}}, a^{\mathcal{A}})$.
Since $(m^{\mathcal{A}}, a^{\mathcal{A}}, b^{\mathcal{A}}, p^{\mathcal{A}}) \in reach^{\mathcal{A}}$, it is enough to prove:

$$\text{if} \quad (m^{\mathcal{A}}, x, y, q) \in reach^{\mathcal{A}} \quad \text{then} \quad (m^{\mathcal{B}}, x, y, compress(q, \mathcal{B}_{\mathsf{addr}})) \in reach^{\mathcal{B}}$$

for each $x, y \in \mathcal{B}_{\mathsf{addr}}$ and $q \in \mathcal{A}_{\mathsf{path}}$.

- If $(m^{\mathcal{A}}, x, y, q) \in reach^{\mathcal{A}}$, $x = y$ and $q = \epsilon$, then $(m^{\mathcal{B}}, x, y, compress(q, \mathcal{B}_{\mathsf{addr}})) \in reach^{\mathcal{B}}$.
- Otherwise, there exist $a_1, \ldots, a_n \in \mathcal{A}_{\mathsf{addr}}$ such that:

$$\begin{array}{llll} i) & q = [a_1, \ldots, a_n] & iii) & m^{\mathcal{A}}(a_r).next^{\mathcal{A}} = a_{r+1}, \text{for } 1 \leq r < n \\ ii) & x = a_1 & iv) & m^{\mathcal{A}}(a_n).next^{\mathcal{A}} = y \end{array}$$

We proceed by induction on n.

- If $n = 1$, then $q = [a_1]$ and therefore $compress(q, \mathcal{B}_{\mathsf{addr}}) = [a_1]$, because $x = a_1 \in \mathcal{B}_{\mathsf{addr}}$. Moreover, $m^{\mathcal{A}}(a_1).next^{\mathcal{A}} = y$ which implies that $m^{\mathcal{B}}(a_1).next^{\mathcal{B}} = y$. Hence, $(m^{\mathcal{B}}, x, y, compress(q, \mathcal{B}_{\mathsf{addr}})) \in reach^{\mathcal{B}}$.
- If $n > 1$, then let $a_i = \mathit{first}(m^{\mathcal{A}}, x, \mathcal{B}_{\mathsf{addr}})$. Since

$$q = [x = a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_n]$$

we have that

$$compress(q, \mathcal{B}_{\mathsf{addr}}) = [x = a_1] \circ compress([a_i, a_{i+1}, \ldots, a_n], \mathcal{B}_{\mathsf{addr}})$$

Now $(m^{\mathcal{A}}, a_i, y, [a_i, a_{i+1}, \ldots, a_n]) \in \mathit{reach}^{\mathcal{A}}$, so by inductive hypothesis we have that

$$(m^{\mathcal{B}}, a_i, y, \mathit{compress}([a_i, a_{i+1}, \ldots, a_n], \mathcal{B}_{\mathsf{addr}})) \in \mathit{reach}^{\mathcal{B}}$$

Moreover, $m^{\mathcal{B}}(x).\mathit{next}^{\mathcal{B}} = a_i$ and therefore

$$(m^{\mathcal{B}}, x, y, \mathit{compress}(q, \mathcal{B}_{\mathsf{addr}})) \in \mathit{reach}^{\mathcal{B}}$$

(2) Case $b^{\mathcal{A}} \notin \mathit{addr2set}(m^{\mathcal{A}}, a^{\mathcal{A}})$.
In this case we have that $p^{\mathcal{A}} = \epsilon$, which implies that $p^{\mathcal{B}} = \epsilon$. Then using a reasoning similar to the previous case we can deduce that $b^{\mathcal{B}} \notin \mathit{addr2set}(m^{\mathcal{B}}, a^{\mathcal{B}})$.

Literals of the form $a = \mathit{fstlock}(m, p)$:
We consider two cases separately:

- $p^{\mathcal{B}} = \epsilon$. In this case $\mathit{fstlock}^{\mathcal{A}}(m^{\mathcal{A}}, \epsilon^{\mathcal{A}}) = \mathit{null}^{\mathcal{A}}$. Now, since $\epsilon^{\mathcal{B}} = \mathit{compress}(\epsilon^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$, then $\mathit{fstlock}^{\mathcal{B}}(m^{\mathcal{B}}, \epsilon^{\mathcal{B}}) = \mathit{null}^{\mathcal{B}}$.
- $p^{\mathcal{B}} = [a_1, \ldots, a_n]$. There are two sub-cases:
  - If for all $1 \le k \le n$, $m^{\mathcal{A}}(a_k^{\mathcal{A}}).\mathit{lockid}^{\mathcal{A}} = \oslash$, then we have that

    $$\mathit{fstlock}^{\mathcal{A}}(m^{\mathcal{A}}, p^{\mathcal{A}}) = \mathit{null}^{\mathcal{A}}$$

    The path returned by $\mathit{compress}$ satisfies that all its addresses also belong to the received path. Therefore, if we have $[\tilde{a}_1, \ldots, \tilde{a}_m] = p^{\mathcal{B}} = \mathit{compress}(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$ then for all $1 \le j \le m$, $m^{\mathcal{B}}(\tilde{a}_j).\mathit{lockid}^{\mathcal{B}} = m^{\mathcal{A}}(\tilde{a}_j).\mathit{lockid}^{\mathcal{A}} = \oslash$, because a_j is present in $p^{\mathcal{A}}$. Hence, $\mathit{fstlock}^{\mathcal{B}}(m^{\mathcal{B}}, p^{\mathcal{B}}) = \mathit{null}^{\mathcal{B}}$.
  - There is a $1 \le k \le n$ such that for all $1 \le j < k$, $m^{\mathcal{A}}(a_j^{\mathcal{A}}).\mathit{lockid}^{\mathcal{A}} = \oslash$ and $m^{\mathcal{A}}(a_k^{\mathcal{A}}).\mathit{lockid}^{\mathcal{A}} \neq \oslash$. Now, by the construction of $\mathcal{B}$, $a^{\mathcal{B}} = a^{\mathcal{A}}$. Let $[\tilde{a}_1, \ldots, \tilde{a}_i, \ldots, \tilde{a}_m] = \mathit{compress}(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$ such that $\tilde{a}_i = a_k^{\mathcal{A}}$. The existence of $a_k^{\mathcal{A}}$ in $p^{\mathcal{B}}$ is guaranteed because $a_k^{\mathcal{A}} = a_k^{\mathcal{B}} \in \mathcal{B}_{\mathsf{addr}}$. A property of $\mathit{compress}$ is that it preserves the order of the elements, that is, if two address are not filtered, then they appear in the same order. It follows that for all $1 \le s < k$, $a_s^{\mathcal{B}}.\mathit{lockid}^{\mathcal{B}} = a_s^{\mathcal{A}}.\mathit{lockid}^{\mathcal{A}} = \oslash$. Since $a_i^{\mathcal{B}}.\mathit{lockid}^{\mathcal{B}} = a_i^{\mathcal{A}}.\mathit{lockid}^{\mathcal{A}} \neq \oslash$, then $\mathit{fstlock}^{\mathcal{B}}(m^{\mathcal{B}}, p^{\mathcal{B}}) = a_i^{\mathcal{B}} = a_k^{\mathcal{B}}$, as desired.

All literals hold in $\mathcal{B}$, so $\mathcal{B}$ is a model of Γ. □

Example 6.4
Consider again the formula $\varphi_{\mathsf{nonNorm}}$ presented in Example 6.3:

$$\varphi_{\mathsf{nonNorm}} \overset{\text{def}}{=} \mathit{reach}(m, \mathit{head}, \mathit{tail}, \mathit{getp}(m, \mathit{head}, \mathit{tail})) \ \wedge \ m[\mathit{tail}].\mathit{next} = \mathit{null}$$

Once it is normalized, we can compute the bounds for the domains of addresses, elements and thread identifiers for proving the satisfiability of $\varphi_{\mathsf{nonNorm}}$ according to Theorem 6.1.

Addresses: We require a domain with at most 9 addresses. We obtain this result because we require 2 addresses for variables $head$ and $tail$, 1 for $null$, 2 more addresses due to the multiplication between the number of memory variables (only m) and the address variables ($head$ and $tail$), 2 due to the possible inequalities between paths and finally 2 due to possible existence of negations of predicates $append$.

Elements: We require a domain with at most 10 elements. We obtain this results since we need 1 element for variable e, and 9 elements because of the multiplication of variable m with the 9 addresses in the domain of addresses.

Thread identifiers: We require a domain with at most 11 thread identifiers. This happens because we require 1 thread identifier due to the variable t plus 9 identifiers, one for each of the memory variables and the size of the address domain, plus 1 for the special identifier $\oslash$.

In this example we can see that the bounds computed by Theorem 6.1 are theoretical, but in practice they can be reduced to more tighter bounds. For instance, the normalized formula does not contain any path inequality or negation of $append$, so, the bounds for the domain of addresses can be safely reduced to 5 elements. ⌟

6.4 Summary

In this chapter we presented TL3, the *Theory of Linked Lists with Locks*. TL3 is obtained as a combination of theories including addresses, elements, cells, memories and paths. TL3 is specifically designed for describing rich properties of concurrent data structures with the memory shape of single-linked lists. This theory is powerful enough to describe the structure of list-like data types, pointer manipulation, explicit heap regions with region separation and lock ownership. All these features make TL3 suitable for describing structural and functional properties of single-linked lists and similar concurrent data types such as stacks and queues.

We showed TL3 to be decidable for quantifier-free formulas. We presented a bounded model theorem, which ensures that given a quantifier-free TL3 formula, it is possible to compute the bounds of the domains for a model of the formula, if one such model exists. The bounds are computed considering only those literals occurring in the TL3 formula. Following this result, the decision procedure for TL3 can determine the satisfiability of a TL3 formula by enumerating all possible models of the given formula.

TL3 remains crucial in the verification of parametrized programs that manipulates single-linked lists. It can automatically determine the validity of the quantifier-free verification conditions generated by the parametrized invariance rules and the parametrized verification diagrams presented in Chapter 3 and Chapter 4 respectively. Examples of the use of TL3 decision procedure for the verification of concurrent data types can be seen in Chapter 10. TL3 is also a building block for richer theories. For instance, TL3 is the backbone of the theories of skiplists that we present in Chapter 7 and Chapter 8.

An alternative to the model based decision procedure presented in this chapter would consist on using a combination methods like Nelson-Oppen [162]. In this case, we would require decision procedures for each independent theory that conforms TL3 and we would need to propagate

equalities and inequalities through a TL3 formula. However, this approach is left as an idea for future development, as discussed in Chapter 11.

In this chapter we have presented various concurrent data types with the memory shape of single-linked lists. Later, in Chapter 10 we present the results from using the TL3 decision procedure described here in the verification of these concurrent data types.

7

TSL_K: Decidable Theories of Skiplists of Bounded Height

> *People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on eachother, like a wall of mini stones.*
>
> Donald Knuth

In order to attack the verification problem of parametrized systems we implicitly interpret the operational semantics of the program as a state transition system. Then, using the deductive techniques presented in Chapter 3 and Chapter 4 we reduce the verification problem to a proof of validity of a finite collection of verification conditions. These verification conditions capture the state of the program (including the data that the program manipulates) and small changes in this data. When dealing with programs that manipulate the skiplist data structures [171], we require decision procedures capable of dealing with skiplist memory shapes in order to automatically prove these verification conditions.

A skiplist is a data structure used to implement sets by maintaining several ordered singly-linked lists in memory, where each level is a sublist of the level below. Skiplists are heavily used in practice because they offer a performance comparable to balanced binary trees and are amenable to more efficient implementation. Skiplists are difficult to reason about automatically because of the sharing between the different layers.

In this chapter we present TSL_K—the *Family of Theories of Concurrent Skiplists with at Most K Levels*— a decidable family of theories that allow to reason about the skiplist memory layout, for

skiplists of unbounded length but bounded height. We construct $\mathsf{TSL_K}$ from $\mathsf{TL3}$, the quantifier-free theory of single-linked lists presented in Chapter 6 by extending $\mathsf{TL3}$ in a non-trivial way with order and sublist of ordered lists. The family of theories $\mathsf{TSL_K}$ is capable of expressing skiplist-like properties of skiplist memory shapes up to a constant number of levels. The $\mathsf{TSL_K}$ family plays a crucial role in the construction and decidability proof of TSL, a theory for skiplists with arbitrary many levels, presented in Chapter 8.

In this chapter we also show that the satisfiability problem for quantifier-free $\mathsf{TSL_K}$ formulas is decidable, making it applicable to the verification of skiplists implementations. The fact that $\mathsf{TSL_K}$ can express skiplist-like properties without using quantifiers, allows $\mathsf{TSL_K}$ to be combined with other theories.

The rest of the chapter is structured as follows. Section 7.1 presents the skiplist data structure including an implementation with bounded levels. Section 7.2 formally presents the family of theories $\mathsf{TSL_K}$. Section 7.3 shows that $\mathsf{TSL_K}$ is decidable by stating and proving a bounded model theorem. Finally, Section 7.4 presents a summary of what have been introduced in this chapter.

7.1 Skiplists

We begin by describing the skiplist data structure. A skiplist [171] is an imperative data structure that implements sets, maintaining several sorted singly-linked lists in dynamic memory. Skiplists are structured in multiple levels, each level consisting of a single linked list. The skiplist property establishes that the list at level $i + 1$ is a sublist of the list at level i. Each node in a skiplist stores a value and at least the pointer corresponding to the lowest level list. Some nodes also contain pointers at higher levels, pointing to the next element present at that level. Due to their simple layout, the main advantage of skiplists is that they are simpler and more efficient to implement than search trees, while search is still (probabilistically) logarithmic.

Fig. 7.1 presents an example of a skiplist with 4 levels (levels are labeled from 0 to 3). The skiplist contains two sentinel nodes pointed by $head$ and $tail$. A skiplist maintains all its nodes ordered and the nodes pointed by $head$ and $tail$ point to the nodes with the lowest and highest possible values, in this case denoted by $-\infty$ and $+\infty$. As can be seen in the figure, there cannot be any node in the skiplist with a level higher than the levels present in $head$ and $tail$. In the example presented in Fig. 7.1 we use $\mathsf{0x01} \dots \mathsf{0x08}$ to denote memory addresses. The set of nodes reachable at each level is a subset of the set of reachable nodes at the level immediately below.

Contrary to single-linked lists implementations, higher-level pointers allow to *skip* many

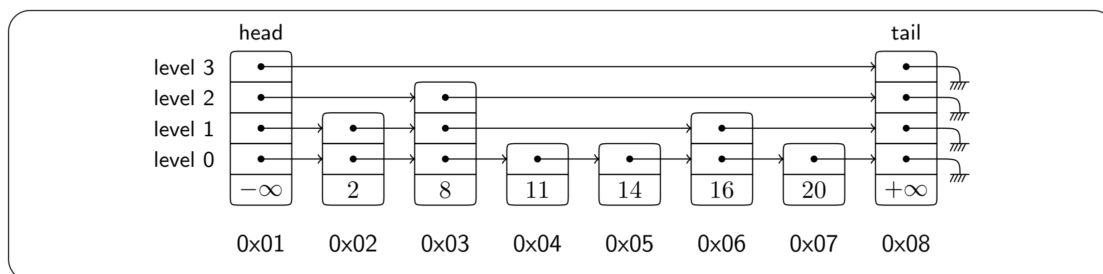


Figure 7.1: A skiplist with 4 levels.

elements during the search, that is why this data structure is called *skiplist*.

Example 7.1
Consider again the skiplist presented in Fig. 7.1. Imagine now that a thread wants to check whether value 20 is in the skiplist. In a traditional single-linked lists all nodes are connected at a unique level and hence, a thread looking for value 20 would require to start from $head$ and traverse the list visiting each node until the node with value 20 is found. Roughly speaking, if we consider the skiplist in Fig. 7.1, this is equivalent to start from $head$ and advance just using the pointers at level 0.

On the other hand, Fig. 7.2 describes how a thread traverses the skiplist while searching for value 20. We use gray to mark the path followed by a thread advancing from head to the node with value 20. The thread performs the search from left to right in a top down fashion, progressing as much as possible in a level without going beyond the value it is searching for before descending.

In a skiplist, a search for value 20 starts at level 3 of node $head$. As the node pointed by $head$ contains a value lower than 20, the thread advances in the skiplist at level 3 checking whether the following node at level 3 contains a value higher or equal than 20. In our example, when the thread is at level 3 in the node pointed by $head$ it looks ahead following the pointer at level 3 and determines that the next node at such level is $tail$ and contains $+\infty$, a value greater than 20. Therefore, the search algorithm decides to move down one level, to level 2 of node $head$. At level 2, the successor of $head$ contains value 8, which is smaller than 20. Hence, as a skiplist keeps the values of the nodes ordered, it is safe to go to node 8 (stored at address 0x03) at level 2 since we are sure that the nodes we are skipping (0x02 in our example) must contain a value lower than 8. This way, the search continues at level 2 until a node containing a greater value is found. As the following node at level 2 is again $tail$, which contains a value greater that 20, the search moves down one level again and advances until the node with value 16 is reached. Finally, once we descend to level 1, the thread finds that the next node contains value 20, finishing the search. ⌟

Note that through the search procedure we have skipped some nodes. This capability of *skipping* unnecessary nodes is what gives the data structure its name. The search is expected to be logarithmic because the probability of any given node occurring at a certain level decreases by $1/2$ as a level increases (see [171] for an analysis of the running time of skiplists).

We now present an implementation of a concurrent bounded skiplist. The implementation we present here corresponds to a *fine-grained locking* implementation which contains for each node

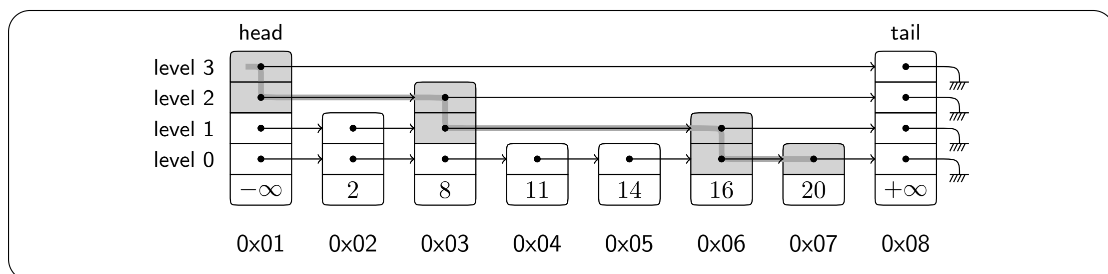


Figure 7.2: Skiplist traversal while searching for value 20.

and level, a lock which protects the pointer at such level. This implementation behaves similar to a lock-coupling single-linked list, in the sense that when a thread traverses the skiplist, it acquires the lock of the node level that it visits, and only releases this lock after the lock of the successor node has been successfully acquired.

Class *BoundedSkiplistNode* defines the nodes of a concurrent skiplist of K levels as follows:

$$\textbf{class} \quad \mathit{BoundedSkiplistNode} \quad \{ \quad \begin{array}{ll} \mathit{Elem} & \mathit{data}; \\ \mathit{Array}\langle \mathit{Addr} \rangle(\mathsf{K}) & \mathit{next}; \\ \mathit{Array}\langle \mathit{Lock} \rangle(\mathsf{K}) & \mathit{locks}; \quad \} \end{array}$$

As can be seen, an object of class *BoundedSkiplistNode* contains the following fields:

- *data*, which keeps the current element stored in the node;
- *next*, an array of length K of pointers which keeps the addresses of the next node in the skiplist at each level;
- *locks*, an array of length K of locks which protect each pointer at all possible levels.

As usual, we assume that the operating system provides the atomic operations *lock* and *unlock* to lock and unlock each of the locks. As for lists, in this case it is also easy to adapt the framework with additional fields in *BoundedSkiplistNode*. For example, we could use different fields for the element stored (for example a value) and for the data used to keep the list ordered (a key).

Our implementation of a concurrent bounded skiplist maintains two global addresses *head* and *tail*, and two ghost global variables *reg* and *elems*:

```
global
  Addr head
  Addr tail
  Set⟨Addr⟩ reg
  Set⟨Elem⟩ elems
```

The global program variables are:

(a) A variable *head*, of type address (pointer), which points to the first node of the skiplist.

(b) A variable *tail*, of type address (pointer), which points to the last node of the skiplist.

(c) A ghost variable *reg*, of type set of addresses, which is used to keep track of the portion of the heap that forms the skiplist.

(d) A ghost variable *elems*, of type set of elements, which represents the collection of elements stored in the skiplist.

In our implementation, *head* and *tail* point to the nodes with the lowest and highest possible values, $-\infty$ and $+\infty$ respectively. The nodes pointed by *head* and *tail* are sentinel nodes which are neither removed nor modified. We assume that the list is initialized with *head* and *tail* already allocated and initialized. The set *reg* is initialized containing only the addresses of *head* and *tail*. Similarly, the set *elems* is initialized containing only the elements initially stored at the nodes pointed by *head* and *tail*. We also assume that K is a constant value, representing the

maximum number of levels in the skiplist. Each verification effort then, proves correctness of the implementation for a particular value of K, for example $\mathsf{K} = 3$.

Our implementation of a concurrent skiplist with at most K levels contains three operations named SEARCH, INSERT and REMOVE:

SEARCH: described in Fig. 7.3, receives an element e and traverses the skiplist in order to determine whether e is stored in the skiplist.

The procedure uses four local variables. An integer local variable i to denote the current level, two local pointers $prev$ and $curr$ which are used to traverse the skiplist and a Boolean $found$ which is set in case element e is found in the skiplist.

As we showed in Example 7.1, the procedure searches element e from left to right and from the highest to the lowest possible level. Initially i is set to the highest level $\mathsf{K} - 1$ (line 1), $prev$ points to $head$ (line 2) and $curr$ points to the node immediately after $head$ at level i

```
    procedure SEARCH(Elem e)
      Int i
      Addr prev, curr
      Bool found
    begin
1:  i := K − 1
2:  prev := head
3:  lock(prev→locks[i])
4:  curr := prev→next[i]
5:  lock(curr→locks[i])
6:  while 0 ≤ i ∧ curr→data ≠ e do
7:      if i < K − 1 then
8:          lock(prev→locks[i])
9:          curr := prev→next[i]
10:         lock(curr.locks[i])
11:         unlock(prev→next[i + 1]→locks[i + 1])
12:         unlock(prev→locks[i + 1])
13:     end if
14:     while curr.data < e do
15:         unlock(prev→locks[i])
16:         prev := curr
17:         curr := prev→next[i]
18:         lock(curr→locks[i])
19:     end while
20:     i := i − 1
21: end while
22: found := (curr→data = e)
23: if i = K − 1 then
24:     unlock(curr→locks[i])
25:     unlock(prev→locks[i])
26: else
27:     unlock(curr→locks[i + 1])
28:     unlock(prev→locks[i + 1])
29: end if
30: return found
    end procedure
```

Figure 7.3: SEARCH procedure for concurrent skiplists of K levels.

(line 4). Then, the locks at level i in the nodes pointed by $prev$ and $curr$ are locked (lines 3 and 5). Intuitively, $curr$ is used to mark the node that is currently being traversed by the procedure and $prev$ is used to point to the node immediately preceding $curr$ at level i.

The loop that follows (lines 6 to 21) is in charge of looking for the node storing element e. This loop runs while we have not reached the bottom of the skiplist ($0 \leq i$) and the element has not been found yet ($curr.data \neq e$). The conditional within the loop (lines 7 to 13) is in charge of releasing the locks in upper levels once the search goes down a level. Because of this, the branch is disabled in the first iteration of the loop, when $i = \mathsf{K} - 1$.

To have a better understanding, the sequence of steps taken for the procedure between line 8 and 12 is depicted as a sequence of snapshots in Fig. 7.4. The first snapshot (Fig. 7.4(a)) corresponds to the moment at which $prev$ points to $head$, $curr$ points to $tail$ and the locks at level 3 are acquired for both nodes. This corresponds exactly to the state of the skiplist presented in Example 7.1, when a thread searching for value 20 has executed lines 1 through 7. As the last statement within the external loop (line 20) decrements the value of i, at this point we have that $i = 2$ and $head$ and $tail$ at level 3 remain locked. When line 8 is executed, the lock at level 2 is also grabbed at the node pointed by $prev$ (Fig. 7.4(b)). At this point, $curr$ is modified to point to the successor node of $prev$ at level $i = 2$ (Fig. 7.4(c)). Line 10 gets the lock at level i corresponding to the node pointed by $curr$ (Fig. 7.4(c)). Finally, lines 11 and 12 release the locks at level $i + 1 = 3$, what leaves the skiplist with only nodes $prev$ and $curr$ locked at level i (Fig. 7.4(d)).

The inner loop between lines 14 and 19 makes the procedure advance through the skiplist at level i. To do so, it first releases the lock at the node pointed by $prev$ (line 15) and then makes $prev$ to advance to point to the same node as $curr$ (line 16). Then, pointer $curr$ is advanced at level i (line 17) and finally the procedure grabs the lock at level i in the new node pointed by $curr$ (line 18). In this sense, the way the procedure advances through the skiplist for a fixed level is similar to the one employed at the lock-coupling concurrent single-linked list presented in Section 6.1.1. This way of progressing is called *hand-in-hand* locking.

At the end of the loop between lines 6 and 21, the following holds:

$$prev {\rightarrow} data < e \wedge curr {\rightarrow} data \geq e$$

Hence, since the nodes in the skiplist are ordered, at line 22 it is possible to determine whether element e is in the skiplist just by checking if $curr {\rightarrow} data = e$. The final section of the procedure (line 23 to 29) releases the locks that were left on the nodes pointed by $prev$ and $curr$. There is a subtle difference regarding the level at which the locks need to be released. If the loop that goes from line 6 to 21 has not been executed because, for example, initially $curr$ pointed at a node which stores value e, then i is never decremented and thus the locks remain at level $i = \mathsf{K} - 1$. On the other hand, if the loop that goes from line 6 to 21 has been executed, it means that i has been decremented and the locks remains at one level higher than the one currently declared by i. Because of this, in this latter case the locks to be released are at level $i + 1$.

The procedure finishes by returning the value of $found$ which indicates whether element e was found in a node of the skiplist.

INSERT: shown in Fig. 7.5, receives an element e and attempts to insert it into the skiplist. INSERT first determines the position at which e should be inserted and then manipulates the pointers of the neighbour nodes accordingly.

The procedure begins by randomly choosing the maximum level of the new node to be inserted (line 1), in case no node with value e is found in the skiplist. Then, from line 2 to 6 the procedure behaves exactly like SEARCH, making $prev$ point to $head$ and $curr$ point to

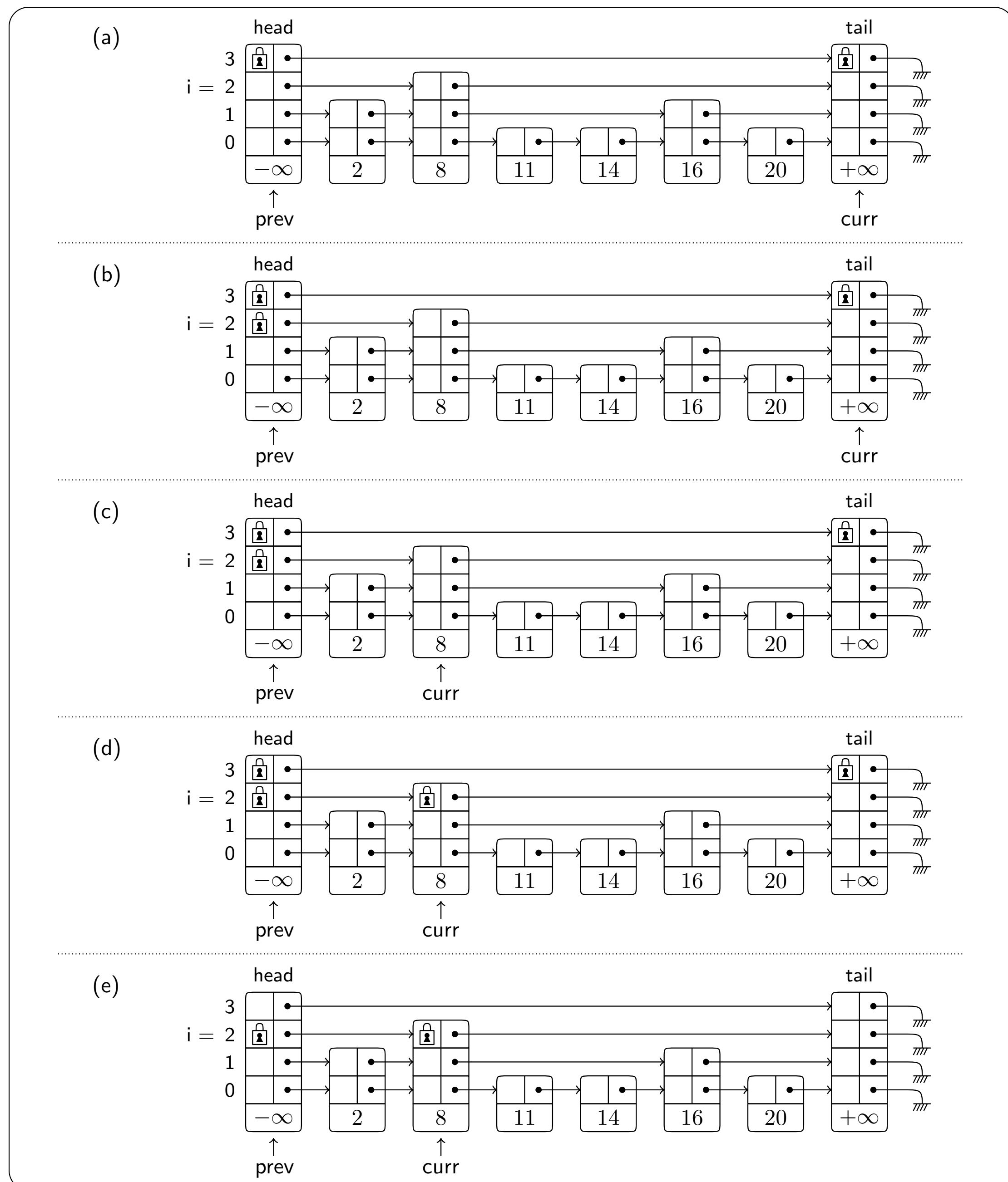


Figure 7.4: Snapshots of the execution of process SEARCH over a concurrent skiplist when descending a level.

the successor of $head$ at level $\mathsf{K}-1$. Moreover, this procedure grabs the locks at level $\mathsf{K}-1$ for the nodes pointed by $prev$ and $curr$.

```
    procedure Insert(Elem e)
      Int i
      Addr prev, curr
      Array⟨Addr⟩(K) upd
    begin
 1: lvl := randomLevel(K)
 2: i := K − 1
 3: prev := head
 4: lock(prev→locks[1])
 5: curr := prev→next[i]
 6: lock(curr→locks[i])
 7: while i ≥ 0 do
 8:     if i < K − 1 then
 9:         lock(prev→locks[i])
10:         curr := prev→next[i]
11:         lock(curr→locks[i])
12:         if i ≥ lvl then
13:             unlock(curr→locks[i + 1])
14:             unlock(prev→locks[i + 1])
15:         end if
16:     end if
17:     while curr→val < e do
18:         unlock(prev→locks[i])
19:         prev := curr
20:         curr := prev→next[i]
21:         lock(curr→locks[i])
22:     end while
23:     upd[i] := prev
24:     i := i − 1
25: end while
26: if curr→data = e then
27:     for i := 0 to lvl do
28:         unlock(upd[i]→next[i]→locks[i])
29:         unlock(upd[i]→locks[i])
30:     end for
31: else
32:     newnode := CreateNode(lvl, e)
33:     for i := 0 to lvl do
34:         newnode→next[i] := upd[i]→next[i]
35:         upd[i]→next[i] := newnode
            ┌─────────────────────────────────┐
            │ if i = 0 then                   │
            │     reg := reg ∪ {newnode}      │
            │     elems := elems ∪ {e}        │
            └─────────────────────────────────┘
36:         unlock(newnode→next[i]→locks[i])
37:         unlock(upd[i]→locks[i])
38:     end for
39: end if
40: return
    end procedure
```

Figure 7.5: INSERT procedure for concurrent skiplists of K levels.

From line 7 to 25 there is a loop that behaves similarly to the outer loop of procedure SEARCH we described above. There are, however, two key differences:

(1) We use an array of addresses *upd* to keep the values of *prev* pointers before decrementing a level (line 23). This is needed because, after the procedure has reached the position where the new node needs to be inserted, it is required to modify pointers belonging to different nodes and at different levels and hence it needs to remember which are these pointers.

(2) The procedure needs to acquire some locks because when it inserts a new node some pointers need to me modified. Because of this, once we descend a level, we will release the locks of the upper level only if we are sure they will not be needed in case a new node needs to be inserted (line 12 to 15).

To illustrate why we need to keep the *prev* pointers and why we cannot release the locks below *lvl*, let's consider an example. Imagine the INSERT procedure is trying to insert a node with $lvl = 2$ containing value 15 in the skiplist depicted in Fig. 7.6(a). Fig. 7.6(b) and Fig. 7.6(c) show the changes that are required in order to insert such node. Fig. 7.6(b)

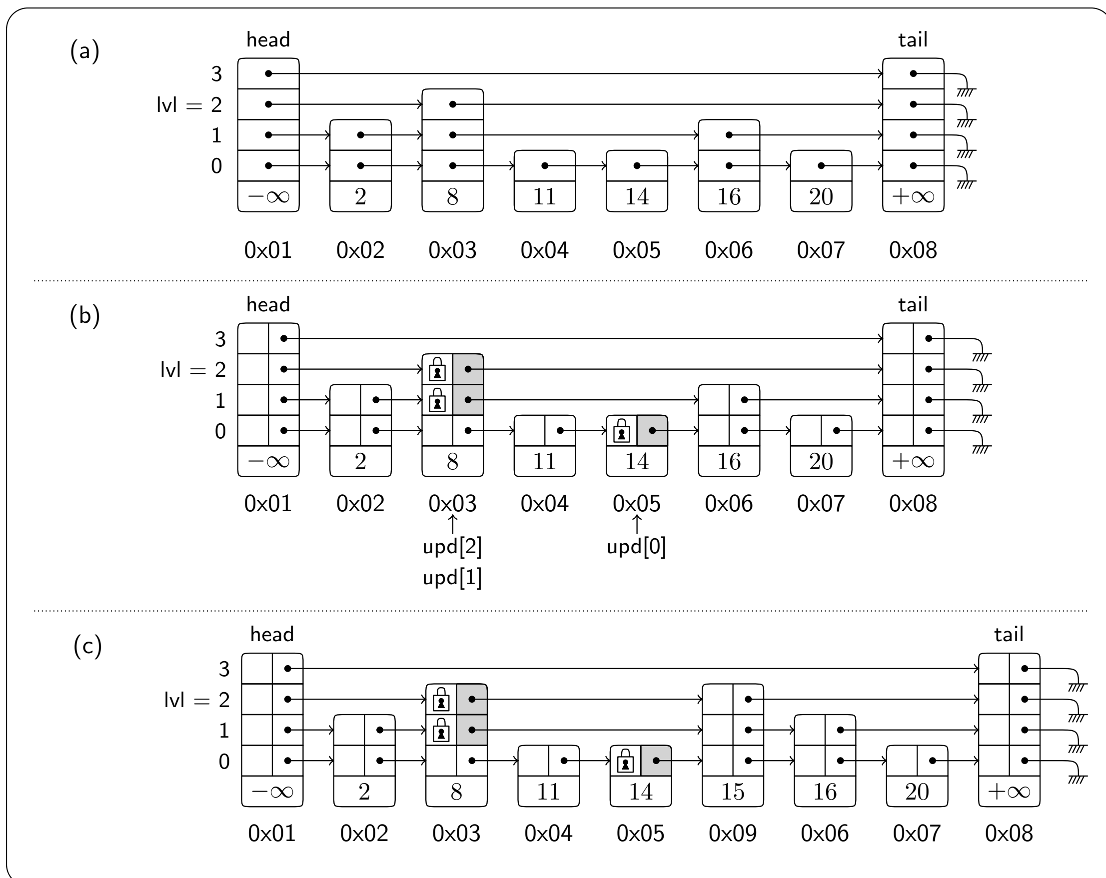


Figure 7.6: Example of inserting a node of 3 levels and value 15 into a skiplist. (a) shows that original skiplist, (b) shows the state before the node is connected and (c) shows the state after the node is connected. We color in gray the pointers that are modified.

shows the skiplist before the node is inserted. We color in gray the pointers that need to be modified when connecting the new node to the skiplist. Note that the gray pointers correspond to the ones stored in the upd array. Additionally, as we need to protect these pointers, the locks at each of these levels are kept thanks to the conditional that goes from line 12 to 15. Finally, the figure at the bottom (Fig. 7.6(c)) shows the state of the skiplist after the new node has been connected.

As usual, once we have finished with the initial loop, the following holds:

$$prev{\rightarrow}data < e \wedge curr{\rightarrow}data \geq e$$

Therefore, if at line 26 we have that $curr{\rightarrow}data = e$, then there is already another node in the skiplist with value e in which case we do not need to insert a new node. In this case, the procedure just releases the locks it had acquired (lines 27 to 30).

On the other hand, if $curr{\rightarrow}data \neq e$, a new node needs to be inserted. In this case the procedure proceeds to allocate a fresh node $newnode$ of height lvl which stores value e (line 32). Finally, in a bottom-up fashion it proceeds to connect $newnode$ to the rest of the skiplist modifying the pointers stored at upd array and releasing all unnecessary locks. Note that $newnode$ starts being part of the skiplist as soon as it is fully connected at level 0, and hence we require to update the ghost variables reg and $elems$ accordingly at line 35 in order to reflect this insertion.

REMOVE: presented in Fig. 7.7, receives as argument an element e and removes the node containing such element form the skiplist by redirecting the next pointer of the previous nodes at all levels appropriately.

Initially, procedure REMOVE acts as SEARCH or INSERT, by making $prev$ point to $head$ (line 2), $curr$ point to the successor of $head$ at the highest possible level (line 4) and setting the locks of both nodes at level i (lines 3 and 5). Similarly, as in SEARCH and REMOVE, the first loop (lines 6 to 20) is in charge of finding the position where the node with value e should be

To do so, the procedure advances through the skiplist from left to right on a top-down way. The main difference in the case of REMOVE is that when the execution goes down one level in the skiplist, it keeps the lock of $prev$ at the level immediately above. This is done because, a priori, it is not known the maximum level at which the pointers will be modified when the node with value e is removed.

Once REMOVE determines the position at which the node with value e should be in the skiplist, it executes a loop (line 21 to 29) to disconnects the node. The loop disconnects the node in a top-down fashion to guarantee that the skiplist property (each level is a sublist of the level immediately below holds). The conditional inside the final loop (line 22) checks whether the successor of the pointer stored in upd at each level points to the node with value e. If so, then the node with value e can be disconnected at level i from the rest of the skiplist (line 23) and its lock can be released (line 24). If the conditional at line 22 does not hold, then we only need to release the lock at the successor of upd at level i (line 26).

Note that the node is disconnected in a top-down way, so the node is not fully removed form the skiplist until it has not been disconnected at level 0. Hence, when level 0 is reached, we

```
procedure REMOVE(Elem e)
   Int i
   Addr prev, curr
   Array⟨Addr⟩(K) upd
begin
 1: i := K − 1
 2: prev := head
 3: lock(prev→locks[i])
 4: curr := prev→next[i]
 5: lock(curr→locks[i])
 6: while i ≥ 0 do
 7:    if i < K − 1 then
 8:       lock(prev→locks[i])
 9:       curr := prev→next[i]
10:       lock(curr→locks[i])
11:    end if
12:    while curr.data < e do
13:       unlock(prev→locks[i])
14:       prev := curr
15:       curr := prev→next[i]
16:       lock(curr→locks[i])
17:    end while
18:    upd[i] := prev
19:    i := i − 1
20: end while
21: for i := K − 1 downto 0 do
22:    if upd[i]→next[i] = curr ∧ curr→data = e then
23:       upd[i]→next[i] := curr→next[i]
          if i = 0 then
             reg := reg − {curr}
             elems := elems − {e}
24:       unlock(curr→locks[i])
25:    else
26:       unlock(upd[i]→next[i]→locks[i])
27:    end if
28:    unlock(upd[i]→locks[i])
29: end for
30: if curr→data = e then
31:    free (curr)
32: end if
33: return
end procedure
```

Figure 7.7: REMOVE procedure for concurrent skiplists of K levels.

can update the ghost variables *reg* and *elems* to reflect the complete removal of the node (line 23).

Finally, the lock at the node pointed by *upd* at level i can be released (line 28) as all changes on the skiplist at such level has been completed. The procedure finishes by freeing the node containing element e in case it has been removed from the skiplist.

It is easy to see that, line 35 of INSERT and line 23 of REMOVE correspond to the linearization points of these methods of the concurrent data type. As usual, in order to verify the data type

```
   procedure MGCSkiplistK()
      Elem e
   begin
1: while true do
2:    e := havocSkiplistElem()
3:    nondet choice
4:          call Search(e)
5:       or call Insert(e)
6:       or call Remove(e)
7:    end choice
8: end while
   end procedure
```

Figure 7.8: Most general client procedure MGCSkiplistK for concurrent skiplist of K levels.

against all possible clients, we create the most general client of the concurrent skiplist, called MGCSkiplistK. This most general client is presented in Fig. 7.8. This program consists of an infinite loop which simply invokes non-deterministically any of the procedures Search, Insert and Remove that implements the data type, with an arbitrary parameter.

7.2 TSL_K: A Family of Theories for Skiplists of Bounded Height

In this section we formally present the family of theories TSL_K which is capable of describing skiplists of bounded height with at most K levels for a fixed constant K. This family of theories is powerful enough to describe rich properties of skiplist heap memory layouts. As with TL3 presented in Chapter 6, each member of the TSL_K family is a multi-sorted first-order theory.

Each theory member in the family TSL_K is a decidable theory with capabilities to reason about reachability in single-linked lists, locks, ordered lists, and sublists of ordered lists. We show that TSL_K enjoys a bounded model property and we provide computable bounds on the size of a large enough model. Again, this fact implies that each theory in TSL_K is decidable simply by performing a search in the finite space of candidate models.

The theory TSL_K to reason about skiplists of height K combines different theories and is built as an extension of the Theory of Concurrent Linked Lists (TL3) presented in Chapter 6 in the following way:

- the reasoning about single level lists is extended to all the K levels. This means that all functions and predicates to reason about single level lists are replicated for all K levels.
- we introduce new functions and predicates to model ordered lists and the sub-paths relation between lists.

We formally define the *Theories of Concurrent Skiplists with bounded Levels*, TSL_K for short, as a combination of theories $\mathsf{TSL}_\mathsf{K} = (\Sigma_{\mathsf{TSL}_\mathsf{K}}, \mathbf{TSLK})$, where

$$\Sigma_{\mathsf{TSL}_\mathsf{K}} \;=\; \Sigma_{\mathsf{level}_\mathsf{K}} \cup \Sigma_{\mathsf{cell}_\mathsf{K}} \cup \Sigma_{\mathsf{mem}_\mathsf{K}} \cup \Sigma_{\mathsf{setaddr}} \cup \Sigma_{\mathsf{settid}} \cup \Sigma_{\mathsf{setelem}} \cup \Sigma_{\mathsf{reach}_\mathsf{K}} \cup \Sigma_{\mathsf{bridge}_\mathsf{K}}$$

Informally, $\Sigma_{\mathsf{level}_\mathsf{K}}$ models the K levels of the skiplist. $\Sigma_{\mathsf{cell}_\mathsf{K}}$ models *cells*, structures containing an element which is also used to keep order between nodes (key), K different addresses (pointers)

and K locks which protect each of the successor pointers. A cell represents a node in the skiplist. $\Sigma_{\mathsf{mem_K}}$ models the memory again as a map from addresses to cells. $\Sigma_{\mathsf{setaddr}}$ models sets of addresses. Σ_{settid} models sets of thread identifiers. $\Sigma_{\mathsf{setelem}}$ models sets of elements. $\Sigma_{\mathsf{reach_K}}$ models finite sequences of non-repeating addresses, to represent acyclic paths in memory. Finally, $\Sigma_{\mathsf{bridge_K}}$ is a *bridge theory* containing auxiliary functions that allow to map paths of addresses to set of addresses, or to obtain the set of addresses reachable from a given address following a chain of $next$ fields.

We describe now the sorts, the signature and restrictions on the interpretation for each of the theories in $\mathsf{TSL_K}$.

7.2.1 Sorts

The sorts shared among these theories are $\mathsf{level_K}$, elem, tid, addr, $\mathsf{cell_K}$, $\mathsf{mem_K}$, path, setaddr, setelem and settid. The intended meaning of these sorts is:

- $\mathsf{level_K}$: levels of the skiplist.
- elem: elements stored in these cells.
- tid: thread identifiers.
- addr: memory addresses.
- $\mathsf{cell_K}$: instances of the class $BoundedSkiplistNode$ above, stored in the heap.
- $\mathsf{mem_K}$: heaps, as maps from addresses to cells.
- path: paths, as finite sequences of non-repeating addresses.
- setaddr: sets of addresses.
- settid: sets of thread identifiers.
- setelem: sets of elements.

The class of models **TSLK** restrict the domain of candidate interpretations $\mathcal{A}$ to satisfy the following:

(a) $\mathcal{A}_{\mathsf{level_K}}$ is the finite collection of levels $0, \dots, \mathsf{K}-1$.

(b) $\mathcal{A}_{\mathsf{elem}}$, $\mathcal{A}_{\mathsf{tid}}$ and $\mathcal{A}_{\mathsf{addr}}$ are discrete sets. We require $\mathcal{A}_{\mathsf{elem}}$ to be a totally ordered set and $\mathcal{A}_{\mathsf{tid}}$ to contain a special symbol $\oslash$.

(c) $\mathcal{A}_{\mathsf{cell_K}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{tid}}^{\mathsf{K}} \times \mathcal{A}_{\mathsf{addr}}^{\mathsf{K}}$.

(d) $\mathcal{A}_{\mathsf{mem_K}} = \mathcal{A}_{\mathsf{cell_K}}^{\mathcal{A}_{\mathsf{addr}}}$.

(e) $\mathcal{A}_{\mathsf{path}}$ is the set of all finite sequences of (pairwise) distinct elements of $\mathcal{A}_{\mathsf{addr}}$.

(f) $\mathcal{A}_{\mathsf{setaddr}}$ is the power-set of $\mathcal{A}_{\mathsf{addr}}$.

(g) $\mathcal{A}_{\mathsf{settid}}$ is the power-set of $\mathcal{A}_{\mathsf{tid}}$.

(h) $\mathcal{A}_{\mathsf{setelem}}$ is the power-set of $\mathcal{A}_{\mathsf{elem}}$.

7.2.2 Signature

We describe now the signature of each theory, listing the sorts used and each of the functions and predicates with their signatures:

a) $\Sigma_{\mathsf{level_K}}$: The only sort used is $\mathsf{level_K}$. The function symbols are:

$$0, \ldots, \mathsf{K}-1 \quad : \quad \mathsf{level_K}$$

The constant functions $0, \ldots, \mathsf{K}-1$ model each of the K levels of the skiplist. The predicate symbols, apart from equality between levels, are:

$$< \quad : \quad \mathsf{level_K} \times \mathsf{level_K}$$

The predicate $<$ describes the order relation between two given skiplist levels.

b) Σ_{elem}: The only sort used is elem. This theory contains two function symbols:

$$\begin{aligned} -\infty &\quad : \quad \mathsf{elem} \\ +\infty &\quad : \quad \mathsf{elem} \end{aligned}$$

The constant functions $-\infty$ and $+\infty$ model the lowest and highest possible values in the domain of elements respectively. The only predicate symbol it contains the theory, apart from equality between elements, is:

$$\preceq \quad : \quad \mathsf{elem} \times \mathsf{elem}$$

The predicate $\preceq$ describes the order relation between skiplist elements. For instance, $e_1 \preceq e_2$, states that element e_1 is lower or equal than element e_2.

c) $\Sigma_{\mathsf{cell_K}}$: The sorts used are $\mathsf{cell_K}$, $\mathsf{level_K}$, elem, tid and addr. The function symbols are:

$$\begin{aligned} error &\quad : \quad \mathsf{cell_K} \\ mkcell &\quad : \quad \mathsf{elem} \times \mathsf{addr}^\mathsf{K} \times \mathsf{tid}^\mathsf{K} \to \mathsf{cell_K} \\ _.data &\quad : \quad \mathsf{cell_K} \to \mathsf{elem} \\ _.next[_] &\quad : \quad \mathsf{cell_K} \times \mathsf{level_K} \to \mathsf{addr} \\ _.lockid[_] &\quad : \quad \mathsf{cell_K} \times \mathsf{level_K} \to \mathsf{tid} \\ _.lock[_,_] &\quad : \quad \mathsf{cell_K} \times \mathsf{level_K} \times \mathsf{tid} \to \mathsf{cell_K} \\ _.unlock[_] &\quad : \quad \mathsf{cell_K} \times \mathsf{level_K} \to \mathsf{cell_K} \end{aligned}$$

The cell $error$ is used to model the return of a memory dereference. The function $mkcell$ is the constructor of cells. The corresponding selectors are the functions $data$, $next$ and $lockid$. Selector $data$ is used to access the element in a cell. As we assume that the domain of elements is a total ordered set, selector $data$ can be used to give an order between cells. Meanwhile, selector $next$ can be used to access the addresses of the successor node at each skiplist level while $lockid$ is used to access the locks that protect the pointers at each level. The function $lock$ receives a cell c, a level l and a thread identifier t and returns a new cell which matches c in every field except that the lock at level l is now assigned to thread t.

Similarly, the function *unlock* receives a cell c and a level l and returns a new cell which coincides with c in every field except that the lock at level l is released (assigned to $\oslash$). There are no predicate symbols in $\Sigma_{\mathsf{cell}_\mathsf{K}}$ except from equality between cells.

d) $\mathbf{\Sigma_{mem_K}}$**:** The sorts used are mem_K, addr and cell_K. The function symbols are:

$$\begin{array}{rcl} null & : & \mathsf{addr} \\ _[_] & : & \mathsf{mem}_\mathsf{K} \times \mathsf{addr} \to \mathsf{cell}_\mathsf{K} \\ upd & : & \mathsf{mem}_\mathsf{K} \times \mathsf{addr} \times \mathsf{cell}_\mathsf{K} \to \mathsf{mem}_\mathsf{K} \end{array}$$

The function *null* models the null address. The function _[_] models a memory dereference that returns a cell given a memory and an address. Finally, the function *upd* is used to create a modified memory given a memory, an address and the cell stored in that address. There are no predicate symbols in $\Sigma_{\mathsf{mem}_\mathsf{K}}$ except from equality between memories.

e) $\mathbf{\Sigma_{setaddr}}$**:** The sorts used are addr and $\mathsf{setaddr}$. The function symbols are:

$$\begin{array}{rcl} \emptyset & : & \mathsf{setaddr} \\ \{_\} & : & \mathsf{addr} \to \mathsf{setaddr} \\ \cup, \cap, \setminus & : & \mathsf{setaddr} \times \mathsf{setaddr} \to \mathsf{setaddr} \end{array}$$

The predicate symbols, in addition to set equality, are:

$$\begin{array}{rcl} \in & : & \mathsf{addr} \times \mathsf{setaddr} \\ \subseteq & : & \mathsf{setaddr} \times \mathsf{setaddr} \end{array}$$

The intended interpretation of these symbols is their usual meaning in set theory, for finite sets of addresses.

f) $\mathbf{\Sigma_{settid}}$**:** The sorts used are tid and settid. The function symbols are:

$$\begin{array}{rcl} \emptyset_\mathrm{T} & : & \mathsf{settid} \\ \{_\}_\mathrm{T} & : & \mathsf{tid} \to \mathsf{settid} \\ \cup_\mathrm{T}, \cap_\mathrm{T}, \setminus_\mathrm{T} & : & \mathsf{settid} \times \mathsf{settid} \to \mathsf{settid} \end{array}$$

The predicate symbols in Σ_{settid}, in addition to set equality, are:

$$\begin{array}{rcl} \in_\mathrm{T} & : & \mathsf{tid} \times \mathsf{settid} \\ \subseteq_\mathrm{T} & : & \mathsf{settid} \times \mathsf{settid} \end{array}$$

Again, the intended interpretation of these symbols is their usual meaning in set theory, for finite sets of thread identifiers.

g) $\mathbf{\Sigma_{setelem}}$**:** The sorts used are elem and $\mathsf{setelem}$. The function symbols are:

$$\begin{array}{rcl} \emptyset_\mathrm{E} & : & \mathsf{setelem} \\ \{_\}_\mathrm{E} & : & \mathsf{elem} \to \mathsf{setelem} \\ \cup_\mathrm{E}, \cap_\mathrm{E}, \setminus_\mathrm{E} & : & \mathsf{setelem} \times \mathsf{setelem} \to \mathsf{setelem} \end{array}$$

The predicate symbols in $\Sigma_{\mathsf{setelem}}$, in addition to set equality, are:

$$\begin{array}{rcl} \in_{\mathrm{E}} & : & \mathsf{elem} \times \mathsf{setelem} \\ \subseteq_{\mathrm{E}} & : & \mathsf{setelem} \times \mathsf{setelem} \end{array}$$

Once again, the intended interpretation of these symbols is their usual meaning in set theory, for finite sets of elements.

h) $\Sigma_{\mathsf{reach}_\mathsf{K}}$: The sorts used are mem_K, addr, $\mathsf{level}_\mathsf{K}$ and path. The function symbols are:

$$\begin{array}{rcl} \epsilon & : & \mathsf{path} \\ [_] & : & \mathsf{addr} \rightarrow \mathsf{path} \end{array}$$

The constant ϵ models the empty path, and the function $[_]$ allows to build a singleton path with only the provided address in it. The predicate symbols in $\Sigma_{\mathsf{reach}_\mathsf{K}}$, apart from equality between paths, are:

$$\begin{array}{rcl} append & : & \mathsf{path} \times \mathsf{path} \times \mathsf{path} \\ reach_\mathsf{K} & : & \mathsf{mem}_\mathsf{K} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level}_\mathsf{K} \times \mathsf{path} \end{array}$$

The predicate $append$ relates two paths with its concatenation. A path must be a sequence of non-repeated elements, so some pairs of paths cannot be concatenated, if they contain common elements. The predicate $reach_\mathsf{K}$ relates two addresses with the path that connects them in a given heap at a certain skiplist level.

i) $\Sigma_{\mathsf{bridge}_\mathsf{K}}$: The sorts used are mem_K, addr, $\mathsf{level}_\mathsf{K}$, $\mathsf{setaddr}$ and path. The function symbols are:

$$\begin{array}{rcl} path2set & : & \mathsf{path} \rightarrow \mathsf{setaddr} \\ getp_\mathsf{K} & : & \mathsf{mem}_\mathsf{K} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level}_\mathsf{K} \rightarrow \mathsf{path} \\ addr2set_\mathsf{K} & : & \mathsf{mem}_\mathsf{K} \times \mathsf{addr} \times \mathsf{level}_\mathsf{K} \rightarrow \mathsf{setaddr} \end{array}$$

The function $path2set$ receives a path and returns the set of addresses present in the given path. The function $addr2set_\mathsf{K}$ returns the set of addresses reachable from a given address by following the $next$ pointers at a certain skiplist level. The function $getp_\mathsf{K}$ returns the path that connects two addresses in a given heap at a certain level, if there is one (or the empty path otherwise). The predicate symbols are:

$$\begin{array}{rcl} ordPath & : & \mathsf{mem}_\mathsf{K} \times \mathsf{path} \end{array}$$

The predicate $ordPath$ captures whether the data stored at the cells obtained from the addresses in a given path when mapped through the given heap are ordered.

7.2.3 Interpretations

We restrict the class of models to **TSLK**, a class of $\Sigma_{\mathsf{TSL}_\mathsf{K}}$-structures that satisfy the following conditions:

a) $\Sigma_{\mathsf{cell}_\mathsf{K}}$: Every interpretation $\mathcal{A}$ of $\Sigma_{\mathsf{cell}_\mathsf{K}}$ must satisfy, for every element $e \in \mathcal{A}_{\mathsf{elem}}$, every tuple of addresses $a_{0..\mathsf{K}-1} \in \mathcal{A}_{\mathsf{addr}}$, every tuple of thread identifiers $t_{0..\mathsf{K}-1} \in \mathcal{A}_{\mathsf{tid}}$, and every

$l \in \mathcal{A}_{\mathsf{level}_\mathsf{K}}$:

- $mkcell^{\mathcal{A}}(e, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1}) = \langle e, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1} \rangle$
- $\langle e, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1} \rangle.data^{\mathcal{A}} = e$
- $\langle e, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1} \rangle.next^{\mathcal{A}}[l] = a_l$
- $\langle e, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1} \rangle.lockid^{\mathcal{A}}[l] = t_l$
- $\langle e, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1} \rangle.lock^{\mathcal{A}}[l, t] = \langle e, a_{0..\mathsf{K}-1}, t_{0..l-1}, t, t_{l+1..\mathsf{K}-1} \rangle$
- $\langle e, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1} \rangle.unlock^{\mathcal{A}}[l, t] = \langle e, a_{0..\mathsf{K}-1}, t_{0..l-1}, \oslash, t_{l+1..\mathsf{K}-1} \rangle$
- $error^{\mathcal{A}}.next^{\mathcal{A}}[l] = null^{\mathcal{A}}$

Essentially, the models in **TSLK** restrict cells to be records consisting of an element, a key to maintain the order between nodes, K addresses, and K thread identifiers which describe the locks at each skiplist level.

b) $\mathbf{\Sigma}_{\mathsf{mem}_\mathsf{K}}$**:** For each $m \in \mathcal{A}_{\mathsf{mem}_\mathsf{K}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $c \in \mathcal{A}_{\mathsf{cell}_\mathsf{K}}$:

- $m[a]^{\mathcal{A}} = m(a)$
- $upd^{\mathcal{A}}(m, a, c) = m_{a \mapsto c}$
- $m^{\mathcal{A}}(null^{\mathcal{A}}) = error^{\mathcal{A}}$

In models in **TSLK**, the memory dereference function returns the cell associated with a given address. The memory update simply transforms the memory into a memory that differs only in the modified address, which points to the given cell. Above, we use $m_{a \mapsto c}$ to denote a memory map that agrees with m in every address, except for address a, which is mapped to cell c. Finally, dereferencing $null$ returns the $error$ cell.

c) $\mathbf{\Sigma}_{\mathsf{setaddr}}$**:** The symbols $\emptyset$, $\{_\}$, $\cup$, $\cap$, $\setminus$, $\in$ and $\subseteq$ are interpreted according to their standard interpretation over finite sets of addresses. Similarly, the symbols $\emptyset_{\mathrm{E}}$, $\{_\}_{\mathrm{E}}$, $\cup_{\mathrm{E}}$, $\cap_{\mathrm{E}}$, $\setminus_{\mathrm{E}}$, $\in_{\mathrm{E}}$ and $\subseteq_{\mathrm{E}}$, and the symbols $\emptyset_{\mathrm{T}}$, $\{_\}_{\mathrm{T}}$, $\cup_{\mathrm{T}}$, $\cap_{\mathrm{T}}$, $\setminus_{\mathrm{T}}$, $\in_{\mathrm{T}}$ and $\subseteq_{\mathrm{T}}$ are interpreted according to their standard interpretations over finite sets of elements and threads respectively.

d) $\mathbf{\Sigma}_{\mathsf{reach}_\mathsf{K}}$**:** The symbol ϵ is interpreted as the empty sequence, and $[a]^{\mathcal{A}}$ is the singleton sequence containing $a \in \mathcal{A}_{\mathsf{addr}}$ as its only element.

- In the case of $append$, its interpretation is as in TL3:

$$([i_1, \dots, i_n], [j_1, \dots, j_m], [i_1, \dots, i_n, j_1, \dots, j_m]) \in append^{\mathcal{A}}$$

  if and only if all $i_1, \dots, i_n, j_1, \dots, j_m$ are all pairwise distinct.

- In the case of $reach_\mathsf{K}$ we need to consider now the levels of the skiplist:

$$(m, i, j, l, p) \in reach_\mathsf{K}{}^{\mathcal{A}}$$

  holds whenever one of the following conditions hold:

  (a) $i = j$ and $p = \epsilon$; or

(b) there exist addresses $i_1, \ldots, i_n \in \mathcal{A}_{\mathsf{addr}}$ such that:

(1) $p = [i_1, \ldots, i_n]$ (3) $m(i_r).next^{\mathcal{A}}[l] = i_{r+1}$, for $1 \leq r < n$

(2) $i_1 = i$ (4) $m(i_n).next^{\mathcal{A}}[l] = j$

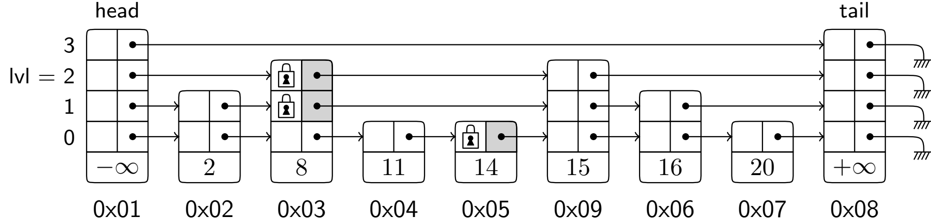
e) $\Sigma_{\mathsf{bridge_K}}$: The interpretation of $addr2set$, $path2set$, $getp_{\mathsf{K}}$ and $ordPath$ are restricted as follows:

- $addr2set^{\mathcal{A}}(m, a, l) = \left\{ b \in \mathcal{A}_{\mathsf{addr}} \mid \text{for some } p \in \mathcal{A}_{\mathsf{path}}, (m, a, b, l, p) \in reach_{\mathsf{K}} \right\}$
- $path2set^{\mathcal{A}}(p) = \{a_1, \ldots, a_n\}$ for $p = [a_1, \ldots, a_n] \in \mathcal{A}_{\mathsf{path}}$
- for each $m \in \mathcal{A}_{\mathsf{mem_K}}$, $p \in \mathcal{A}_{\mathsf{path}}$, $l \in \mathcal{A}_{\mathsf{level_K}}$ and $a_{init}, a_{end} \in \mathcal{A}_{\mathsf{addr}}$:

$$getp_{\mathsf{K}}{}^{\mathcal{A}}(m, a_{init}, a_{end}, l) = \begin{cases} p & \text{if } (m, a_{init}, a_{end}, l, p) \in reach_{\mathsf{K}}{}^{\mathcal{A}} \\ \epsilon & \text{otherwise} \end{cases}$$

- $ordPath^{\mathcal{A}}(m, p)$ if and only if:
  - $p = \epsilon$, or
  - $p = [a]$, or
  - $p = [a_1, \ldots, a_n]$ with $n \geq 2$ and $m(a_i).key^{\mathcal{A}} \preceq m(a_{i+1}).key^{\mathcal{A}}$ for all $1 \leq i < n$.

Example 7.2
Consider the skiplist snapshot depicted in Fig. 7.6(c):



This snapshot represents an intermediate state of a skiplist while a new node in being inserted. Assuming that locks shown in the figure are owned by thread T_1, the following interpretation $\mathcal{A}$ is in the class **TSLK**:

$$\begin{aligned} \mathcal{A}_{\mathsf{addr}} &= \{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08\} \\ \mathcal{A}_{\mathsf{elem}} &= \{-\infty, 2, 8, 11, 14, 16, 20, +\infty\} \\ \mathcal{A}_{\mathsf{tid}} &= \{T_1, \oslash\} \\ \mathcal{A}_{\mathsf{mem_K}} &= \{m : \mathcal{A}_{\mathsf{addr}} \to \mathcal{A}_{\mathsf{cell_K}}\} \end{aligned}$$

where

$$\begin{aligned} null^{\mathcal{A}} &= 0x00 \\ error^{\mathcal{A}} &= \langle\ -\infty\ , 0x00\ , 0x00\ , 0x00\ , 0x00\ , \oslash\ , \oslash\ , \oslash\ , \oslash\ \rangle \end{aligned}$$

$$
\begin{array}{lcl}
m(\mathtt{0x00}) & = & \langle\ -\infty\ , \mathtt{0x00}\ , \mathtt{0x00}\ , \mathtt{0x00}\ , \mathtt{0x00}\ , \oslash\ , \oslash\ , \oslash\ , \oslash\ \rangle \\
m(\mathtt{0x01}) & = & \langle\ -\infty\ , \mathtt{0x02}\ , \mathtt{0x02}\ , \mathtt{0x03}\ , \mathtt{0x08}\ , \oslash\ , \oslash\ , \oslash\ , \oslash\ \rangle \\
m(\mathtt{0x02}) & = & \langle\ 2\ , \mathtt{0x03}\ , \mathtt{0x03}\ , \mathtt{0x00}\ , \mathtt{0x00}\ , \oslash\ , \oslash\ , \oslash\ , \oslash\ \rangle \\
m(\mathtt{0x03}) & = & \langle\ 8\ , \mathtt{0x04}\ , \mathtt{0x06}\ , \mathtt{0x08}\ , \mathtt{0x00}\ , \oslash\ , T_1\ , T_1\ , \oslash\ \rangle \\
m(\mathtt{0x04}) & = & \langle\ 11\ , \mathtt{0x05}\ , \mathtt{0x00}\ , \mathtt{0x00}\ , \mathtt{0x00}\ , \oslash\ , \oslash\ , \oslash\ , \oslash\ \rangle \\
m(\mathtt{0x05}) & = & \langle\ 14\ , \mathtt{0x06}\ , \mathtt{0x00}\ , \mathtt{0x00}\ , \mathtt{0x00}\ , T_1\ , \oslash\ , \oslash\ , \oslash\ \rangle \\
m(\mathtt{0x06}) & = & \langle\ 16\ , \mathtt{0x07}\ , \mathtt{0x08}\ , \mathtt{0x00}\ , \mathtt{0x00}\ , \oslash\ , \oslash\ , \oslash\ , \oslash\ \rangle \\
m(\mathtt{0x07}) & = & \langle\ 20\ , \mathtt{0x08}\ , \mathtt{0x00}\ , \mathtt{0x00}\ , \mathtt{0x00}\ , \oslash\ , \oslash\ , \oslash\ , \oslash\ \rangle \\
m(\mathtt{0x08}) & = & \langle\ +\infty\ , \mathtt{0x00}\ , \mathtt{0x00}\ , \mathtt{0x00}\ , \mathtt{0x00}\ , \oslash\ , \oslash\ , \oslash\ , \oslash\ \rangle
\end{array}
$$

For predicates $reach_{\mathsf{K}}$ and $ordPath$ we have, that:

$$
\begin{array}{lcl}
(m, \mathtt{0x01}, \mathtt{0x01}, 0, \epsilon) & \in & reach^{\mathcal{A}} \\
(m, \mathtt{0x01}, \mathtt{0x01}, 0, [\mathtt{0x01}]) & \notin & reach^{\mathcal{A}} \\
(m, \mathtt{0x01}, \mathtt{0x08}, 1, [\mathtt{0x01}, \mathtt{0x02}, \mathtt{0x03}, \mathtt{0x06}]) & \in & reach^{\mathcal{A}} \\
(m, \mathtt{0x01}, \mathtt{0x08}, 2, [\mathtt{0x01}, \mathtt{0x03}]) & \in & reach^{\mathcal{A}} \\
(m, \mathtt{0x01}, \mathtt{0x08}, 3, [\mathtt{0x01}]) & \in & reach^{\mathcal{A}} \\
 & & \\
(m, \epsilon) & \in & ordPath^{\mathcal{A}} \\
(m, [\mathtt{0x04}]) & \in & ordPath^{\mathcal{A}} \\
(m, [\mathtt{0x03}, \mathtt{0x05}, \mathtt{0x07}]) & \in & ordPath^{\mathcal{A}} \\
(m, [\mathtt{0x03}, \mathtt{0x06}, \mathtt{0x03}]) & \notin & ordPath^{\mathcal{A}}
\end{array}
$$

For functions $next$ and $lockid$ we have:

$$
\begin{array}{c}
\langle 8, \mathtt{0x04}, \mathtt{0x06}, \mathtt{0x08}, \mathtt{0x00}, \oslash, T_1, T_1, \oslash\rangle.next^{\mathcal{A}}[0] = \mathtt{0x04} \\
\langle 8, \mathtt{0x04}, \mathtt{0x06}, \mathtt{0x08}, \mathtt{0x00}, \oslash, T_1, T_1, \oslash\rangle.next^{\mathcal{A}}[1] = \mathtt{0x06} \\
\langle 8, \mathtt{0x04}, \mathtt{0x06}, \mathtt{0x08}, \mathtt{0x00}, \oslash, T_1, T_1, \oslash\rangle.next^{\mathcal{A}}[3] = \mathtt{0x00}
\end{array}
$$

$$
\begin{array}{c}
\langle 8, \mathtt{0x04}, \mathtt{0x06}, \mathtt{0x08}, \mathtt{0x00}, \oslash, T_1, T_1, \oslash\rangle.lockid^{\mathcal{A}}[0] = \oslash \\
\langle 8, \mathtt{0x04}, \mathtt{0x06}, \mathtt{0x08}, \mathtt{0x00}, \oslash, T_1, T_1, \oslash\rangle.lockid^{\mathcal{A}}[2] = T_1
\end{array}
$$

If, for instance, $c^{\mathcal{A}}$ is the interpretation given above for the cell stored at address 0x03, we have for functions $lock$ and $unlock$ that:

$$
\begin{array}{c}
c.lock^{\mathcal{A}}[0, T_2] = \langle 8, \mathtt{0x04}, \mathtt{0x06}, \mathtt{0x08}, \mathtt{0x00}, T_2, T_1, T_1, \oslash\rangle \\
c.lock^{\mathcal{A}}[1, T_2] = \langle 8, \mathtt{0x04}, \mathtt{0x06}, \mathtt{0x08}, \mathtt{0x00}, \oslash, T_2, T_1, \oslash\rangle
\end{array}
$$

$$c.unlock^{\mathcal{A}}[0] = \langle 8, \text{0x04}, \text{0x06}, \text{0x08}, \text{0x00}, \oslash, T_1, T_1, \oslash \rangle$$
$$c.unlock^{\mathcal{A}}[1] = \langle 8, \text{0x04}, \text{0x06}, \text{0x08}, \text{0x00}, \oslash, \oslash, T_1, \oslash \rangle$$

Finally, for functions $addr2set_{\mathsf{K}}$ and $getp_{\mathsf{K}}$ we have, for instance:

$$addr2set_{\mathsf{K}}{}^{\mathcal{A}}(m, \text{0x01}, 1) = \{\text{0x01}, \text{0x02}, \text{0x03}, \text{0x06}, \text{0x08}\}$$
$$addr2set_{\mathsf{K}}{}^{\mathcal{A}}(m, \text{0x01}, 2) = \{\text{0x01}, \text{0x03}, \text{0x08}\}$$

$$getp_{\mathsf{K}}{}^{\mathcal{A}}(m, \text{0x01}, \text{0x01}, 1) = \epsilon$$
$$getp_{\mathsf{K}}{}^{\mathcal{A}}(m, \text{0x01}, \text{0x04}, 1) = \epsilon$$
$$getp_{\mathsf{K}}{}^{\mathcal{A}}(m, \text{0x01}, \text{0x08}, 2) = [\text{0x01}, \text{0x03}]$$

┘

7.2.4 Satisfiability of TSL_K

We now show that the satisfiability problem of quantifier-free first order TSL_K formulas is decidable by showing that enjoys the bounded model property in the following sense. Given a formula φ, there are bounds for the sizes of the domains of all sorts, such that if there is a model of φ then there is a model within the bounds. Moreover, these bounds can be effectively computed from φ. The fact that TSL_K has the bounded model property with respect to domains elem, addr, and $\mathsf{level}_\mathsf{K}$, implies that TSL_K is decidable because one can enumerate all possible $\Sigma_{\mathsf{TSL}_\mathsf{K}}$-structures up to the cardinality provided by the bounds.

To compute the domain bounds we proceed this way. First, given a formula φ we consider a subset of TSL_K-literals, called normalized literals. All other literals can be rewritten using only normalized literals. As usual, considering only normalized literals aids in simplifying the theoretical developments later. Then, after normalizing all literal occurring in φ, we transform φ into its disjunctive normal form $\varphi_1 \vee \cdots \vee \varphi_n$, where each φ_i is a conjunction of flat TSL_K literals. We now define the set of normalized TSL_K-literals.

Definition 7.1 (TSL_K-normalized Literals).
A TSL_K-literal is normalized if it is a flat literal of the form:

$e_1 \neq e_2$	$a_1 \neq a_2$	$l_1 \neq l_2$
$a = null$	$c = error$	$t_1 \neq t_2$
$e_1 \preceq e_2$	$m_2 = upd(m_1, a, c)$	$c = m[a]$
$l_1 < l_2$	$c = mkcell(e, a_{0..K-1}, t_{0..K-1})$	
$s = \{a\}$	$s_1 = s_2 \cup s_3$	$s_1 = s_2 \setminus s_3$
$r = \{t\}_\mathrm{T}$	$r_1 = r_2 \cup_\mathrm{T} r_3$	$r_1 = r_2 \setminus_\mathrm{T} r_3$
$x = \{e\}_\mathrm{E}$	$x_1 = x_2 \cup_\mathrm{E} x_3$	$x_1 = x_2 \setminus_\mathrm{E} x_3$
$p_1 \neq p_2$	$p = [a]$	

$s = path2set(p)$ $\quad$ $append(p_1, p_2, p_3)$ $\quad$ $\neg append(p_1, p_2, p_3)$
$s = addr2set_K(m, a, l)$ $\quad$ $p = getp_K(m, a_1, a_2, l)$ $\quad$ $ordPath(m, p)$

where e, e_1 and e_2 are elem*-variables; a, a_0, a_1, $a_2, \ldots, a_{K-1}$ are* addr*-variables; t, t_0, t_1, $t_2, \ldots, t_{K-1}$ are* tid*-variables; c is a* cell_K*-variable; m, m_1 and m_2 are* mem_K*-variables; p, p_1, p_2 and p_3 are* path*-variables; s, s_1, s_2 and s_3 are* setaddr*-variables; r, r_1, r_2 and r_3 are* settid*-variables; x, x_1, x_2 and x_3 are* setelem*-variables and l, l_1 and l_2 are* $\mathsf{level}_\mathsf{K}$*-variables.* ⌟

Lemma 7.1 (TSL_K Normalization):
Every non-normalized TSL_K-literal can be rewritten into an equivalent quantifier-free formula that contains only TSL_K-normalized literals. ⌟

Proof. We use the following equivalences to convert each non-normalized TSL_K literal into formulas containing only TSL_K-normalized literals:

$$e = c.data \quad \leftrightarrow \quad \begin{pmatrix} (\exists_{\mathsf{addr}} a_{0..\mathsf{K}-1}\ \exists_{\mathsf{tid}} t_{0..\mathsf{K}-1}) \\ [c = mkcell\,(e, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1})] \end{pmatrix} \tag{7.1}$$

$$a = c.next[l] \quad \leftrightarrow \quad \begin{pmatrix} (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{addr}} a_{0..l-1}, a_{l+1..K-1}\ \exists_{\mathsf{tid}} t_{0..\mathsf{K}-1}) \\ [c = mkcell\,(e, a_{0..l-1}, a, a_{l+1..\mathsf{K}-1}, t_{0..\mathsf{K}-1})] \end{pmatrix} \tag{7.2}$$

$$t = c.lockid[l] \quad \leftrightarrow \quad \begin{pmatrix} (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{addr}} a_{0..\mathsf{K}-1}\ \exists_{\mathsf{tid}} t_{0..l-1}, t_{l+1..\mathsf{K}-1}) \\ [c = mkcell\,(e, a_{0..\mathsf{K}-1}, t_{0..l-1}, t, t_{l+1..\mathsf{K}-1})] \end{pmatrix} \tag{7.3}$$

$$c_1 = c_2.lock[l,t] \quad \leftrightarrow \quad \begin{pmatrix} (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{addr}} a_{0..\mathsf{K}-1}\ \exists_{\mathsf{tid}} t_{0..\mathsf{K}-1}) \\ c_1 = mkcell(e, a_{0..\mathsf{K}-1}, t_{0..l-1}, t, t_{l+1..\mathsf{K}-1}) \ \wedge \\ c_2 = mkcell(e, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1}) \end{pmatrix} \tag{7.4}$$

$$c_1 = c_2.unlock[l] \quad \leftrightarrow \quad \begin{pmatrix} (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{addr}} a_{0..\mathsf{K}-1}\ \exists_{\mathsf{tid}} t_{0..\mathsf{K}-1}) \\ c_1 = mkcell(e, a_{0..\mathsf{K}-1}, t_{0..l-1}, \oslash, t_{l+1..\mathsf{K}-1}) \ \wedge \\ c_2 = mkcell(e, a]_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1}) \end{pmatrix} \tag{7.5}$$

$$c_1 \neq_{\mathsf{cell}_\mathsf{K}} c_2 \quad \leftrightarrow \quad \begin{pmatrix} \left(\exists_{\mathsf{elem}} e_1, e_2\ \exists_{\mathsf{addr}} a_{0..\mathsf{K}-1}, b_{0..\mathsf{K}-1}\ \exists_{\mathsf{tid}} t_{0..\mathsf{K}-1}, w_{0..\mathsf{K}-1}\right) \\ c_1 = mkcell(e_1, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1}) \ \wedge \\ c_2 = mkcell(e_2, b_{0..\mathsf{K}-1}, w_{0..\mathsf{K}-1}) \ \wedge \\ \left(e_1 \neq e_2 \vee \bigvee_{n=0}^{\mathsf{K}-1} a_n \neq b_n \vee \bigvee_{n=0}^{\mathsf{K}-1} t_n \neq w_n\right) \end{pmatrix} \tag{7.6}$$

$$m_1 \neq_{\mathsf{mem}_\mathsf{K}} m_2 \quad \leftrightarrow \quad (\exists_{\mathsf{addr}} a)\,[m_1[a] \neq m_2[a]] \tag{7.7}$$

$$s_1 \neq_{\mathsf{setaddr}} s_2 \quad \leftrightarrow \quad (\exists_{\mathsf{addr}} a)\,[a \in (s_1 \setminus s_2) \cup (s_2 \setminus s_1)] \tag{7.8}$$

$$s = \emptyset \quad \leftrightarrow \quad s = s \setminus s \tag{7.9}$$

$$s_3 = s_1 \cap s_2 \quad \leftrightarrow \quad s_3 = (s_1 \cup s_2) \setminus ((s_1 \setminus s_2) \cup (s_2 \setminus s_1)) \tag{7.10}$$

$$a \in s \quad \leftrightarrow \quad \{a\} \subseteq s \tag{7.11}$$

$$s_1 \subseteq s_2 \quad \leftrightarrow \quad s_2 = s_1 \cup s_2 \tag{7.12}$$

$$r_1 \neq_{\mathsf{settid}} r_2 \quad \leftrightarrow \quad (\exists_{\mathsf{tid}} t)\,[t \in_{\mathsf{T}} (r_1 \setminus_{\mathsf{T}} r_2) \cup_{\mathsf{T}} (r_2 \setminus_{\mathsf{T}} r_1)] \tag{7.13}$$

$$r = \emptyset_{\mathrm{T}} \quad \leftrightarrow \quad r = r \setminus_{\mathrm{T}} r \tag{7.14}$$

$$r_3 = r_1 \cap_{\mathrm{T}} r_2 \quad \leftrightarrow \quad r_3 = (r_1 \cup_{\mathrm{T}} r_2) \setminus_{\mathrm{T}} ((r_1 \setminus_{\mathrm{T}} r_2) \cup_{\mathrm{T}} (r_2 \setminus_{\mathrm{T}} r_1)) \tag{7.15}$$

$$t \in_{\mathrm{T}} r \quad \leftrightarrow \quad \{t\}_{\mathrm{T}} \subseteq_{\mathrm{T}} r \tag{7.16}$$

$$r_1 \subseteq_{\mathrm{T}} r_2 \quad \leftrightarrow \quad r_2 = r_1 \cup_{\mathrm{T}} r_2 \tag{7.17}$$

$$x_1 \neq_{\mathsf{setelem}} x_2 \quad \leftrightarrow \quad (\exists_{\mathsf{elem}} e)\, [e \in_{\mathrm{E}} (x_1 \setminus_{\mathrm{E}} x_2) \cup_{\mathrm{E}} (x_2 \setminus_{\mathrm{E}} x_1)] \tag{7.18}$$

$$x = \emptyset_{\mathrm{E}} \quad \leftrightarrow \quad x = x \setminus_{\mathrm{E}} x \tag{7.19}$$

$$x_3 = x_1 \cap_{\mathrm{E}} x_2 \quad \leftrightarrow \quad x_3 = (x_1 \cup_{\mathrm{E}} x_2) \setminus_{\mathrm{E}} ((x_1 \setminus_{\mathrm{E}} x_2) \cup_{\mathrm{E}} (x_2 \setminus_{\mathrm{E}} x_1)) \tag{7.20}$$

$$e \in_{\mathrm{E}} x \quad \leftrightarrow \quad \{e\}_{\mathrm{E}} \subseteq_{\mathrm{E}} x \tag{7.21}$$

$$x_1 \subseteq_{\mathrm{E}} x_2 \quad \leftrightarrow \quad x_2 = x_1 \cup_{\mathrm{E}} x_2 \tag{7.22}$$

$$p = \epsilon \quad \leftrightarrow \quad append(p, p, p) \tag{7.23}$$

$$reach_{\mathsf{K}}(m, a_1, a_2, l, p) \quad \leftrightarrow \quad a_2 \in addr2set_{\mathsf{K}}(m, a_1, l) \land p = getp_{\mathsf{K}}(m, a_1, a_2, l) \tag{7.24}$$

$$\neg ordPath(m, p) \quad \leftrightarrow \quad \left(\begin{array}{l} \exists_{\mathsf{addr}} a_1, a_2\ \exists_{\mathsf{cell_K}} c_1, c_2 \\ \quad \left(\begin{array}{lcll} append(p_1, p_2, p) & & & \land \\ a_1 & \in & path2set(p_1) & \land \\ a_2 & \in & path2set(p_2) & \land \\ c_1 & = & m[a_1] & \land \\ c_2 & = & m[a_2] & \land \\ c_1 & = & mkcell(e_1, b_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1}) & \land \\ c_2 & = & mkcell(e_2, d_{0..\mathsf{K}-1}, w_{0..\mathsf{K}-1}) & \land \\ e_2 & \preceq & e_1 & \land \\ e_2 & \neq & e_1 & \end{array} \right) \end{array} \right) \tag{7.25}$$

We prove each equivalence separately. In each case, we assume a model $\mathcal{A}$ of the left hand side of the equivalence and show that $\mathcal{A}$ is a model of the corresponding formula on the right hand side of the equivalence. Similarly, we assume a model $\mathcal{B}$ of the formula on the right and prove that $\mathcal{B}$ is a model of the literal on the left.

- Equivalence (7.1). Let $\mathcal{A}$ be a model of the literal $e = c.data$. Then, $c^{\mathcal{A}}$ is an element of $\mathcal{A}_{\mathsf{cell_K}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{addr}}^{\mathsf{K}} \times \mathcal{A}_{\mathsf{tid}}^{\mathsf{K}}$, so there are x in $\mathcal{A}_{\mathsf{elem}}$, $i_{0..\mathsf{K}-1}$ in $\mathcal{A}_{\mathsf{addr}}$ and $u_{0..\mathsf{K}-1}$ in $\mathcal{A}_{\mathsf{tid}}$ with $c^{\mathcal{A}} = (x, i_{0..\mathsf{K}-1}, u_{0..\mathsf{K}-1})$. By the restriction on the class **TSLK** of models, $\mathcal{A}$ must satisfy that $e^{\mathcal{A}} = x$, and $mkcell^{\mathcal{A}}(x, b_{0..\mathsf{K}-1}, u_{0..\mathsf{K}-1}) = c^{\mathcal{A}}$. Hence, by taking $a_0^{\mathcal{A}} = i_0, \ldots, a_{\mathsf{K}-1}^{\mathcal{A}} = i_{\mathsf{K}-1}$ and $t_0^{\mathcal{A}} = u_0, \ldots, t_{\mathsf{K}-1}^{\mathcal{A}} = u_{\mathsf{K}-1}$ it follows that

$$(\exists_{\mathsf{addr}} a_{0..\mathsf{K}-1}\ \exists_{\mathsf{tid}} t_{0..\mathsf{K}-1})\, [c = mkcell\,(e, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1})] \tag{7.26}$$

holds in $\mathcal{A}$. For the other direction, we assume $\mathcal{B}$ is a model of (7.26). It follows, by the interpretation in $\mathcal{B}$ of $data$, that $c^{\mathcal{B}}.data^{\mathcal{B}}$ is $e^{\mathcal{B}}$ as desired.

- Equivalences (7.2) and (7.3) are analogous to equivalence (7.1).

- Equivalence (7.4). Let $\mathcal{A}$ be a model of $c_1 = c_2.lock[l, t]$. It follows that

$$c_1^{\mathcal{A}} = \langle x, i_{0..\mathsf{K}-1}, u_{0..l-1}, t^{\mathcal{A}}, u_{l+1..\mathsf{K}-1} \rangle \quad \text{and} \quad c_2^{\mathcal{A}} = \langle x, i_{0..\mathsf{K}-1}, u_{0..\mathsf{K}-1} \rangle$$

for some $x \in \mathcal{A}_{\mathsf{elem}}$, $i_{0..\mathsf{K}-1} \in \mathcal{A}_{\mathsf{addr}}$ and $u_{0..\mathsf{K}-1} \in \mathcal{A}_{\mathsf{tid}}$. Hence, by taking $e^{\mathcal{A}} = x$, $a_0^{\mathcal{A}} = i_0, \ldots, a_{\mathsf{K}-1}^{\mathcal{A}} = i_{\mathsf{K}-1}$ and $t_0^{\mathcal{A}} = u_0, \ldots, t_{\mathsf{K}-1}^{\mathcal{A}} = u_{\mathsf{K}-1}$ it follows that:

$$c_1 = mkcell(e, a_{0..\mathsf{K}-1}, t_{0..l-1}, t, t_{l+1..\mathsf{K}-1}) \quad \text{and} \quad c_2 = mkcell(e, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1})$$

hold in $\mathcal{A}$. For the other direction, every model $\mathcal{B}$ of

$$c_1 = mkcell(e, a_{0..\mathsf{K}-1}, t_{0..l-1}, t, t_{l+1..\mathsf{K}-1}) \ \wedge \ c_2 = mkcell(e, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1})$$

is such that

$$\begin{array}{l} c_1^{\mathcal{B}} = \langle e^{\mathcal{B}}, a_0^{\mathcal{B}}, \ldots, a_{\mathsf{K}-1}^{\mathcal{B}}, t_0^{\mathcal{B}}, \ldots, t_{l-1}^{\mathcal{B}}, t^{\mathcal{B}}, t_{l+1}^{\mathcal{B}}, \ldots, t_{\mathsf{K}-1}^{\mathcal{B}} \rangle \quad \text{and} \\ c_2^{\mathcal{B}} = \langle e^{\mathcal{B}}, a_0^{\mathcal{B}}, \ldots, a_{\mathsf{K}-1}^{\mathcal{B}}, t_0^{\mathcal{B}}, \ldots, t_{\mathsf{K}-1}^{\mathcal{B}} \rangle \end{array}$$

If follows, by the interpretation of $lock$ that both $c_1^{\mathcal{B}}$ and $c_2^{\mathcal{B}}.lock^{\mathcal{B}}(l^{\mathcal{B}}, t^{\mathcal{B}})$ are the cell $\langle e^{\mathcal{B}}, a_0^{\mathcal{B}}, \ldots, a_{\mathsf{K}-1}^{\mathcal{B}}, t_0^{\mathcal{B}}, \ldots, t_{l-1}^{\mathcal{B}}, t^{\mathcal{B}}, t_{l+1}^{\mathcal{B}}, \ldots, t_{\mathsf{K}-1}^{\mathcal{B}} \rangle$.

- Equivalence (7.5). Analogous to (7.4).

- Equivalence (7.6). Consider a model $\mathcal{A}$ of $c_1 \neq_{\mathsf{cell}} c_2$ and let $c_1^{\mathcal{A}} = \langle x, i_{0..\mathsf{K}-1}, u_{0..\mathsf{K}-1} \rangle$ and $c_2^{\mathcal{A}} = \langle y, j_{0..\mathsf{K}-1}, w_{0..\mathsf{K}-1} \rangle$. Since $(c_1 \neq_{\mathsf{cell}} c_2)^{\mathcal{A}}$ holds, it follows that either:

  - $x \neq y$, or
  - $i_n \neq j_n$ for some $n \in 0, \ldots, \mathsf{K} - 1$, or
  - $u_n \neq w_n$ for some $n \in 0, \ldots, \mathsf{K} - 1$.

  For the other direction, consider a model $\mathcal{B}$ of

$$\begin{array}{rcl} c_1 = mkcell(e_1, a_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1}) & & \wedge \\ c_2 = mkcell(e_2, b_{0..\mathsf{K}-1}, w_{0..\mathsf{K}-1}) & & \wedge \\ \left(e_1 \neq e_2 \ \vee \ \bigvee_{n=0}^{\mathsf{K}-1} a_n \neq b_n \ \vee \ \bigvee_{n=0}^{\mathsf{K}-1} t_n \neq w_n \right) & & \end{array}$$

  Hence,

$$\begin{array}{l} c_1^{\mathcal{B}} = \langle e_1^{\mathcal{B}}, a_0^{\mathcal{B}}, \ldots, a_{\mathsf{K}-1}^{\mathcal{B}}, t_0^{\mathcal{B}}, \ldots, t_{\mathsf{K}-1}^{\mathcal{B}} \rangle \quad \text{and} \\ c_2^{\mathcal{B}} = \langle e_2^{\mathcal{B}}, b_0^{\mathcal{B}}, \ldots, b_{\mathsf{K}-1}^{\mathcal{B}}, w_0^{\mathcal{B}}, \ldots, w_{\mathsf{K}-1}^{\mathcal{B}} \rangle \end{array}$$

  Then, if $e_1^{\mathcal{B}} \neq e_2^{\mathcal{B}}$, $a_n^{\mathcal{B}} \neq b_n^{\mathcal{B}}$ or $t_n^{\mathcal{B}} \neq w_n^{\mathcal{B}}$ for some $n \in 0, \ldots, \mathsf{K} - 1$ we have that $c_1^{\mathcal{B}} \neq c_2^{\mathcal{B}}$.

- Equivalence (7.7). The proof is analogous to the proof for equivalence (6.7) from Lemma 6.1 introduced in Chapter 6.

- Equivalences (7.8), (7.9), (7.10), (7.11) and (7.12). The proof is analogous to the proof for equivalences (6.8), (6.9), (6.10), (6.11) and (6.12) involving sets of addresses from Lemma 6.1 introduced in Chapter 6.

- Equivalences (7.13), (7.14), (7.15), (7.16) and (7.17). The proof is analogous to the proof for equivalences (6.13), (6.14), (6.15), (6.16) and (6.17) involving sets of thread identifiers from Lemma 6.1 introduced in Chapter 6.

- Equivalences (7.18), (7.19), (7.20), (7.21) and (7.22). The proof is analogous to the proof for equivalences (6.18), (6.19), (6.20), (6.21) and (6.22) involving sets of elements from Lemma 6.1 introduced in Chapter 6.

- Equivalence (7.23). Is analogous to the proof for equivalence (6.23) from Lemma 6.1 introduced in Chapter 6.

- Equivalence (7.24). Assume that in model $\mathcal{A}$, the literal $reach_{\mathsf{K}}(m, a_1, a_2, l, p)$ holds. The interpretation in **TSLK** of $reach_{\mathsf{K}}$ makes:

  $$addr2set_{\mathsf{K}}{}^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, l^{\mathcal{A}}) = \{j \in \mathcal{A}_{\mathsf{addr}} \mid \text{for some path } p,\ (m^{\mathcal{A}}, a_1^{\mathcal{A}}, j, l^{\mathcal{A}}, p^{\mathcal{A}}) \in reach_{\mathsf{K}}{}^{\mathcal{A}}\}$$

  Since $(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, l^{\mathcal{A}}, p^{\mathcal{A}}) \in reach_{\mathsf{K}}{}^{\mathcal{A}}$, it follows that $a_2^{\mathcal{A}} \in addr2set_{\mathsf{K}}{}^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, l^{\mathcal{A}})$ as desired. Similarly, $getp_{\mathsf{K}}{}^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, l^{\mathcal{A}}) = p^{\mathcal{A}}$.

  For the other direction, let $\mathcal{B}$ be a model of:

  $$a_2 \in addr2set_{\mathsf{K}}(m, a_1, l) \wedge p = getp_{\mathsf{K}}(m, a_1, a_2, l)$$

  The second conjunct, $(p = getp_{\mathsf{K}}(m, a_1, a_2, l))$, implies that $(m^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, l^{\mathcal{B}}, p^{\mathcal{B}}) \in reach_{\mathsf{K}}{}^{\mathcal{B}}$ and then $reach_{\mathsf{K}}(m, a_1, a_2, l, p)$ holds in $\mathcal{B}$.

- Equivalence (7.25). Let $\mathcal{A}$ be a model of $\neg ordPath(m, p)$. Then, according to the restriction of **TSLK** models, we have that p is neither ϵ nor a path containing a single address. Then, we have that p should be $[a_1, \ldots, a_n]$ with $n \geq 2$ such that there exists an $1 \leq i < n$ for which $m(a_i).data^{\mathcal{A}} \not\preceq m(a_{i+1}).data^{\mathcal{A}}$, which is equivalent to:

  $$m(a_{i+1}).data^{\mathcal{A}} \preceq m(a_i).data^{\mathcal{A}} \ \wedge\ m(a_{i+1}).data^{\mathcal{A}} \neq m(a_i).data^{\mathcal{A}}$$

  This implies that there are elements x_1, x_2 in $\mathcal{A}_{\mathsf{elem}}$; z_1, z_2 in $\mathcal{A}_{\mathsf{cell_K}}$, $r_{0..\mathsf{K}-1}, s_{0..\mathsf{K}-1}$ in $\mathcal{A}_{\mathsf{addr}}$ and $t_{0..\mathsf{K}-1}, w_{0..\mathsf{K}-1}$ in $\mathcal{A}_{\mathsf{tid}}$ such that:

  $$\begin{array}{ccccccc} z_1 = m(a_i) & \wedge & z_1 = \langle x_1, b_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1}\rangle & \wedge & x_2 \preceq^{\mathcal{A}} x_1 & \wedge \\ z_2 = m(a_{i+1}) & \wedge & z_2 = \langle x_2, d_{0..\mathsf{K}-1}, w_{0..\mathsf{K}-1}\rangle & \wedge & x_2 \neq x_1 & \end{array}$$

  Moreover, there exist paths q_1, q_2 in $\mathcal{A}_{\mathsf{path}}$ such that:

  $$append^{\mathcal{A}}(q_1, q_2, p) \ \wedge\ a_i \in path2set(q_1) \ \wedge\ a_{i+1} \in path2set(q_2)$$

  In particular, a_i is the last address in path q_1 and a_{i+1} in the first address in path q_2. Hence, by taking $c_1^{\mathcal{A}} = z_1$, $c_2^{\mathcal{A}} = z_2$, $e_1^{\mathcal{A}} = x_1$, $e_2^{\mathcal{A}} = x_2$, $p_1^{\mathcal{A}} = q_1$, $p_2^{\mathcal{A}} = q_2$, $n_1^{\mathcal{A}} = a_i$, $n_2^{\mathcal{A}} = a_{i+1}$ and

for all $0 \leq j < \mathsf{K}$ both, $b_j^{\mathcal{A}} = r_j$ and $d_j^{\mathcal{A}} = s_j$, it follows that:

$$\begin{array}{l}\Big(\exists_{\mathsf{addr}} n_1, n_2 \; \exists_{\mathsf{cell_K}} c_1, c_2 \; \exists_{\mathsf{elem}} e_1, e_2 \; \exists_{\mathsf{path}} p_1, p_2 \\ \exists_{\mathsf{addr}} b_{0..\mathsf{K}-1}, d_{0..\mathsf{K}-1} \; \exists_{\mathsf{tid}} t_{0..\mathsf{K}-1}, w_{0..\mathsf{K}-1}\Big) \\ \left(\begin{array}{lcll} \multicolumn{3}{l}{append(p_1, p_2, p)} & \wedge \\ n_1 & \in & path2set(p_1) & \wedge \\ n_2 & \in & path2set(p_2) & \wedge \\ c_1 & = & m[n_1] & \wedge \\ c_2 & = & m[n_2] & \wedge \\ c_1 & = & mkcell(e_1, b_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1}) & \wedge \\ c_2 & = & mkcell(e_2, d_{0..\mathsf{K}-1}, w_{0..\mathsf{K}-1}) & \wedge \\ e_2 & \preceq & e_1 & \wedge \\ e_2 & \neq & e_1 & \end{array}\right)\end{array}$$

holds in $\mathcal{A}$ as desired.

For the other direction, let $\mathcal{B}$ be a model of:

$$\Big(\exists_{\mathsf{addr}} n_1, n_2 \; \exists_{\mathsf{cell_K}} c_1, c_2 \; \exists_{\mathsf{elem}} e_1, e_2 \; \exists_{\mathsf{path}} p_1, p_2$$
$$\exists_{\mathsf{addr}} b_{0..\mathsf{K}-1}, d_{0..\mathsf{K}-1} \; \exists_{\mathsf{tid}} t_{0..\mathsf{K}-1}, w_{0..\mathsf{K}-1}\Big)$$

$$n_1 \in path2set(p_1) \quad \wedge \tag{7.27}$$
$$n_2 \in path2set(p_2) \quad \wedge \tag{7.28}$$
$$c_1 = m[n_1] \quad \wedge \tag{7.29}$$
$$c_2 = m[n_2] \quad \wedge \tag{7.30}$$
$$c_1 = mkcell(e_1, b_{0..\mathsf{K}-1}, t_{0..\mathsf{K}-1}) \quad \wedge \tag{7.31}$$
$$c_2 = mkcell(e_2, d_{0..\mathsf{K}-1}, w_{0..\mathsf{K}-1}) \quad \wedge \tag{7.32}$$
$$e_2 \preceq e_1 \quad \wedge \tag{7.33}$$
$$e_2 \neq e_1 \tag{7.34}$$

Due to (7.27) and (7.28), p cannot be ϵ. Moreover, conditions (7.27) to (7.34) imply that p is a path composed by at least two different addresses. Then, let $p = [x_1, \ldots, x_n]$ with $x_1, \ldots, x_n$ in $\mathcal{B}_{\mathsf{addr}}$. Conditions (7.27) to (7.34) imply that there are y_1 and y_2 in $x_1, \ldots, x_n$ such that

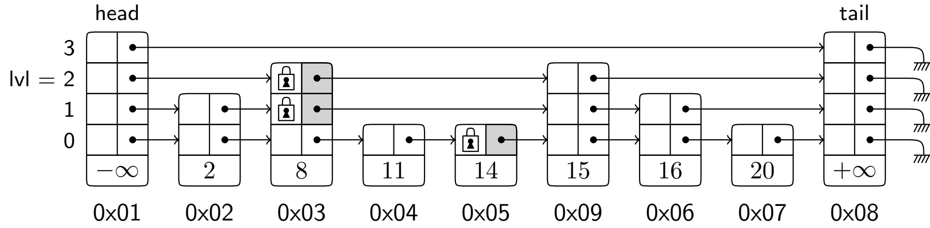
$$m(y_1).data^{\mathcal{B}} \not\preceq m(y_2).data^{\mathcal{B}}$$

Even more, y_1 occurs in path p before y_2. Hence, we can choose y_1 and y_2 from $x_1, \ldots, x_n$ to be consecutive. If y_1 and y_2 happen not to be consecutive, then there are $x_s, \ldots, x_t$ between y_1 and y_2. However, we can get rid of all the addresses in $x_s, \ldots, x_t$ as follows. If $m(y_1).data^{\mathcal{B}} \preceq m(x_s).data^{\mathcal{B}}$, then we can pick x_s as the new y_1 and proceed inductively with $x_{s+1}, \ldots, x_t$ addresses. On the other hand, if $m(y_1).data^{\mathcal{B}} \not\preceq m(x_s).data^{\mathcal{B}}$ then we can simply take x_s as y_2. Hence, we can conclude that $\neg ordPath^{\mathcal{B}}(m, p)$.

All (7.1)–(7.25) are valid $\mathsf{TSL_K}$ equivalences, which concludes the proof. □

Orienting the (7.1)–(7.25) equivalences from left to right allows us to eliminate non-normalized literals from a given $\mathsf{TSL_K}$ formula, resulting in a $\mathsf{TSL_K}$ formula that only contains normalized literals. Note also, that all quantifiers introduced in these equivalences are existential quantifiers, which can be pushed (with renaming to avoid capturing if necessary) to the front of the formula. Hence, a quantifier-free formula (which is implicitly existentially quantified) results into a quantifier-free formula after the rewriting step.

Example 7.3
Consider again this skiplist, introduced in Fig. 7.6(c):



Let $heap$ be a variable of sort $\mathsf{mem_K}$ representing the heap and let a_1, a_2 and a_3 be arbitrary variables of sort addr. Consider now the following predicate:

$$\neg ordPath(heap, [a_1, a_2, a_3]) \tag{7.35}$$

According to Definition 7.1, literal 7.35 is not normalized. In fact, according to Lemma 7.1, we can normalize this literal by translating it into the following formula:

$$\begin{array}{l} \Big(\exists_{\mathsf{addr}} x_1, x_2\ \exists_{\mathsf{cell_K}} c_1, c_2\ \exists_{\mathsf{elem}} e_1, e_2\ \exists_{\mathsf{path}} q_1, q_2, q_3, q_4, p_1, p_2 \\ \exists_{\mathsf{addr}} b_0, b_1, b_2, b_3\ \exists_{\mathsf{addr}} d_0, d_1, d_2, d_3\ \exists_{\mathsf{tid}} t_0, t_1, t_2, t_3\ \exists_{\mathsf{tid}} w_0, w_1, w_2, w_3\Big) \\ \left(\begin{array}{ll} q_1 = [a_1] \wedge q_2 = [a_2] \wedge q_3 = [a_3] \wedge append(q_1, q_2, q_4) \wedge append(q_4, q_3, p) & \wedge \\ append(p_1, p_2, p) \wedge x_1 \in path2set(p_1) \wedge x_2 \in path2set(p_2) & \wedge \\ c_1 = m[x_1] \wedge c_2 = m[x_2] \wedge e_2 \preceq e_1 \wedge e_2 \neq e_1 & \wedge \\ c_1 = mkcell(e_1, b_0, b_1, b_2, b_3, t_0, t_1, t_2, t_3) & \wedge \\ c_2 = mkcell(e_2, d_0, d_1, d_2, d_3, w_0, w_1, w_2, w_3) & \end{array}\right) \end{array} \tag{7.36}$$

Moreover, according to the interpretation $\mathcal{A}$ introduced in Example 8.3, there is an assignment to variables a_1, a_2 and a_3 which makes literal (7.35) (and thus formula (7.4)) satisfiable. Consider, for instance the assignment:

$$a_1 = \text{0x03} \quad a_2 = \text{0x06} \quad a_3 = \text{0x05}$$

The variables introduced by the existential quantifiers in (7.4) can be instantiated as follows:

$$\begin{array}{llll} x_1 = \text{0x06} & b_0 = \text{0x07} & d_0 = \text{0x09} & p = [\text{0x03}, \text{0x06}, \text{0x05}] \\ x_2 = \text{0x05} & b_1 = \text{0x08} & d_1 = \text{0x00} & p_1 = [\text{0x03}, \text{0x06}] \end{array}$$

$$\begin{array}{llll} e_1 = 16 & b_2 = \text{0x00} & d_2 = \text{0x00} & p_2 = [\text{0x05}] \\ e_2 = 14 & b_3 = \text{0x00} & d_3 = \text{0x00} & \\ t_0 = \emptyset & t_1 = t_2 = t_3 = \emptyset & & \\ w_0 = T_1 & w_1 = w_2 = w_3 = \emptyset & & \end{array}$$

forming a model of formula (7.4). ⌟

Following the result of Lemma 7.1, the resulting formula after normalization can then be converted into its disjunctive normal form, obtaining the following result.

Lemma 7.2 (Normalized Literals):
Every TSL_K-formula is equivalent to a disjunction of conjunctions of normalized TSL_K-literals. ⌟

The phase of normalizing a formula is commonly known (e.g. [177]) as the *variable abstraction phase*. Note that after the normalization process each normalized literal belongs to just one theory.

7.3 Decidability of TSL_K

As TL3, the theory TSL_K enjoys of the finite model property. In this section we prove that TSL_K does in fact enjoy the bounded model property presented in Definition 2.2 with respect to domains level_K, elem, addr and tid. Moreover, we show how to compute for a given TSL_K formula φ, a (polynomial) bound on the size of level_K, elem, addr and tid of a sufficiently large model. In other words, if there is no model within the bounds, then φ is unsatisfiable. As with TL3, note that a bound on the domain of the sorts level_K, elem, addr and tid is enough to also obtain bounds on the domains of the remaining sorts (cell_K, mem_K, path, setaddr, setelem and settid) because the domains of these latter sorts are constructed from the domains of level_K, elem, addr and tid due to the restrictions imposed in the class of models **TSLK**. Once again, these bounds imply that TSL_K is decidable because one can enumerate all Σ_{TSL_K}-structures up to the cardinality given by the bound of the finite model theorem, and check whether any of the structures is indeed a model of the given formula.

Consider an arbitrary TSL_K-interpretation $\mathcal{A}$ satisfying a conjunction of normalized TSL_K-literals Γ. We show that if $\mathcal{A}$ contains domains $\mathcal{A}_{\mathsf{level}_K}$, $\mathcal{A}_{\mathsf{elem}}$, $\mathcal{A}_{\mathsf{addr}}$ and $\mathcal{A}_{\mathsf{tid}}$ then there are finite sets $\mathcal{B}_{\mathsf{level}_K}$, $\mathcal{B}_{\mathsf{elem}}$, $\mathcal{B}_{\mathsf{addr}}$ and $\mathcal{B}_{\mathsf{tid}}$ with bounded cardinalities, where the finite bound on the sizes can be computed from Γ. These sets can in turn be used to obtain a finite interpretation $\mathcal{B}$ satisfying Γ, since all the other sorts are bounded by the sizes of these sets.

7.3.1 Auxiliary Functions for Model Transformation

Before proceeding with the proof that TSL_K enjoys the bounded model property, we first define some auxiliary functions.

Definition 7.2 ($\mathsf{TSL_K}$ Many Jumps).
Given a memory m, an address a and a level l, we use the following notation for $s \geq 1$:

$$m^{[s]}(a).next[l] = \begin{cases} m(a).next[l] & \text{if } s = 1 \\ m(m^{[s-1]}(a).next[l]).next[l] & \text{if } s > 1 \end{cases}$$

This definition is a generalization of Definition 6.2 for K levels. ⌟

We start by defining the function $\mathit{first}_\mathsf{K}$. This function returns the first *relevant* element of the domain of addresses at a given level which is necessary to maintain the valuation of all functions and predicates. Later, all irrelevant elements will be removed from the large domain to obtain the bounded domain.

Let $X \subseteq \mathcal{A}_\mathsf{addr}$, $m : \mathcal{A}_\mathsf{addr} \to \mathcal{A}_\mathsf{elem} \times \mathcal{A}_\mathsf{addr}^\mathsf{K} \times \mathcal{A}_\mathsf{tid}^\mathsf{K}$, let a be an address in X and let l be a level in $\mathcal{A}_{\mathsf{level_K}}$. Then, the function $\mathit{first}_\mathsf{K}(m, a, l, X)$ is defined as follows:

$$\mathit{first}_\mathsf{K}(m, a, l, X) = \begin{cases} \mathit{null} & \text{if for all } r \geq 1 \\ & \quad m^{[r]}(a).next[l] \notin X \\ & \\ m^{[s]}(a).next[l] & \text{if for some } s \geq 1 \\ & \quad m^{[s]}(a).next[l] \in X, \\ & \text{and for all } r < s \\ & \quad m^{[r]}(a).next[l] \notin X \end{cases}$$

Basically, given a set of addresses X, $\mathit{first}_\mathsf{K}$ chooses the next address in X that can be reached from a given address by repeatedly following the $next[l]$ pointer at a given level l. We will build later a small model by filtering out unnecessary intermediate nodes and use $\mathit{first}_\mathsf{K}$ to bypass properly all removed nodes, preserving the important connectivity properties.

Lemma 7.3 (Function $\mathit{first}_\mathsf{K}$):
Let $X \subseteq \mathcal{A}_\mathsf{addr}$, $m_1, m_2 : \mathcal{A}_\mathsf{addr} \to \mathcal{A}_\mathsf{elem} \times \mathcal{A}_\mathsf{addr}^K \times \mathcal{A}_\mathsf{tid}^K$, $a_1, a_2 \in X$ such that $a_1 \neq a_2$, $c \in \mathcal{A}_\mathsf{elem} \times X^K \times \mathcal{A}_\mathsf{tid}^K$ and $l \in \mathcal{A}_{\mathsf{level_K}}$. Then:

(a) If $m_1(a_1).next[l] \in X$, then $\mathit{first}_K(m_1, a_1, l, X) = m_1(a_1).next[l]$.

(b) If $m_1 = upd(m_2, a_1, c)$, then $\mathit{first}_K(m_1, a_2, l, X) = \mathit{first}_K(m_2, a_2, l, X)$. ⌟

Proof. We proof (a) and (b) separately.

(a) Immediate from the definition of $\mathit{first}_\mathsf{K}$.

(b) Let $m_1 = upd(m_2, a_1, c)$. We consider two possible cases:

(1) $m_1^{[r]}(a_2).next[l] \notin X$ for all $r \geq 1$. By induction it can be shown that $m_1^{[r]}(a_2) = m_2^{[r]}(a_2)$ for each $r \geq 1$. It follows that $\mathit{first}_\mathsf{K}(m_1, a_2, l, X) = \mathit{null} = \mathit{first}_\mathsf{K}(m_2, a_2, l, X)$.

(2) $m_1^{[s]}(a_2).next[l] \in X$ for some $s \geq 1$. In this case, assume without loss of generality that $\mathit{first}_{\mathsf{K}}(m_1, a_2, l, X) = m_1^{[s]}(a_2).next[l]$. By induction it can be shown that $m_1^{[r]}(a_2) = m_2^{[r]}(a_2)$ for each $1 \leq r < s$. It follows that $\mathit{first}_{\mathsf{K}}(m_1, a_2, l, X) = m_1^{[s]}(a_2).next[l] = m_2^{[s]}(a_2).next[l] = \mathit{first}_{\mathsf{K}}(m_2, a_2, l, X)$. □

Now, we define the function $\mathit{unordered}$ which given a memory m and a path p, returns a set containing two addresses within p that witness the failure to preserve the $data$ order in p:

$$\mathit{unordered}(m, [i_1, \ldots, i_n]) \quad = \quad \begin{cases} \emptyset & \text{if } n = 0 \text{ or } n = 1 \\ \{i_1, i_2\} & \text{if } m(i_2).data \preceq m(i_1).data \\ & \text{ and } m(i_2).data \neq m(i_1).data \\ \mathit{unordered}(m, [i_2, \ldots, i_n]) & \text{otherwise} \end{cases}$$

If there exist two addresses that witness an order violation, then $\mathit{unordered}$ returns the first two consecutive addresses whose keys violate the order.

Lemma 7.4 (Function $\mathit{unordered}$):
Let p be a path such that $p = [a_1, \ldots, a_n]$ with $n \geq 2$ and let m be a memory. If there is a_i with $1 \leq i < n$, such that $m(a_{i+1}).data \preceq m(a_i).data$ and $m(a_{i+1}).data \neq m(a_i).data$, then $\mathit{unordered}(m, p) \neq \emptyset$. ⌟

Proof. We proceed by induction. Consider $n = 2$ and let $p = [a_1, a_2]$ such that $m(a_2).data \preceq m(a_1).data$ and $m(a_2).data \neq m(a_1).data$. Then, by definition of $\mathit{unordered}$, we have that $\mathit{unordered}(m, p) = \{a_1, a_2\} \neq \emptyset$.

Now assume $n > 2$ and let $p = [a_1, \ldots, a_n]$. If $m(a_2).data \preceq m(a_1).data$ and $m(a_2).data \neq m(a_1).data$, then $\mathit{unordered}(m, p) = \{a_1, a_2\} \neq \emptyset$. On the other hand, if $m(a_1).data \preceq m(a_2).data$, we know that there is a a_i, with $2 \leq i < n$, with $m(a_{i+1}).data \preceq m(a_i).data$ and $m(a_{i+1}).data \neq m(a_i).data$. Therefore, by induction we have that $\mathit{unordered}(m, [a_2, \ldots, a_n]) \neq \emptyset$ and by definition of $\mathit{unordered}$:

$$\mathit{unordered}(m, p) \;=\; \mathit{unordered}(m, [a_2, \ldots, a_n]) \;\neq\; \emptyset$$ □

Throughout the proof that $\mathsf{TSL_K}$ enjoys of the bounded model property, we will also require the use of the following auxiliary functions:

- $\mathit{compress}$: given a path p and a set X of addresses, $\mathit{compress}$ filters out all addresses in p which do not belong to set X.
- diseq: given two paths p and q, diseq returns a set of addresses which is responsible for the inequality between q and p.
- common: given two paths p and q, common returns an address which is common to both p and q.

All these auxiliary functions were already defined in Section 6.3.1.

7.3.2 A Bounded Model Theorem for $\mathsf{TSL_K}$

The following theorem establishes the existence of a small model with bounded domains whenever a $\mathsf{TSL_K}$ formula is satisfiable.

Theorem 7.1 ($\mathsf{TSL_K}$ Bounded Model Property):
Let Γ be a conjunction of normalized TSL_K-literals. Let $\mathsf{E} = |V_{\mathsf{elem}}(\Gamma)|$, $\mathsf{A} = |V_{\mathsf{addr}}(\Gamma)|$, $\mathsf{M} = |V_{\mathsf{mem_K}}(\Gamma)|$ *and,* $\mathsf{P} = |V_{\mathsf{path}}(\Gamma)|$. *Then the following are equivalent:*

1. Γ *is TSL_K-satisfiable.*
2. Γ *is true in a TSL_K interpretation $\mathcal{B}$ such that*

$$
\begin{aligned}
|\mathcal{B}_{\mathsf{addr}}| &\leq \mathsf{A} + 1 + K \times \mathsf{M} \times \mathsf{A} + 2 \times \mathsf{P}^2 + 2 \times \mathsf{P}^3 + 2 \times \mathsf{M} \times \mathsf{P} \\
|\mathcal{B}_{\mathsf{elem}}| &\leq \mathsf{E} + \mathsf{M} \times |\mathcal{B}_{\mathsf{addr}}| \\
|\mathcal{B}_{\mathsf{tid}}| &\leq \mathsf{T} + K \times \mathsf{M} \times |\mathcal{B}_{\mathsf{addr}}| + 1
\end{aligned}
$$

┘

Proof. We show both directions separately.

$(2 \to 1)$ is immediate, because a bounded model is a model, so the formula is satisfiable.

We now consider the other direction: $(1 \to 2)$. Let $\mathcal{A}$ be a $\mathsf{TSL_K}$-interpretation satisfying a set of normalized $\mathsf{TSL_K}$-literals Γ. We use $\mathcal{A}$ to construct a $\mathsf{TSL_K}$-interpretation $\mathcal{B}$ which satisfies Γ, using the auxiliary definitions introduced above, and show that the domains in $\mathcal{B}$ are bounded as in the statement of the theorem. In particular, for each domain we have:

- In $\mathcal{B}$, $\mathcal{B}_{\mathsf{level_K}} = \mathcal{A}_{\mathsf{level_K}} = [0 \ldots K-1]$
- The domain $\mathcal{B}_{\mathsf{addr}}$ for sort addr is defined as follows:

$$
\begin{aligned}
\mathcal{B}_{\mathsf{addr}} = \; & V_{\mathsf{addr}}^{\mathcal{A}} \cup \left\{ null^{\mathcal{A}} \right\} \cup & (7.37)\\
& \left\{ m^{\mathcal{A}}(a^{\mathcal{A}}).next^{\mathcal{A}}[l] \mid \right. \\
& \qquad \left. \text{for every } m \in V_{\mathsf{mem_K}}, a \in V_{\mathsf{addr}} \text{ and } l \in \mathcal{B}_{\mathsf{level_K}} \right\} \cup & (7.38)\\
& \left\{ diseq(p^{\mathcal{A}}, q^{\mathcal{A}}) \mid \text{for each literal } p \neq q \text{ in } \Gamma \right\} \cup & (7.39)\\
& \left\{ common(p_1{}^{\mathcal{A}}, p_2{}^{\mathcal{A}}) \mid \right. \\
& \qquad \text{for each literal } \neg append(p_1, p_2, p_3) \text{ in } \Gamma \text{ with} \\
& \qquad \left. path2set^{\mathcal{A}}(p_1{}^{\mathcal{A}}) \cap path2set^{\mathcal{A}}(p_2{}^{\mathcal{A}}) \neq \emptyset \right\} \cup & (7.40)\\
& \left\{ common(p_1{}^{\mathcal{A}} \circ p_2{}^{\mathcal{A}}, p_3{}^{\mathcal{A}}) \mid \right. \\
& \qquad \text{for each literal } \neg append(p_1, p_2, p_3) \text{ in } \Gamma \text{ with} \\
& \qquad \left. path2set^{\mathcal{A}}(p_1{}^{\mathcal{A}}) \cap path2set^{\mathcal{A}}(p_2{}^{\mathcal{A}}) = \emptyset \right\} \cup & (7.41)\\
& \left\{ unordered(m^{\mathcal{A}}, p^{\mathcal{A}}) \mid \text{for each literal } \neg ordPath(m, p) \text{ in } \Gamma \right\} & (7.42)
\end{aligned}
$$

Essentially, $\mathcal{B}_{\mathsf{addr}}$ is a subset of $\mathcal{A}_{\mathsf{addr}}$ where the following addresses are preserved in the domain of sort addr:

- Because of (7.37), all the addresses in the domain $\mathcal{A}_{\mathsf{addr}}$ that correspond to variables are preserved in $\mathcal{B}_{\mathsf{addr}}$. There are at most A of these addresses. The element modeling $null$ in $\mathcal{A}_{\mathsf{addr}}$ is also kept.
- According to (7.38) for each variable of sort address, each variable of sort memory and each level, the address accessible through $next$ is also preserved. There are at most $\mathsf{K} \times \mathsf{M} \times \mathsf{A}$ of these addresses.
- (7.39) ensures that one or two addresses (as returned by $diseq$) are kept for every literal of the form $p \neq q$. Since there are at most P paths in Γ, there are at most P^2 literals of the form $p \neq q$, which results in preserving at most $2 \times \mathsf{P}^2$ addresses.
- There are two reasons that can explain why a literal $(\neg append(p_1, p_2, p_3)$ holds in $\mathcal{A}$. The first one is when both paths p_1 and p_2 share addresses. In this case, their concatenation is not a legal path, because legal paths cannot contain repeated addresses, so (7.40) keeps a common address present in both paths to witness this sharing. In the second case, paths p_1 and p_2 do not share any address, so their concatenation is a legal path, but this resulting path is not equal to p_3. In this case, (7.41) keeps one or two addresses as returned by $diseq$ to witness this difference. Since there are P paths in Γ there are at most P^3 literals of the form $\neg append(p_1, p_2, p_3)$ in Γ, which results in at most $2 \times \mathsf{P}^3$ addresses preserved.
- Finally, if a literal $(\neg ordPath(m, p))$ holds there exist two addresses in path p whose values, according to the map provided by memory m, are not ordered. Since there are at most M variables of sort mem_K and P variables of sort path, then we may require at most $2 \times \mathsf{M} \times \mathsf{P}$ addresses to preserve literal $(\neg ordPath(m, p))$. These required addresses are considered by (7.42).

- The domain $\mathcal{B}_{\mathsf{elem}}$ for sorts elem is as follows:

$$\mathcal{B}_{\mathsf{elem}} = V^{\mathcal{A}}_{\mathsf{elem}} \cup \left\{ m^{\mathcal{A}}(v).data^{\mathcal{A}} \mid m \in V_{\mathsf{mem}_\mathsf{K}} \text{ and } v \in \mathcal{B}_{\mathsf{addr}} \right\}$$

  The domain $\mathcal{B}_{\mathsf{elem}}$ is built from $\mathcal{A}_{\mathsf{tid}}$ by simply keeping interpretations of variables of sort tid and the values stored at addresses that have already been kept in $\mathcal{B}_{\mathsf{addr}}$.

- Finally, the domain $\mathcal{B}_{\mathsf{tid}}$ for sort tid is obtained as follows:

$$\mathcal{B}_{\mathsf{tid}} = V^{\mathcal{A}}_{\mathsf{tid}} \cup \left\{ m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}[l] \mid m \in V_{\mathsf{mem}_\mathsf{K}}, a \in V_{\mathsf{addr}} \text{ and } l \in \mathcal{B}_{\mathsf{level}_\mathsf{K}} \right\} \cup \left\{ \oslash \right\}$$

  As can be seen, the domain $\mathcal{B}_{\mathsf{tid}}$ is constructed from $\mathcal{A}_{\mathsf{elem}}$ by keeping interpretations of variables of sort elem, the thread identifiers used at cells pointed by addresses that has already been kept in $\mathcal{B}_{\mathsf{addr}}$ and finally $\oslash$ to represent the absent of a thread identifier.

The domains $\mathcal{B}_{\mathsf{level}_\mathsf{K}}$, $\mathcal{B}_{\mathsf{elem}}$, $\mathcal{B}_{\mathsf{addr}}$ and $\mathcal{B}_{\mathsf{tid}}$ described above clearly satisfy the cardinality constraints expressed in Theorem 7.1. The interpretations of the rest of the domains are obtained using the restrictions of **TSLK**, shown in Section 7.2.3:

- $\mathcal{B}_{\mathsf{cell}_\mathsf{K}} = \mathcal{B}_{\mathsf{elem}} \times \mathcal{B}_{\mathsf{addr}}^{\mathsf{K}} \times \mathcal{B}_{\mathsf{tid}}^{\mathsf{K}}$.
- $\mathcal{B}_{\mathsf{mem}_\mathsf{K}} = \mathcal{B}_{\mathsf{cell}_\mathsf{K}}^{\mathcal{B}_{\mathsf{addr}}}$.
- $\mathcal{B}_{\mathsf{path}}$ is the set of all finite sequences of (pairwise) distinct elements of $\mathcal{B}_{\mathsf{addr}}$.
- $\mathcal{B}_{\mathsf{setaddr}}$ is the power-set of $\mathcal{B}_{\mathsf{addr}}$.
- $\mathcal{B}_{\mathsf{settid}}$ is the power-set of $\mathcal{B}_{\mathsf{tid}}$.
- $\mathcal{B}_{\mathsf{setelem}}$ is the power-set of $\mathcal{B}_{\mathsf{elem}}$.

All these domains are finite, because $\mathcal{B}_{\mathsf{level}_\mathsf{K}}$, $\mathcal{B}_{\mathsf{elem}}$, $\mathcal{B}_{\mathsf{addr}}$ and $\mathcal{B}_{\mathsf{tid}}$ are finite.

We are left to show the interpretation of all variables and function symbols, and to prove that $\mathcal{B}$ is a model of Γ. The interpretation of variables and symbols in $\mathcal{B}$ is:

$$
\begin{array}{rcll}
error^{\mathcal{B}} & = & error^{\mathcal{A}} & \\
null^{\mathcal{B}} & = & null^{\mathcal{A}} & \\
l^{\mathcal{B}} & = & l^{\mathcal{A}} & \text{for each } l \in V_{\mathsf{level}_\mathsf{K}} \\
e^{\mathcal{B}} & = & e^{\mathcal{A}} & \text{for each } e \in V_{\mathsf{elem}} \\
a^{\mathcal{B}} & = & a^{\mathcal{A}} & \text{for each } a \in V_{\mathsf{addr}} \\
t^{\mathcal{B}} & = & t^{\mathcal{A}} & \text{for each } t \in V_{\mathsf{tid}} \\
c^{\mathcal{B}} & = & c^{\mathcal{A}} & \text{for each } c \in V_{\mathsf{cell}_\mathsf{K}} \\
m^{\mathcal{B}}(v) & = & \left(\begin{array}{ll}
m^{\mathcal{A}}(v).data^{\mathcal{A}} & , \\
\mathit{first}_\mathsf{K}(m^{\mathcal{A}}, v, 0, \mathcal{B}_{\mathsf{addr}}) & , \\
\quad \ldots & , \\
\mathit{first}_\mathsf{K}(m^{\mathcal{A}}, v, \mathsf{K}-1, \mathcal{B}_{\mathsf{addr}}) & , \\
m^{\mathcal{A}}(v).lockid^{\mathcal{A}}[0] & , \\
\quad \ldots & , \\
m^{\mathcal{A}}(v).lockid^{\mathcal{A}}[\mathsf{K}-1] &
\end{array}\right) & \text{for each } m \in V_{\mathsf{mem}_\mathsf{K}} \text{ and } v \in \mathcal{B}_{\mathsf{addr}} \\
s^{\mathcal{B}} & = & s^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}} & \text{for each } s \in V_{\mathsf{setaddr}} \\
r^{\mathcal{B}} & = & r^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{tid}} & \text{for each } r \in V_{\mathsf{settid}} \\
x^{\mathcal{B}} & = & x^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{elem}} & \text{for each } x \in V_{\mathsf{setelem}} \\
p^{\mathcal{B}} & = & compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) & \text{for each } p \in V_{\mathsf{path}}
\end{array}
$$

All variables and constants in $\mathcal{B}$ are interpreted as in $\mathcal{A}$ except that $next$ pointers use $\mathit{first}_\mathsf{K}$ to point to the next reachable address at each level that has been preserved in $\mathcal{B}_{\mathsf{addr}}$. Similarly, sets of addresses, elements and thread identifiers are pruned to contain only elements kept in $\mathcal{B}_{\mathsf{addr}}$, $\mathcal{B}_{\mathsf{elem}}$ and $\mathcal{B}_{\mathsf{tid}}$ respectively. Finally, paths are filtered out so that they contain only addresses in $\mathcal{B}_{\mathsf{addr}}$ using the function $compress$. It is easy to check that $\mathcal{B}$ is an interpretation of Γ and hence a candidate model of Γ.

So, it just remains to be seen that $\mathcal{B}$ satisfies all literals in Γ assuming that $\mathcal{A}$ does, which let us conclude that $\mathcal{B}$ is indeed a model of Γ. We reason by cases considering all possible literals:

Literals of the form $e_1 \neq e_2$**,** $a_1 \neq a_2$**,** $l_1 \neq l_2$**,** $k_1 \neq k_2$**,** $t_1 \neq t_2$**,** $l_1 < l_2$ **and** $e_1 \preceq e_2$**:**

Immediate, because the interpretation of all variables is preserved from $\mathcal{A}$ into $\mathcal{B}$. For instance, consider the case of the literal $l_1 < l_2$. Since the valuation for variables is preserved, $l_1^{\mathcal{B}} = l_1^{\mathcal{A}}$ and $l_2^{\mathcal{B}} = l_2^{\mathcal{A}}$. Hence $l_1 < l_2$ holds in $\mathcal{B}$ because it holds in $\mathcal{A}$.

Literals of the form $a = null$ **and** $c = error$**:**

Immediate, following a reasoning similar to the previous case.

Literals of the form $c = mkcell(e, a_0, \ldots, a_{\mathsf{K}-1}, t_0, \ldots, t_{\mathsf{K}-1})$ **:**

$$\begin{aligned} c^{\mathcal{B}} &= c^{\mathcal{A}} \\ &= \left(e^{\mathcal{A}}, a_0{}^{\mathcal{A}}, \ldots, a_{\mathsf{K}-1}{}^{\mathcal{A}}, t_0{}^{\mathcal{A}}, \ldots, t_{\mathsf{K}-1}{}^{\mathcal{A}}\right) \\ &= \left(e^{\mathcal{B}}, a_0{}^{\mathcal{B}}, \ldots, a_{\mathsf{K}-1}{}^{\mathcal{B}} t_0{}^{\mathcal{B}}, \ldots, t_{\mathsf{K}-1}{}^{\mathcal{B}}\right) \end{aligned}$$

Note that for all $0 \leq i < \mathsf{K}$, $a_i{}^{\mathcal{A}} = a_i{}^{\mathcal{B}}$ because a_i occurs in Γ and $a_i \in V_{\mathsf{addr}}^{\mathcal{A}}$. Then, due to (7.37) from Theorem 7.1 we know that the interpretation of all variables of sort addr in $\mathcal{A}$ are also in $\mathcal{B}$. This implies that $a_i{}^{\mathcal{B}} = a_i{}^{\mathcal{A}}$.

Literals of the form $c = m[a]$**:**

$$\begin{aligned} \Big(m[a]\Big)^{\mathcal{B}} &= m^{\mathcal{B}}(a^{\mathcal{B}}) \\ &= m^{\mathcal{B}}(a^{\mathcal{A}}) \\ &= \Big(m^{\mathcal{A}}(a^{\mathcal{A}}).data^{\mathcal{A}}, \\ &\quad first_{\mathsf{K}}(m^{\mathcal{A}}, a^{\mathcal{A}}, 0, \mathcal{B}_{\mathsf{addr}}), \ldots, first_{\mathsf{K}}(m^{\mathcal{A}}, a^{\mathcal{A}}, \mathsf{K}-1, \mathcal{B}_{\mathsf{addr}}), \\ &\quad m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}(0), \ldots, m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}(\mathsf{K}-1)\Big) \\ &= \Big(m^{\mathcal{A}}(a^{\mathcal{A}}).data^{\mathcal{A}}, \\ &\quad m^{\mathcal{A}}(a^{\mathcal{A}}).next^{\mathcal{A}}(0), \ldots, m^{\mathcal{A}}(a^{\mathcal{A}}).next^{\mathcal{A}}(\mathsf{K}-1), \qquad (7.43) \\ &\quad m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}(0), \ldots, m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}(\mathsf{K}-1)\Big) \\ &= m^{\mathcal{A}}(a^{\mathcal{A}}) \\ &= c^{\mathcal{A}} \\ &= c^{\mathcal{B}} \end{aligned}$$

where step (7.43) is justified by Lemma 7.3(a) and (7.38) from the proof of Theorem 7.1.

Literals of the form $m = upd(\tilde{m}, a, c)$**:**

We want to prove that $m^{\mathcal{B}} = \tilde{m}^{\mathcal{B}}_{a^{\mathcal{B}} \mapsto c^{\mathcal{B}}}$. We consider two cases:

- $m(a)$: In this case,

$$m^{\mathcal{B}}(a^{\mathcal{B}}) \;=\; m^{\mathcal{A}}(a^{\mathcal{A}}) \;=\; c^{\mathcal{A}} \;=\; c^{\mathcal{B}}$$

- $m(d)$ for $d \neq a$: In this other case,

$$\begin{aligned} m^{\mathcal{B}}(d) &= \Big(m^{\mathcal{A}}(d).data^{\mathcal{A}}, \\ &\quad first_{\mathsf{K}}(m^{\mathcal{A}}, d, 0, \mathcal{B}_{\mathsf{addr}}), \ldots, first_{\mathsf{K}}(m^{\mathcal{A}}, d, \mathsf{K}-1, \mathcal{B}_{\mathsf{addr}}) \\ &\quad m^{\mathcal{A}}.lockid^{\mathcal{A}}(0), m^{\mathcal{A}}.lockid^{\mathcal{A}}(\mathsf{K}-1) \Big) \\ &= \Big(\tilde{m}^{\mathcal{A}}(d).data^{\mathcal{A}}, \\ &\quad first_{\mathsf{K}}(\tilde{m}^{\mathcal{A}}, d, 0, \mathcal{B}_{\mathsf{addr}}), \ldots, first_{\mathsf{K}}(\tilde{m}^{\mathcal{A}}, d, \mathsf{K}-1, \mathcal{B}_{\mathsf{addr}}) \quad (7.44) \\ &\quad m^{\mathcal{A}}.lockid^{\mathcal{A}}(0), m^{\mathcal{A}}.lockid^{\mathcal{A}}(\mathsf{K}-1) \Big) \\ &= \tilde{m}^{\mathcal{B}}(d) \end{aligned}$$

Where step (7.44) is justified by Lemma 7.3(b).

Literals of the form $s = \{a\}$, $s_1 = s_2 \cup s_3$ and $s_1 = s_2 \setminus s_3$:
Analogous to the normalization of literals for sets of addresses presented in the proof of Theorem 6.1.

Literals of the form $x = \{e\}_{\mathtt{E}}$, $x_1 = x_2 \cup_{\mathtt{E}} x_3$ and $x_1 = x_2 \setminus_{\mathtt{E}} x_3$:
Analogous to the normalization of literals for sets of elements presented in the proof of Theorem 6.1.

Literals of the form $r = \{t\}_{\mathtt{T}}$, $r_1 = r_2 \cup_{\mathtt{T}} r_3$ and $r_1 = r_2 \setminus_{\mathtt{T}} r_3$:
Analogous to the normalization of literals for sets of thread identifiers presented in the proof of Theorem 6.1.

Literals of the form $p_1 \neq p_2$, $p = [a]$, $s = path2set(p)$, $append(p_1, p_2, p_3)$ and $\neg append(p_1, p_2, p_3)$:
Analogous to the proof of literals $(p_1 \neq p_2)$, $(p = [a])$, $(s = path2set(p))$, $(append(p_1, p_2, p_3))$ and $(\neg append(p_1, p_2, p_3))$ presented in the proof of Theorem 6.1.

Literals of the form $s = addr2set_{\mathsf{K}}(m, a, l)$:
The proof is analogous to the proof of $addr2set_{\mathsf{K}}(m, a, l)$ presented in Theorem 6.1, with the difference that now, for the $\mathsf{TSL_K}$ literal $addr2set_{\mathsf{K}}$, we need to reason about levels.

Literals of the form $p = getp_{\mathsf{K}}(m, a, b, l)$:
The proof is analogous to the proof of $getp(m, a, b)$ presented in Theorem 6.1, with the difference that now, for the $\mathsf{TSL_K}$ literal $getp_{\mathsf{K}}$, we need to reason about levels.

Literals of the form $ordPath(m, p)$:
Assume that $(m^{\mathcal{A}}, p^{\mathcal{A}}) \in ordPath^{\mathcal{A}}$. We need to show that $(m^{\mathcal{B}}, p^{\mathcal{B}}) \in ordPath^{\mathcal{B}}$, that is $(m^{\mathcal{B}}, compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) \in ordPath^{\mathcal{B}}$. We proceed by induction on p.

- If $p = \epsilon$, by definition of $compress$ and $ordPath$, we have that $(m^{\mathcal{B}}, \epsilon^{\mathcal{B}}) \in ordPath^{\mathcal{B}}$.
- If $p = [a_1]$, we know that $(m^{\mathcal{A}}, [a_1]^{\mathcal{A}}) \in ordPath^{\mathcal{A}}$ and therefore we can conclude that $p^{\mathcal{B}} = compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$. Then, if $a_1^{\mathcal{A}} \in \mathcal{B}_{\mathsf{addr}}$, we have that $p^{\mathcal{B}} = [a_1]^{\mathcal{B}}$ and then $(m^{\mathcal{B}}, p^{\mathcal{B}}) \in ordPath^{\mathcal{B}}$ holds. On the other hand, if $a_1^{\mathcal{A}} \notin \mathcal{B}_{\mathsf{addr}}$, then $p^{\mathcal{B}} = \epsilon^{\mathcal{B}}$ and once more $(m^{\mathcal{B}}, p^{\mathcal{B}}) \in ordPath^{\mathcal{B}}$ holds.

- If $p = [a_1, \ldots, a_{n+1}]$ with $n \geq 1$, then we have two possible cases:

  (1) $a_1^{\mathcal{A}} \notin \mathcal{B}_{\mathsf{addr}}$. In this case:

$$compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) = compress([a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})$$

  and by induction:

$$(m^{\mathcal{B}}, compress([a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) \in ordPath^{\mathcal{B}}$$

  we conclude that $(m^{\mathcal{B}}, compress([a_1, a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) \in ordPath^{\mathcal{B}}$.

  (2) $a_1^{\mathcal{A}} \in \mathcal{B}_{\mathsf{addr}}$. In this case, by induction:

$$(m^{\mathcal{B}}, compress([a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) \in ordPath^{\mathcal{B}}$$

  Moreover, since $m^{\mathcal{A}}(a_1^{\mathcal{A}}).data^{\mathcal{A}} \preceq m^{\mathcal{A}}(a_2^{\mathcal{A}}).data^{\mathcal{A}}$ then $m^{\mathcal{B}}(a_1^{\mathcal{A}}).data^{\mathcal{B}} \preceq m^{\mathcal{B}}(a_2^{\mathcal{A}}).data^{\mathcal{B}}$. Hence,

$$(m^{\mathcal{B}}, compress([a_1, a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}})) \in ordPath^{\mathcal{B}}$$

Since the value of each normalized literal is preserved in $\mathcal{B}$ with respect to $\mathcal{A}$, then $\mathcal{B}$ is a model of Γ that satisfies the cardinality constraints. □

Example 7.4
Consider the following $\mathsf{TSL_K}$ formula, introduced in Example 7.3, which corresponds to the normalization of the $\mathsf{TSL_K}$ formula $(\neg ordPath(heap, [a_1, a_2, a_3]))$:

$$\begin{array}{l}
\exists_{\mathsf{addr}} x_1, x_2\ \exists_{\mathsf{cell_K}} c_1, c_2\ \exists_{\mathsf{elem}} e_1, e_2\ \exists_{\mathsf{path}} q_1, q_2, q_3, q_4, p_1, p_2 \\
\exists_{\mathsf{addr}} b_0, b_1, b_2, b_3\ \exists_{\mathsf{addr}} d_0, d_1, d_2, d_3\ \exists_{\mathsf{tid}} t_0, t_1, t_2, t_3\ \exists_{\mathsf{tid}} w_0, w_1, w_2, w_3 \Big) \\
\left(\begin{array}{ll}
q_1 = [a_1] \wedge q_2 = [a_2] \wedge q_3 = [a_3] \wedge append(q_1, q_2, q_4) \wedge append(q_4, q_3, p) & \wedge \\
append(p_1, p_2, p) \wedge x_1 \in path2set(p_1) \wedge x_2 \in path2set(p_2) & \wedge \\
c_1 = m[x_1] \wedge c_2 = m[x_2] \wedge e_2 \preceq e_1 \wedge e_2 \neq e_1 & \wedge \\
c_1 = mkcell(e_1, b_0, b_1, b_2, b_3, t_0, t_1, t_2, t_3) & \wedge \\
c_2 = mkcell(e_2, d_0, d_1, d_2, d_3, w_0, w_1, w_2, w_3) &
\end{array}\right)
\end{array}$$

The result of Theorem 7.1 establishes that we can compute bounds for the domains of levels, elements, ordered keys, addresses and thread identifiers such that if the formula is satisfiable, then there exists a $\mathsf{TSL_K}$ model with it domains within these bounds. According to Theorem 7.1 we require:

Levels: All levels are preserved.

Addresses: We would require at most 582 addresses. This result comes from the following calculation:

- 13, due to the number of variables of sort address present in the formula;

- 1, for the special $null$ address;
- 52, considering the $next$ pointers at all 4 possible skiplist levels ($4 \times 1 \times 13$);
- 504, due to the possible inequality between paths (2×6^2 and 2×6^3 respectively);
- 12, because of the possible existence of negations of predicates $append$.

Elements: We would require a domain with a maximum number of 584 elements as we need 2 elements because of the 2 variables of sort element and 582 elements due to the elements that may be stored at each cell pointed by the addresses in the domain of addresses (1×582).

Thread identifiers: We would require a domain with a maximum of 2333 thread identifiers. We come to this conclusion considering:

- 4 thread identifiers because of the variables of sort tid appearing in the formula.
- 2328 thread identifiers in case there exists the need to lock all cells pointed by addresses in the computed domain at all possible levels ($4 \times 1 \times 582$).
- 1 special thread identifier to represent the special identifier $\oslash$.

Once again, the bounds computed by Theorem 7.1 are theoretical. In practice, it would be very inefficient to analyze all possible models with a domain of 574 addresses, for instance. As before, some considerations can be taken into account in order to tighten the bounds of the computed domains. For example, in formula 7.4 there are neither path inequalities nor occurrences of negation of the $append$ predicate. This allows to discard 516 addresses. Moreover, if we are trying to check the satisfiability of literal $(\neg ordPath(heap, [a_1, a_2, a_3]))$ of a skiplist like the one depicted in Fig. 7.6(b) we can use, for instance, variable propagation in order to reduce the number of needed variables. For example, we know that b_2, b_3, d_1, d_2 and d_3 point to the same element $null$, hence we can simply replace the occurrences of these variables with $null$ without affecting the satisfiability of the formula. This way we need to consider only 8 variables of sort address, which reduces the maximum size of the required address domain to 61.

Later, in Section 9.2.4, we will describe in more detail other tactics and considerations that help in reducing the size of the computed domains. ⌟

7.4 Summary

In this chapter we presented TSL$_K$, the *Family of Theories of Concurrent Skiplists with at Most K Levels*. Each member of the TSL$_K$ family is a theory capable of reasoning about concurrent skiplists with a fixed number of levels.

As for TL3, TSL$_K$ is obtained as a combination of theories of addresses, elements, cells, heaps and paths among others. In fact, TSL$_K$ extends non-trivially TL3 by adding functions and predicates of single-linked list to all levels of a skiplist. Moreover, TSL$_K$ introduces built-in predicates to reason about order of single-linked lists.

Each theory within TSL$_K$ is suitable for describing structural and functional properties of concurrent and non concurrent implementations of skiplists with a bounded and fixed number of levels.

In this chapter we showed that the satisfiability problem for TSL_K quantifier-free formulas is decidable. We presented a bounded model theorem, which ensures that given a quantifier-free TSL_K formula, it is possible to compute the bounds of the domains for a model of the formula (if such model exists) considering only the literals occurring in the formula. Following this result, the TSL_K decision procedure can determine the satisfiability of a TSL_K formula by enumerating all possible models of the given formula.

The TSL_K family remains crucial in the verification of parametrized programs that manipulates data structures of the shape of a skiplist. The decision procedure presented in this chapter can be used to automatically verify the validity of the quantifier-free verification conditions generated using the parametrized invariance rules and the parametrized verification diagrams techniques presented in Chapter 3 and Chapter 4. Examples of the use of TSL_K decision procedure can be consulted in Chapter 10.

Additionally, TSL_K is a fundamental building block in the construction of even more powerful skiplist theories. In Chapter 8 we present TSL, a theory for skiplists with an unbounded number of levels. The decidability of TSL relies, among other things, on the fact that the satisfiability of TSL_K quantifier-free formulas is decidable. The decision procedure for TSL quantifier-free formulas we present in the next chapter uses a decision procedure for TSL_K quantifier-free formulas.

8

TSL: A Decidable Theory for Skiplists with Unbounded Levels

> *No matter how beautiful the theory, one irritating fact can dismiss the entire formulism, so it has to be proven.*
>
> Michio Kaku

In Chapter 7 we presented $\mathsf{TSL_K}$, a family of decidable theories capable of describing rich structural and functional properties of data structures with the shape of a skiplist. The main disadvantage of the theory of families $\mathsf{TSL_K}$ is that they are restricted to skiplists with a fixed number of K levels, which makes these theories unsuitable for the efficient verification of real world skiplist implementations. The problem of skiplists with a fixed number of levels is that its average performance decreases as more elements are inserted, as the skiplist saturates and it loses its ability to *skip* many nodes with a single jump. Because of this, most implementations of skiplists either can grow dynamically to any height or limit the maximum height of any node to a large value like 32, making impractical the use of $\mathsf{TSL_K}$. In both cases, implementations use a program variable to store the current highest level in use.

In this chapter we present TSL, *The Theory of Skiplists with Unbounded Levels*. TSL is a theory suitable for the formal verification of skiplists with an arbitrary number of levels, whose number can additionally be dynamically modified. In this chapter we formally introduce TSL, we show can it can be used in the verification of real world skiplist implementations and we

show it decidable. The TSL decision procedure we present in this chapter works by reducing the satisfiability problem of a TSL quantifier-free formula to queries to the TSL_K and Presburger arithmetic decision procedures. Despite using TSL_K as a backbone decision procedure, the version of TSL that we present here is capable of reasoning only about sequential implementations of skiplists. We leave the problem of a concurrent version of TSL as future work.

The rest of the chapter is structured as follows. Section 8.1 presents an implementation of a skiplist with an unbounded number of levels. Section 8.2 formally presents TSL. Section 8.3 describes the procedure for normalizing TSL formulas. Section 8.4 demonstrates that TSL is decidable by reducing the satisfiability problem of TSL quantifier-free formulas to queries to decision procedures for TSL_K and Presburger arithmetic. Finally, Section 8.5 concludes.

8.1 A Skiplist with an Unbounded Number of Levels

In Section 7.1 of Chapter 7 we introduced the skiplist data structure and we presented some of the basic conditions and properties satisfied by a skiplist, such as list inclusion between the nodes connected at a certain level and the nodes connected at the level immediately below. A disadvantage of the skiplist presented in that chapter is that it contains a constant number K of levels, where K could be arbitrary large, but fixed. This constraint on the number of levels, makes a skiplist inefficient as more elements are inserted into the skiplist. The efficiency of a skiplist comes from its ability to *skip* the traversal of nodes only connected at levels lower than the current level.

Because of this, real world implementations usually either fix a large number of levels (generally 32) or they store the current maximum level in a variable that can grow. By doing so, the skiplist can have arbitrary as many levels as it requires, preserving the average performance of operations over the skiplist.

In order to illustrate the use of the decision procedure for skiplists of unbounded height we present in this chapter, we will first present an implementation of a skiplist which keeps the maximum number of levels in a variable, letting it grow as required.

The nodes of a skiplist with an unbounded number of levels are defined as instances of the *UnboundedSkiplistNode* class, declared as follows:

$$\textbf{class}\quad \mathit{UnboundedSkiplistNode}\quad \{\quad \begin{array}{ll} \mathit{Elem} & \mathit{data}; \\ \mathit{Array}\langle \mathit{Addr}\rangle & \mathit{next}; \\ \mathit{Int} & @\mathit{level}; \quad \} \end{array}$$

An object of class *UnboundedSkiplistNode* contains the following fields:

- $data$, which contains the actual value stored in the node;
- $next$, an (unbounded) array of pointers which keeps the addresses of the next node in the skiplist at each level;
- $level$, a ghost field which stores the highest level stored in $next$ which is not $null$. In other words, the size of array $next$.

As usual, in order to simplify the presentation, we assume that the elements stored in $data$ are used to keep the order. It is easy to modify the *UnboundedSkiplistNode* class to store pairs

$(data, key)$ of values and keys, where $data$ keeps the elements of the skiplist and key keeps a key to maintain the nodes of the skiplist ordered.

The implementation of the skiplist with an unbounded number of levels we present here maintain two global addresses $head$ and $tail$, one global integer variable $maxLevel$ and two ghost global variables reg and $elems$:

global
$Addr\ head$
$Addr\ tail$
$Int\ maxLevel$
$Set\langle Addr\rangle\ reg$
$Set\langle Elem\rangle\ elems$

We describe now the intended meaning of these global variables:

(a) A variable $head$, of type address, which points to the first node of the skiplist.

(b) A variable $tail$, of type address, which points to the last node of the skiplist.

(c) A variable $maxLevel$, of type integer, which keeps the maximum current level of the whole skiplist.

(d) A ghost variable reg, of type set of addresses, which is used to keep track of the portion of the heap that forms the skiplist.

(e) A ghost variable $elems$, of type set of elements, which represents the collection of elements stored in the skiplist.

As before, we assume that the nodes pointed by $head$ and $tail$ store $-\infty$ and $+\infty$, that is, the lowest and highest possible values respectively. Additionally, the nodes pointed by $head$ and $tail$ are sentinel nodes which we assume are initialized to two different nodes which conform an *empty* skiplist containing only the $-\infty$ and $+\infty$ values. The nodes pointed by $head$ and $tail$ cannot be removed from the skiplist or have their $data$ field modified. Similarly, reg is initialized containing only the addresses of $head$ and $tail$, and $elems$ is initialized containing only $-\infty$ and $+\infty$.

Example 8.1 (Unbounded Skiplist Initialization)
According to the description we gave above, Fig. 8.1 shows the representation of an unbounded

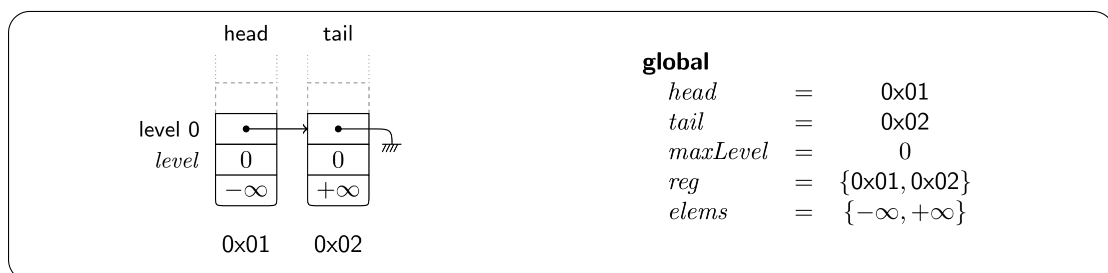


Figure 8.1: An initialized unbounded skiplist.

initialized skiplist. ⌟

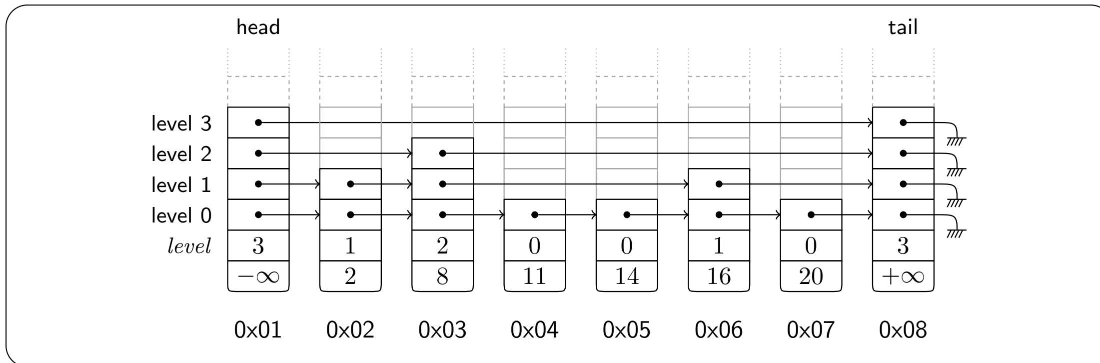


Figure 8.2: An example of an unbounded skiplist.

Example 8.2
Fig. 8.2 presents an example of an unbounded skiplist after some elements have been inserted. Note how each node contains an unbounded array of addresses which are used to store the pointers to the following nodes at all skiplist levels. As can be seen, the $level$ field is used to store the maximum level of the node which does not points to $null$. In the figure, we gray out the array levels which point to $null$. For example, the node stored at address 0x03 contains value 8 and an array $next$ with all levels from level 0 up to level 2 containing pointers different than $null$. Note that, to be a valid skiplist, the data structure cannot contain nodes with some null level in between of two non null levels.

As all used levels in this particular example are between 0 and 3, we have that in this case $maxLevel$ is 3.

The implementation of the unbounded skiplist provides three main operations to manipulate the data structure. These operations are SEARCH, INSERT and REMOVE:

SEARCH: described in Fig. 8.3, receives an element e and traverses the skiplist in order to determine whether e is present or not in the skiplist. Procedure SEARCH uses 3 local variables:

- An integer variable i, which denotes the level of the skiplist being currently traversed.
- Two variables, $prev$ and $curr$, of sort address which are used to keep track of the position at a certain level of the skiplist while it is traversed.

The procedure presented here is slightly simpler than the SEARCH procedure presented in Section 7.1, mainly because now we do not need to manipulate locks protecting sections of the skiplist. The procedure begins by assigning $prev$ to the $head$ of the skiplist (line 1). Then, it assigns $curr$ to the node following $head$ at the currently highest possible level of the skiplist (line 2) and assigns to local variable i the highest level of the skiplist (line 3).

The loop (lines 4 to 11) performs the search of the element e in the skiplist. The loop executes while there are levels in the skiplist to explore ($0 \leq i$) and the element e has not yet been found ($curr \to data \neq e$). Inside the loop, $curr$ is assigned to the element following $prev$ at level i (line 5). This corresponds to starting the search on a new level.

```
procedure SEARCH(Elem e)
   Int i
   Addr prev, curr
begin
 1: prev := head
 2: curr := prev→next[maxLevel]
 3: i := maxLevel
 4: while 0 ≤ i ∧ curr→data ≠ e do
 5:     curr := prev→next[i]
 6:     while curr→data < e do
 7:         prev := curr
 8:         curr := prev→next[i]
 9:     end while
10:     i := i − 1
11: end while
12: return curr→data = e
end procedure
```

Figure 8.3: SEARCH procedure for an unbounded skiplists.

The internal loop (line 6 to 9) performs the search using pointers $prev$ and $curr$ to explore the skiplist at level i. This inner loop is executed while the element e has not been found in the current level. As the nodes in the skiplist are ordered, this corresponds to checking whether ($curr{\rightarrow}data < e$). If this condition holds, then pointers $prev$ and $curr$ advance one node of the skiplist by assigning $prev$ to the current node pointed by $curr$ (line 7) and $curr$ to the node following $prev$ at level i (line 8). At the end of this inner loop we have that $curr{\rightarrow}data \geq e$, meaning that e was not found in level i and thus we can safely decrement i (line 10) to perform the search at a lower level in the next iteration of the outer loop.

At the end of SEARCH, if $curr$ points to the node containing e then $true$ is returned, otherwise SEARCH returns $false$ to denote that e was not in the skiplist (line 12).

INSERT: presented in Fig. 8.4, this procedure receives an element e as argument and proceeds to insert e into the skiplist. If e was already in the skiplist, then INSERT does not modify the skiplist.

Procedure INSERT uses 6 local variables:

- Two integer variables, i and lvl, which are used to maintain the current skiplist level being traversed and the level of the new node to be inserted respectively.
- Two variables $prev$ and $curr$ of type address, which are used to keep track of the nodes the procedure is traversing.
- An array of addresses upd, which stores the pointers that need to be modified in order to insert the new node in the skiplist.
- A Boolean variable $valueWasIn$, which indicates if e was already in the skiplist, and consequently indicates whether INSERT does not require to modify the skiplist.

INSERT begins by initializing $valueWasIn$ to $false$ (line 1). That is equivalent to initially assuming that e is not in the skiplist. Then INSERT randomly decides the level of the new

node to be inserted by performing a call to the *randomLevel* and storing the random level in local variable *lvl* (line 2).

At this point two things can happen: the level assigned to *lvl* is lower or equal to the current *maxLevel*, or it is greater than the current *maxLevel*. If *lvl* is higher that *maxLevel*, then the height of the sentinel nodes *head* and *tail* needs to be updated in lines 3 to 11. Additionally,

```
    procedure INSERT(Elem e)
       Int i, lvl
       Addr prev, curr
       Array⟨Addr⟩ upd
       Bool valueWasIn
    begin
 1: valueWasIn := false
 2: lvl := randomLevel()
 3: if lvl > maxLevel then
 4:     i := maxLevel + 1
 5:     while i ≤ lvl do
 6:         head→next[i] := tail
 7:         tail→next[i] := null
 8:         maxLevel := i
 9:         i := i + 1
10:     end while
11: end if
12: upd := newArray⟨Addr⟩(maxLevel + 1)
13: prev := head
14: curr := prev→next[maxLevel]
15: i := maxLevel
16: while 0 ≤ i ∧ ¬valueWasIn do
17:     curr := prev→next[i]
18:     while curr→data < e do
19:         prev := curr
20:         curr := prev→next[i]
21:     end while
22:     upd[i] := prev
23:     i := i − 1
24:     valueWasIn := (curr→data = e)
25: end while
26: if ¬valueWasIn then
27:     newnode := CreateNode(lvl, e)
28:     i := 0
29:     while i ≤ lvl do
30:         newnode→next[i] := upd[i]→next[i]
31:         upd[i]→next[i] := newnode
            if i = 0 then
                reg := reg ∪ {newnode}
                elems := elems ∪ {e}
32:         i := i + 1
33:     end while
34: end if
35: return ¬valueWasIn
    end procedure
```

Figure 8.4: Procedure INSERT for unbounded skiplists.

the value of $maxLevel$ is also updated (line 8). At line 11, $lvl \leq maxLevel$ always holds.

At this point, INSERT allocates upd with $maxLevel$ slots (line 12). Pointer $prev$ is then initialized to point $head$ (line 13), $curr$ is initialized to point to the node following $head$ at the current maximum skiplist level (line 14) and i is initialized with the current value of $maxLevel$.

INSERT now is ready to start the search of e in the skiplist. The loop between lines 16 and 25 is similar to the inner loop of SEARCH. In this loop, pointers $prev$ and $curr$ advance in the current level as far as possible until $curr$ points to a node whose element in greater of equal than e. A novelty at INSERT is that when $curr$ finds an element greater or equal than e, the node pointed by $prev$ is stored in the upd array (line 22). This ensures that at the end of the loop, for all possible level j between 0 and $maxLevel$, the following holds:

$$upd[j]{\to}data < e \quad \wedge \quad upd[j]{\to}next[j]{\to}data \geq e$$

This way, upd is used to store all pointers to the nodes that need to be modified once the node containing element e is inserted. Similarly, inside the loop the Boolean variable $valueWasIn$ is updated to reflect whether a node with element e is present in the skiplist.

After exiting the loop, if e was not found in the skiplist, then a new node with e is inserted into the skiplist (lines 26 to 34). First INSERT creates a new node $newnode$ which contains element e and lvl levels in its $next$ array (line 27). Then, $newnode$ is connected to the rest of the skiplist by modifying the $next$ array of $newnode$ and the $next$ array at the nodes pointed by upd (lines 29 to 33). Note that the connection of $newnode$ is done increasingly, starting from level 0 up to level lvl. By connecting $newnode$ this way, we ensure that the skiplist property is preserved. As soon as $newnode$ is connected to the skiplist (that is, it is connected at level 0), the node can be considered part of the skiplist, so the ghost variables reg and $elems$ can be updated accordingly (line 31). Finally, the procedure returns $true$ if a new node with element e was created and inserted into the skiplist and $false$ if element e was already in the skiplist, and thus it was not needed to be inserted.

REMOVE: shown in Fig. 8.5, this procedure receives an element e as argument and proceeds to remove it from the skiplist. If e is not in the skiplist, then the data structure is left unchanged.

Procedure REMOVE uses 6 local variables:

- Two integer variables, i and $removeFrom$, which are used to denote the current skiplist level being traversed and the highest level at which the node containing element e is connected to the rest of the skiplist respectively.
- Two variables $prev$ and $curr$ of type address for the traversal.
- An array of addresses upd, which keeps the pointers that needs to be modified in order to remove the node containing element e from the skiplist.
- A Boolean variable $valueWasIn$ which is used to store whether a node containing e was found in the skiplist.

REMOVE begins by assigning to $removeFrom$ the highest possible level, that is, $maxLevel$ (line 1). Then, REMOVE proceeds to initialize pointers $prev$ and $curr$ so that they point to

the head of the skiplist and the node following the head at the highest possible level (line 2 and 3). As the search for the position where e should be in the skiplist is done in a top-down fashion, the procedure initializes i with the highest possible level of the skiplist (line 4).

At this point, the procedure looks for e in the skiplist. This search is done in the loop that goes from line 5 to line 16. The loop itself is very similar to the loops in SEARCH and INSERT. This loop updates upd array to keep a record of the pointers it needs to modify once the node containing element e is found. Note that if while traversing a level the node containing element e was not found, then the value of $removeFrom$ is decreased (line 12). Once the loop has finished, we have that for every level j between 0 and $maxLevel$, the following holds:

$$upd[j]{\rightarrow}data < e \quad \wedge \quad upd[j]{\rightarrow}next[j]{\rightarrow}data \geq e$$

```
    procedure REMOVE(Elem e)
      Int i, removeFrom
      Addr prev, curr
      Array⟨Addr⟩(maxLevel + 1) upd
      Bool valueWasIn
    begin
 1: removeFrom := maxLevel
 2: prev := head
 3: curr := pred→next[maxLevel]
 4: i := maxLevel
 5: while i ≥ 0 do
 6:     curr := prev→next[i]
 7:     while curr→data < e do
 8:         prev := curr
 9:         curr := prev→next[i]
10:     end while
11:     if curr→data ≠ e then
12:         removeFrom := i − 1
13:     end if
14:     upd[i] := prev
15:     i := i − 1
16: end while
17: valueWasIn := (curr→data = e)
18: if valueWasIn then
19:     i := removeFrom
20:     while i ≥ 0 do
21:         upd[i]→next[i] := curr→next[i]
            if i = 0 then
                reg := reg − {curr}
                elems := elems − {e}
22:         i := i − 1
23:     end while
24:     free (curr)
25: end if
26: return valueWasIn
    end procedure
```

Figure 8.5: REMOVE procedure for an unbounded skiplists.

Moreover, $removeFrom$ equals to the highest level at which the node with element e is connected with the rest of the skiplist if such node was found or (-1) if no node storing e was found in the skiplist.

At this point, the Boolean variable $valueWasIn$ is set to reflect whether e was present in the skiplist (line 17). If e was not found, then REMOVE returns. On the contrary, if e was found, then the procedure proceeds to remove it from the skiplist (line 18 to 25). In order to preserve the skiplist shape of the data structure, the node containing e is unlinked starting from $removeFrom$ level and proceeding down to level 0. In order to remove the node storing e, REMOVE modifies the pointers stored at upd (line 21). Once the node with element e is disconnected at level 0, REMOVE modifies the ghost sets reg and $elems$ to reflect that e is not in the skiplist (line 21). Finally, after the node has been completely disconnected form the skiplist, the memory from this node can be reclaimed (line 24). REMOVE returns $true$ if e was removed from the skiplist or $false$ if e was not present in the skiplist.

Once again, in order to verify the data type against all possible clients, we can create the most general client of the unbounded skiplist, called MGCSKIPLIST. The most general client is presented in Fig. 8.6. The most general client non-deterministically performs calls to all skiplist operations, to exercise all possible sequences of calls. We later use procedure MGCSKIPLIST to verify properties like skiplist preservation.

8.2 TSL: A Theory for Skiplists with Unbounded Levels

In this section we formally present TSL, a multi-sorted first-order theory which is amenable for describing rich properties of skiplists with an unbounded number of levels. Later, we will show that TSL is decidable by proving that the decision problem of the satisfiability of TSL formulas can be reduced to a satisfiability problem of $\mathsf{TSL_K}$ formulas, the decidable family of theories presented in Chapter 7.

We define the *Theory of Skiplists with Unbounded Levels* TSL as a combination of theories $\mathsf{TSL} = (\Sigma_{\mathsf{TSL}}, \mathbf{TSL})$, where

$$\Sigma_{\mathsf{TSL}} = \Sigma_{\mathsf{level}} \cup \Sigma_{\mathsf{array}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{setaddr}} \cup \Sigma_{\mathsf{setelem}} \cup \Sigma_{\mathsf{reach}} \cup \Sigma_{\mathsf{bridge}}$$

```
   procedure MGCSkiplist()
     Elem e
   begin
1: while true do
2:    e := havocSkiplistElem()
3:    nondet choice
4:           call Search(e)
5:       or  call Insert(e)
6:       or  call Remove(e)
7:    end choice
8: end while
   end procedure
```

Figure 8.6: Most general client procedure MGCSKIPLIST for unbounded skiplists.

Informally, Σ_{level} models levels of the skiplist. The main difference between Σ_{level} form TSL and $\Sigma_{\mathsf{level}_\mathsf{K}}$ from TSL_K is that $\Sigma_{\mathsf{level}_\mathsf{K}}$ models levels as a finite set $\{0, \ldots, \mathsf{K}-1\}$, while Σ_{level} contains all natural numbers. Σ_{array} models arrays indexed by levels that store addresses. Σ_{cell} models *cells*, structures containing an element (data) which is also used to keep cells ordered, an array from levels to addresses (pointers to the successor node at each level) and the current maximum level of the cell. A cell represents a node in a skiplist. The main difference between the signature Σ_{cell} introduced here and the signature $\Sigma_{\mathsf{cell}_\mathsf{K}}$ presented in Chapter 7 is that Σ_{cell} contains an array of addresses, while $\Sigma_{\mathsf{cell}_\mathsf{K}}$ keeps K pointers to the next cells. Σ_{mem} models the memory. Σ_{reach}, as $\Sigma_{\mathsf{reach}_\mathsf{K}}$ from TSL_K, models finite sequences of non-repeating addresses, which is used to represent acyclic paths in the memory. The main difference between $\Sigma_{\mathsf{reach}_\mathsf{K}}$ and Σ_{reach} is that the latter operates on levels in Σ_{level} instead of $\Sigma_{\mathsf{level}_\mathsf{K}}$. Finally, as in TSL_K, Σ_{bridge} is a *bridge theory* containing auxiliary functions similar to the ones contained in Σ_{bridge} presented in Chapter 7. The main difference in this case is the introduction of a predicate $skiplist$ which captures whenever there is a shape of a well formed skiplist in the heap.

We describe now the sorts, the signature and restrictions on the interpretation for each of the theories in TSL.

8.2.1 Sorts

The sorts shared among these theories are level, elem, addr, array, cell, mem, path, setaddr and setelem. The intended meaning of these sorts is:

- level: levels of the skiplist.
- elem: elements stored in the skiplist.
- addr: memory addresses.
- array: arrays from levels to addresses.
- cell: skiplist nodes stored in the heap.
- mem: heaps, represented as maps from addresses to cells.
- path: paths, as finite sequences of non-repeating addresses.
- setaddr: sets of addresses.
- setelem: sets of elements.

The class of Σ_{TSL}-structures restrict the domain of the sorts to interpretations $\mathcal{A}$ that satisfy the following:

(a) $\mathcal{A}_{\mathsf{level}}$ is the set of natural numbers with their usual order.

(b) $\mathcal{A}_{\mathsf{elem}}$ and $\mathcal{A}_{\mathsf{addr}}$ are discrete sets.

(c) $\mathcal{A}_{\mathsf{array}} = \mathcal{A}_{\mathsf{addr}}^{\mathcal{A}_{\mathsf{level}}}$.

(d) $\mathcal{A}_{\mathsf{cell}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{array}} \times \mathcal{A}_{\mathsf{level}}$.

(e) $\mathcal{A}_{\mathsf{mem}} = \mathcal{A}_{\mathsf{cell}}^{\mathcal{A}_{\mathsf{addr}}}$.

(f) $\mathcal{A}_{\mathsf{path}}$ is the set of all finite sequences of (pairwise) distinct elements of $\mathcal{A}_{\mathsf{addr}}$.

(g) $\mathcal{A}_{\mathsf{setaddr}}$ is the power-set of $\mathcal{A}_{\mathsf{addr}}$.

(h) $\mathcal{A}_{\mathsf{setelem}}$ is the power-set of $\mathcal{A}_{\mathsf{elem}}$.

8.2.2 Signature

The functions and predicates for theory Σ_{elem}, $\Sigma_{\mathsf{setaddr}}$ and $\Sigma_{\mathsf{setelem}}$ are the same as the ones presented in Section 7.2.2 of Chapter7 for $\mathsf{TSL_K}$. Something similar happens with Σ_{mem}, for which the only difference between the signature $\Sigma_{\mathsf{mem_K}}$ presented in Section 7.2.2 of Chapter 7 and the signature Σ_{mem} presented here is that Σ_{mem} maps addresses to unbounded skiplist cells as captured by Σ_{cell} instead of the bounded cells of K levels described by $\Sigma_{\mathsf{cell_K}}$. We now focus on the theories that TSL does not share with $\mathsf{TSL_K}$. For each of these new theories we list now their sorts and each of the functions and predicates with their signatures:

a) Σ_{level}: The only sort used is $\mathsf{level_K}$. The function symbols are:

$$\begin{array}{rcl} 0 & : & \mathsf{level} \\ s & : & \mathsf{level} \to \mathsf{level} \end{array}$$

The constant function 0 models the natural number 0. Function s models the successor operation. Given a variable l of sort level, we generally use $l+1$ for $s(l)$. The only predicate symbol in this theory, apart from equality, is:

$$< \ : \ \mathsf{level} \to \mathsf{level}$$

The predicate $<$ describes the order relation between natural numbers.

b) Σ_{array}: The sorts used are array, level and addr. The function symbols are:

$$\begin{array}{rcl} _[_] & : & \mathsf{array} \times \mathsf{level} \to \mathsf{addr} \\ _\{_ \leftarrow _\} & : & \mathsf{array} \times \mathsf{level} \times \mathsf{addr} \to \mathsf{array} \end{array}$$

The function $_[_]$ models an array dereference that returns an address given an array and a level. The function $_\{_ \leftarrow _\}$ is used to create a modified array given an array, a level and the address to be stored in the new array at the given level. There are no predicate symbols in Σ_{array}, except array equality.

c) Σ_{cell}: The sorts used are cell, level, elem, array and addr. The function symbols are:

$$\begin{array}{rcl} error & : & \mathsf{cell} \\ mkcell & : & \mathsf{elem} \times \mathsf{array} \times \mathsf{level} \to \mathsf{cell} \\ _.data & : & \mathsf{cell} \to \mathsf{elem} \\ _.arr & : & \mathsf{cell} \to \mathsf{array} \\ _.max & : & \mathsf{cell} \to \mathsf{level} \end{array}$$

The cell $error$ is used to model the return of an incorrect memory dereference. Function $mkcell$ is the constructor of cells. The corresponding selectors are the functions $data$, arr

and max. Selector $data$ accesses the data stored in the cell, arr returns the array with the pointers to the successor nodes and max provides the maximum current level of successor pointers in the node. There are no predicate symbols in Σ_{cell} except equality between cells.

d) Σ_{reach}: The sorts used are mem, addr, level and path. The function symbols are:

$$
\begin{aligned}
\epsilon &: \mathsf{path} \\
[_] &: \mathsf{addr} \to \mathsf{path}
\end{aligned}
$$

As in $\mathsf{TSL_K}$, the constant ϵ models the empty path, and the function $[_]$ allows to build a singleton path from the provided address. The predicate symbols in Σ_{reach}, apart from equality, are:

$$
\begin{aligned}
append &: \mathsf{path} \times \mathsf{path} \times \mathsf{path} \\
reach &: \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level} \times \mathsf{path}
\end{aligned}
$$

As in $\mathsf{TSL_K}$, the predicate $append$ relates two paths with its concatenation. The predicate $reach$ relates two addresses with the path that connects them in a given heap at a certain skiplist level.

d) Σ_{bridge}: The sorts used are mem, addr, level, setaddr and path. The function symbols are:

$$
\begin{aligned}
path2set &: \mathsf{path} \to \mathsf{setaddr} \\
getp &: \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level} \to \mathsf{path} \\
addr2set &: \mathsf{mem} \times \mathsf{addr} \times \mathsf{level} \to \mathsf{setaddr}
\end{aligned}
$$

As in $\mathsf{TSL_K}$, the function $path2set$ returns the set of addresses presented in a given path. The function $addr2set$ returns the set of addresses reachable from a given address by following the successor pointers at a certain skiplist level. Similarly, the function $getp$ returns the path that connects two addresses in a given heap at a certain level, if there is one (or the empty path otherwise).

The predicate symbols in Σ_{bridge} are:

$$
\begin{aligned}
ordPath &: \mathsf{mem} \times \mathsf{path} \\
skiplist &: \mathsf{mem} \times \mathsf{setaddr} \times \mathsf{level} \times \mathsf{addr} \times \mathsf{addr}
\end{aligned}
$$

Predicate $ordPath$ holds whenever the data stored at the cells obtained by dereferencing the addresses in the path (using the given memory) are ordered. Predicate $skiplist$ captures whether the cells stored in a region of the heap have the shape of a skiplist. This predicate relates the cells accessible between the provided addresses considering pointers up to the given level. The predicate $skiplist$ also checks that the addresses of all cells conforming the skiplist are exactly the ones stored in the given set of addresses. Contrary to $\mathsf{TSL_K}$, the predicate $skiplist$ is a native predicate in the theory, not a predicate defined in terms of other predicates.

8.2.3 Interpretations

We restrict the class of models to **TSL**, a class of Σ_{TSL}-structures that satisfy the following conditions:

a) $\Sigma_{\mathsf{mem}}, \Sigma_{\mathsf{setaddr}}, \Sigma_{\mathsf{setelem}}$: These theories are interpreted as in $\mathsf{TSL_K}$.

b) Σ_{level}: In TSL, an interpretation $\mathcal{A}$ of Σ_{level} interprets the functions and predicates of Σ_{level} as the natural numbers with addition and order. That is, for every level $l \in \mathcal{A}_{\mathsf{level}}$:

- $0^{\mathcal{A}} = 0$
- $s^{\mathcal{A}}(l) = l + 1$

Also, $l < l'$ is interpreted as the usual order within the natural numbers.

c) Σ_{array}: Every interpretation $\mathcal{A}$ of Σ_{array} must satisfy, that for every array $A, B \in \mathcal{A}_{\mathsf{array}}$, level $l \in \mathcal{A}_{\mathsf{level}}$, and address $a \in \mathcal{A}_{\mathsf{addr}}$:

- $A[l]^{\mathcal{A}} = A(l)$
- $A\{l \leftarrow a\}^{\mathcal{A}} = A_{new}$, where $A_{new}(l) = a$ and $A_{new}(i) = A(i)$ for $i \neq l$

d) Σ_{cell}: Every interpretation $\mathcal{A}$ of Σ_{cell} must satisfy, for every element $e \in \mathcal{A}_{\mathsf{elem}}$, array $A \in \mathcal{A}_{\mathsf{array}}$ and level $l \in \mathcal{A}_{\mathsf{level}}$:

- $mkcell^{\mathcal{A}}(e, A, l) = \langle e, A, l \rangle$
- $error^{\mathcal{A}}.arr^{\mathcal{A}}(l) = null^{\mathcal{A}}$
- $\langle e, A, l \rangle.data^{\mathcal{A}} = e$
- $\langle e, A, l \rangle.arr^{\mathcal{A}} = A$
- $\langle e, A, l \rangle.max^{\mathcal{A}} = l$

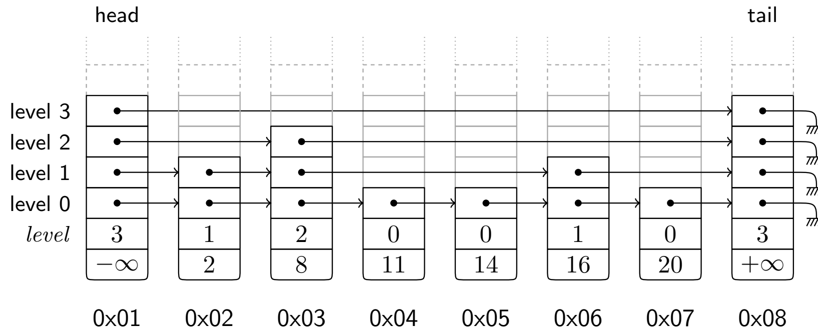
Essentially, **TSL** models restrict cells to be records consisting of an element, an array of addresses and a level.

e) Σ_{reach}: The symbols ϵ, $[_]$, $append$ and $reach$ are interpreted as in $\mathsf{TSL_K}$.

f) Σ_{bridge}: The interpretation of $addr2set$, $path2set$, $getp$ and $ordPath$ are restricted as in $\mathsf{TSL_K}$. For the predicate $skiplist$ we force every interpretation $\mathcal{A}$ of Σ_{bridge} to satisfy for every memory $m \in \mathcal{A}_{\mathsf{mem}}$, path $p \in \mathcal{A}_{\mathsf{path}}$, set of address $r \in \mathcal{A}_{\mathsf{setaddr}}$, level $l \in \mathcal{A}_{\mathsf{level_K}}$ and addresses $a, b \in \mathcal{A}_{\mathsf{addr}}$ the following:

$$\begin{array}{l} skiplist^{\mathcal{A}}(m, r, l, a, b) \quad \text{iff} \\ \left[\begin{array}{lll} r = addr2set^{\mathcal{A}}(m, a, 0) & \wedge & ordList^{\mathcal{A}}(m, getp^{\mathcal{A}}(m, a, null, 0)) \quad \wedge \\ b \in addr2set^{\mathcal{A}}(m, a, l) & \wedge & \text{for all } x \in r \,.\, m(x).max^{\mathcal{A}} \leq l \quad \wedge \\ m(b).arr^{\mathcal{A}}(l) = null^{\mathcal{A}} & \wedge & \\ \multicolumn{3}{l}{\left[\begin{array}{l} (0 = l) \ \vee \\ \left[\begin{array}{l} \text{for some } l_p \,.\, s^{\mathcal{A}}(l_p) = l \ \wedge \\ \text{for all } i \in 0, \ldots, l_p \,. \\ \left[\begin{array}{c} b \in addr2set^{\mathcal{A}}(m, a, i) \quad \wedge \\ m(b).arr^{\mathcal{A}}(i) = null^{\mathcal{A}} \ \wedge \\ path2set^{\mathcal{A}}(getp^{\mathcal{A}}(m, a, null, s^{\mathcal{A}}(i))) \ \subseteq \\ path2set^{\mathcal{A}}(getp^{\mathcal{A}}(m, a, null, i)) \end{array}\right] \end{array}\right] \end{array}\right]} \end{array}\right] \end{array} \tag{8.1}$$

Example 8.3
Consider again the following skiplist snapshot, introduced in Fig. 8.2:



The following interpretation $\mathcal{A}$ is in the class **TSL**:

$$\begin{aligned} \mathcal{A}_{\mathsf{addr}} &= \{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08\} \\ \mathcal{A}_{\mathsf{elem}} &= \{-\infty, 2, 8, 11, 14, 16, 20, +\infty\} \\ \mathcal{A}_{\mathsf{mem}} &= \{m : \mathcal{A}_{\mathsf{addr}} \to \mathcal{A}_{\mathsf{cell}}\} \end{aligned}$$

where

$$\begin{aligned} null^{\mathcal{A}} &= 0x00 \\ error^{\mathcal{A}} &= \langle\ -\infty \,,\, A_{null} \,,\, 0\ \rangle \end{aligned}$$

and A_{null} is the array from levels to addresses that for all natural i, $A_{null}[i] = null$. Moreover,

$$\begin{aligned} m(0x00) &= \langle\ -\infty \,,\, A_{null} \,,\, 0\ \rangle \\ m(0x01) &= \langle\ -\infty \,,\, A_1 \,,\, 3\ \rangle \\ m(0x02) &= \langle\ 2 \,,\, A_2 \,,\, 1\ \rangle \\ m(0x03) &= \langle\ 8 \,,\, A_3 \,,\, 2\ \rangle \\ m(0x04) &= \langle\ 11 \,,\, A_4 \,,\, 0\ \rangle \end{aligned}$$

$$\begin{array}{rcl} m(\mathsf{0x05}) & = & \langle\ 14\ , A_5\ , 0\ \rangle \\ m(\mathsf{0x06}) & = & \langle\ 16\ , A_6\ , 1\ \rangle \\ m(\mathsf{0x07}) & = & \langle\ 20\ , A_7\ , 0\ \rangle \\ m(\mathsf{0x08}) & = & \langle\ +\infty\ , A_8\ , 8\ \rangle \end{array}$$

where for each of the arrays used above, we have:

$$\begin{array}{llllll} A_1[0] = \mathsf{0x02} & A_1[1] = \mathsf{0x02} & A_1[2] = \mathsf{0x03} & A_1[3] = \mathsf{0x08} & A_1[i] = \mathsf{0x00} & \text{for all } i \geq 4 \\ A_2[0] = \mathsf{0x03} & A_2[1] = \mathsf{0x03} & A_2[i] = \mathsf{0x00} & & & \text{for all } i \geq 2 \\ A_3[0] = \mathsf{0x04} & A_3[1] = \mathsf{0x06} & A_3[2] = \mathsf{0x08} & A_3[i] = \mathsf{0x00} & & \text{for all } i \geq 3 \\ A_4[0] = \mathsf{0x05} & A_4[i] = \mathsf{0x00} & & & & \text{for all } i \geq 1 \\ A_5[0] = \mathsf{0x06} & A_5[i] = \mathsf{0x00} & & & & \text{for all } i \geq 1 \\ A_6[0] = \mathsf{0x07} & A_6[1] = \mathsf{0x08} & A_6[i] = \mathsf{0x00} & & & \text{for all } i \geq 2 \\ A_7[0] = \mathsf{0x08} & A_7[i] = \mathsf{0x00} & & & & \text{for all } i \geq 1 \\ A_8[i] = \mathsf{0x00} & & & & & \text{for all } i \geq 0 \end{array}$$

According to this interpretation we have, for instance, that:

$$(m, \{\mathsf{0x01}, \mathsf{0x02}, \mathsf{0x03}, \mathsf{0x04}, \mathsf{0x05}, \mathsf{0x06}, \mathsf{0x07}, \mathsf{0x08}\}, 3, \mathsf{0x01}, \mathsf{0x00}) \in \mathit{skiplist}^{\mathcal{A}} \tag{8.2}$$

$$(m, \{\mathsf{0x01}, \mathsf{0x02}, \mathsf{0x03}, \mathsf{0x04}, \mathsf{0x05}, \mathsf{0x06}, \mathsf{0x07}, \mathsf{0x08}\}, 4, \mathsf{0x01}, \mathsf{0x00}) \in \mathit{skiplist}^{\mathcal{A}} \tag{8.3}$$

$$(m, \{\mathsf{0x01}, \mathsf{0x02}, \mathsf{0x03}, \mathsf{0x04}, \mathsf{0x06}, \mathsf{0x07}, \mathsf{0x08}\}, 3, \mathsf{0x01}, \mathsf{0x00}) \notin \mathit{skiplist}^{\mathcal{A}} \tag{8.4}$$

$$(m, \{\mathsf{0x01}, \mathsf{0x02}, \mathsf{0x03}\}, 2, \mathsf{0x01}, \mathsf{0x04}) \notin \mathit{skiplist}^{\mathcal{A}} \tag{8.5}$$

$$(m, \{\mathsf{0x06}, \mathsf{0x07}, \mathsf{0x08}\}, 1, \mathsf{0x06}, \mathsf{0x00}) \in \mathit{skiplist}^{\mathcal{A}} \tag{8.6}$$

In the case of (8.2), the predicate holds because it represents the whole skiplist depicted in Fig. 8.2. This example shows that after removing arbitrary many levels from the top of a skiplist, the resulting structure keeps the shape of a skiplist. On the contrary, considering higher levels may not lead to a skiplist structure. For example, (8.3) does not hold because at level 4, the address 0x08 is not reachable from 0x01. In the case of (8.4), the predicate does not hold because address 0x05 is missing in the set of addresses. Predicate (8.5) fails because address 0x03 is not null terminated all all levels from level 0 to level 2. However, note how predicate (8.6) does hold since if we consider the portion of the memory that goes from address 0x06 to address 0x08 from level 0 up to level 1 the resulting structure has the shape of a skiplist. ⌟

8.3 Normalization of TSL

In this section we prove the decidability of the satisfiability problem of quantifier-free TSL formulas. The decision procedure proceeds by reducing the satisfiability problem of quantifier-free TSL formulas to the satisfiability of quantifier-free $\mathsf{TSL_K}$ formulas and quantifier-free Presburger

arithmetic formulas. We start from a TSL formula φ in disjunctive normal form: $\varphi_1 \vee \cdots \vee \varphi_n$ so the procedure only needs to check the satisfiability of a conjunction of TSL literals in φ_i. The rest of this section describes the decision procedure and proves its correctness.

We begin by identifying the set of normalized literals of TSL.

Definition 8.1 (TSL-normalized literals).
A TSL-literal is normalized if it is a flat literal of the form:

$e_1 \neq e_2$	$a_1 \neq a_2$	$l_1 \neq l_2$
$e_1 \preceq e_2$	$a = null$	$c = error$
$c = mkcell(e, A, l)$	$m_2 = upd(m_1, a, c)$	$c = m[a]$
$l_1 < l_2$	$l = q$	$l_1 = s(l_2)$
$s = \{a\}$	$s_1 = s_2 \cup s_3$	$s_1 = s_2 \setminus s_3$
$x = \{e\}_{\mathrm{E}}$	$x_1 = x_2 \cup_{\mathrm{E}} x_3$	$x_1 = x_2 \setminus_{\mathrm{E}} x_3$
$a = A[l]$	$B = A\{l \leftarrow a\}$	
$p_1 \neq p_2$	$p = [a]$	$ordList(m, p)$
$s = path2set(p)$	$append(p_1, p_2, p_3)$	$\neg append(p_1, p_2, p_3)$
$s = addr2set(m, a, l)$	$p = getp(m, a_1, a_2, l)$	$skiplist(m, s, l, a_1, a_2)$

where e, e_1 and e_2 are elem*-variables; a, a_1 and a_2 are* addr*-variables; c is a* cell*-variable; m, m_1 and m_2 are* mem*-variables; p, p_1, p_2 and p_3 are* path*-variables; s, s_1, s_2 and s_3 are* setaddr*-variables; x, x_1, x_2 and x_3 are* setelem*-variables; A and B* array*-variables; l, l_1 and l_2 are* level*-variables and q is an* level *constant.* ⌟

The set of non-normalized literals consists of all flat literals not given in Definition 8.1.

Lemma 8.1:
Every non-normalized TSL-literal can be rewritten into an equivalent formula that contains only TSL-normalized literals. ⌟

Proof. As in the proof of Lemma 7.1 that allows to normalize $\mathsf{TSL_K}$ literals, we present equivalences and prove their validity. The left hand sides of these equivalences correspond to non-normalized TSL literals. Note that the right hand sides are quantifier free and contain only normalized literals. These equivalences allow to substitute non-normalized literals with equivalent formulas containing only normalized literals. We now present only the equivalences that correspond to new functions or predicates not present nor similar to the ones that belong to $\mathsf{TSL_K}$. These are:

$$e = c.data \quad \leftrightarrow \quad (\exists_{\mathsf{array}} A \ \exists_{\mathsf{level}} l) \quad [c = mkcell\,(e, A, l)] \tag{8.7}$$

$$A = c.arr \quad \leftrightarrow \quad (\exists_{\mathsf{elem}} e \ \exists_{\mathsf{level}} l) \quad [c = mkcell\,(e, A, l)] \tag{8.8}$$

$$l = c.max \quad \leftrightarrow \quad (\exists_{\mathsf{elem}} e \ \exists_{\mathsf{array}} A) \quad [c = mkcell\,(e, A, l)] \tag{8.9}$$

$$m_1 \neq_{\mathsf{mem}} m_2 \quad \leftrightarrow \quad (\exists_{\mathsf{addr}} a) \quad [m_1[a] \neq m_2[a]] \tag{8.10}$$

For normalizing the literal $\neg skiplist(m, r, l, a_i, a_e)$ we present the following equivalence. For simplicity in the presentation, the right hand-side contains non-normalized literals (not including $\neg skiplist$), which in turn can be normalized using the other equivalences:

$$\neg skiplist(m, r, l, a_i, a_e) \leftrightarrow$$

$$\left[\begin{array}{l}(\exists p : \mathsf{path}) \\ \qquad p = getp(m, a_i, null, 0) \wedge \neg ordList(m, p)\end{array}\right] \vee \quad \text{(NSL1)}$$

$$\left[\begin{array}{l}(\exists a : \mathsf{addr})(\exists c : \mathsf{cell}) \\ \qquad a \in r \ \wedge \ c = m[a] \ \wedge \ l < c.max\end{array}\right] \vee \quad \text{(NSL2)}$$

$$\left[\begin{array}{l}(\exists s : \mathsf{setaddr}) \\ \qquad s = addr2set(m, a_i, 0) \ \wedge \ s \neq r\end{array}\right] \vee \quad \text{(NSL3)}$$

$$\left[\begin{array}{l}(\exists l_1 : \mathsf{level})(\exists c : \mathsf{cell}) \\ \qquad 0 \leq l_1 \ \wedge \ l_1 \leq l \ \wedge \ c = m[a_e] \ \wedge \ c.next(l_1) \neq null\end{array}\right] \vee \quad \text{(NSL4)}$$

$$\left[\begin{array}{l}(\exists l_1 : \mathsf{level})(\exists s : \mathsf{setaddr}) \\ \qquad 0 \leq l_1 \ \wedge \ l_1 \leq l \ \wedge \ s = addr2set(m, a_i, l_1) \ \wedge \ a_e \notin s\end{array}\right] \vee \quad \text{(NSL5)}$$

$$\left[\begin{array}{l}(\exists l_{low}, l_{high} : \mathsf{level})(\exists p_{low}, p_{high} : \mathsf{path})(\exists s_{low}, s_{high} : \mathsf{setaddr}) \\ \qquad 0 \leq l_{low} \ \wedge \ l_{high} \leq l \quad \wedge \quad l_{high} = s(l_{low}) \quad \wedge \\ \qquad p_{low} = getp(m, a_i, a_e, l_{low}) \quad \wedge \quad s_{low} = path2set(p_{low}) \quad \wedge \\ \qquad p_{high} = getp(m, a_i, a_e, l_{high}) \quad \wedge \quad s_{high} = path2set(p_{high}) \quad \wedge \\ \qquad \neg(s_{high} \subseteq s_{low})\end{array}\right] \quad \text{(NSL6)}$$

For literals of the form $(m_1 \neq_{\mathsf{mem}} m_2)$, $(s_1 \neq_{\mathsf{setaddr}} s_2)$, $(s = \emptyset)$, $(s_3 = s_1 \cap s_2)$, $(s_1 \subseteq s_2)$, $(a \in s)$, $(x_1 \neq_{\mathsf{setelem}} x_2)$, $(x = \emptyset_{\mathrm{E}})$, $(x_3 = x_1 \cap_{\mathrm{E}} x_2)$, $(x_1 \subseteq_{\mathrm{E}} x_2)$, $(e \in_{\mathrm{E}} x)$, $(p = \epsilon)$, $(reach(m, a_1, a_2, l, p))$ and $(\neg ordList(m, p))$ the proof of the equivalences are identical to the one presented for $\mathsf{TSL_K}$ in Lemma 7.1. We prove the rest of the equivalences separately. For each case we assume a model $\mathcal{A}$ of the first literal and show that $\mathcal{A}$ is a model of the corresponding formula in the right hand side. Similarly, we assume a model $\mathcal{B}$ of the formula on the right and prove that $\mathcal{B}$ is a model of the literal on the left.

- Equivalence (8.7). Given $\mathcal{A}$, $c^{\mathcal{A}}$ is an element of $\mathcal{A}_{\mathsf{cell}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{array}} \times \mathcal{A}_{\mathsf{level}}$, so there are x in $\mathcal{A}_{\mathsf{elem}}$, w in $\mathcal{A}_{\mathsf{array}}$ and n in $\mathcal{A}_{\mathsf{level}}$ with $c^{\mathcal{A}} = (x, w, n)$. By the restriction on the class **TSL** of models, $\mathcal{A}$ must satisfy that $e^{\mathcal{A}} = x$, and $mkcell^{\mathcal{A}}(x, w, n) = c^{\mathcal{A}}$. Hence, by taking $A^{\mathcal{A}} = w$ and $l^{\mathcal{A}} = n$ the following holds in $\mathcal{A}$:

$$(\exists_{\mathsf{array}} A \ \exists_{\mathsf{level}} l) \left[c = mkcell\,(e, A, l)\right]$$

  For the other direction, we assume $\mathcal{B}$ is a model of

$$(\exists_{\mathsf{array}} A \ \exists_{\mathsf{level}} l) \left[c = mkcell\,(e, A, l)\right]$$

  It follows, by the interpretation in $\mathcal{B}$ of $data$, that $c^{\mathcal{B}}.data^{\mathcal{B}}$ is $e^{\mathcal{B}}$ as desired.

- Equivalences (8.8) and (8.9). The proof is analogous to (8.7).
- Equivalence (8.10). This equivalence follows easily from the extensionality principle for

arrays.

- $\neg skiplist(m, r, l, a_i, a_e)$. The translation encodes all conditions that are needed in order to violate the skiplist shape according to the restriction to TSL interpretations. Specifically:
  - Formula (NSL1) captures whether the path that connects address a_i to address a_e at level 0 is not ordered.
  - Formula (NSL2) holds whenever there is a node in the skiplist whose height is higher than the maximum level of the skiplist.
  - Formula (NSL3) captures whether some node of the skiplist is stored at an address which does not belong to region r or whether some address in r is not reachable from a_i. Equivalently, whether r is indeed the region of the skiplist.
  - Formula (NSL4) captures whether a_e, that is the last node of the skiplist, at any level between 0 and l is not null terminated.
  - Formula (NSL5) holds whenever address a_e cannot be reached from a_i at any level between 0 and l.
  - Finally, formula (NSL5) is satisfied whenever some node at some level is not included into the set of nodes at the level immediately below.

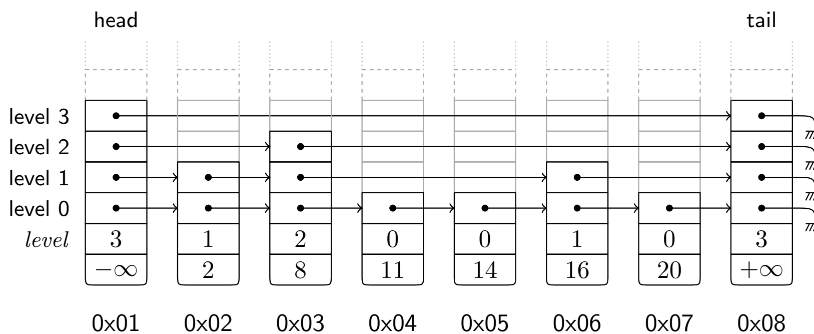
  Predicates (NSL1), (NSL2), (NSL3), (NSL4), (NSL5) and (NSL6) are obtained by negating the interpretation of skiplist in TSL and splitting all cases of (8.1).

  It follows that if a model $\mathcal{A}$ of literal $\neg skiplist(m, r, l, a_i, a_e)$ holds, then $\mathcal{A}$ is also a model of the formula obtained by the disjunction of (NSL1) to (NSL6). Similarly, if we assume a model $\mathcal{B}$ then it is also easy to see that it is a model of the literal $\neg skiplist(m, r, l, a_i, a_e)$.

Hence, all the equivalences presented above are valid TSL equivalences, which concludes the proof. □

Orienting these equivalences from left to right allows us to eliminate non-flat literals from a given TSL formula, resulting in a TSL formula that only contains normalized literals. Note also that, once again, all quantifiers introduced in these equivalences are existential quantifiers, which can be pushed (with renaming to avoid capturing if necessary) to the front of the formula. Hence, a quantifier-free formula (which is implicitly existentially quantified) results into a quantifier-free formula after the rewriting step.

Example 8.4
We now present some TSL formulas we will use as running examples for the rest of the section. Consider once again the following skiplist, presented in Fig. 8.2:



and the following formula ψ_{SAT}:

$$\psi_{\mathsf{SAT}} \quad : \quad i = 0 \ \wedge \ A = heap[head].arr \ \wedge \ B = A\{i \leftarrow tail\}$$

This formula establishes that A is the array stored at the node pointed by $head$ in the memory $heap$ and that B is the array obtained by replacing in array A the pointer at level i with $tail$. Clearly, ψ_{SAT} describes a formula which is satisfiable. We can now normalize ψ_{SAT}, obtaining:

$$\psi_{\mathsf{SAT}}^{\mathrm{norm}} \quad : \quad i = 0 \ \wedge \ \begin{pmatrix} c = heap[head] \ \wedge \\ c = mkcell(e, A, l) \end{pmatrix} \ \wedge \ B = A\{i \leftarrow tail\}.$$

Let us now consider a little more complex TSL formula ψ_{UNSAT}, which is unsatisfiable:

$$\psi_{\mathsf{UNSAT}} \quad : \quad \begin{cases} skiplist(heap, reg, l, head, tail) \wedge l = 3 & \wedge \qquad \text{(U1)} \\ head \neq null \wedge tail \neq null \wedge head \neq tail & \wedge \qquad \text{(U2)} \\ i < l \wedge a \neq tail \wedge a \neq null \wedge a \in addr2set(heap, head, i) & \wedge \qquad \text{(U3)} \\ c.arr = heap[a].arr\{i \leftarrow null\} \wedge m = upd(heap, a, c) & \wedge \qquad \text{(U4)} \\ skiplist(m, reg, l, head, tail) & \qquad\quad \text{(U5)} \end{cases}$$

The formula ψ_{UNSAT} describes a situation in which an incorrect manipulation of a skiplist of height 3 (like the one depicted in Fig. 8.2) leads to a data structure which does not satisfy the skiplist shape property anymore. In ψ_{UNSAT}, we have that:

- Condition (U1) states that there is a skiplist of height 3 stored in the memory $heap$ which goes from address $head$ up to address $tail$.
- Condition (U2) sets some conditions for the addresses $head$ and $tail$, like they must not be equal between them, nor equal to $null$.
- Condition (U3) establishes the presence of an address a, different from $tail$ and $null$, which is reachable form $head$ at an arbitrary level i which is strictly lower than l.
- Condition (U4) says that c is a cell whose array of pointers arr matches the array of pointers stored at the cell pointed by a in $heap$, except for the address at level i, which is set to $null$. This condition also establishes that memory m is a copy of memory $heap$, except that the cell at address a is replaced by cell c.
- Finally, condition (U5) says that in memory m has the shape of a skiplist of height 3 that goes from $head$ up to $tail$ and whose cells are stored in the addresses that make the set of addresses reg.

Formula ψ_{UNSAT} is unsatisfiable because conditions (U1), (U2), (U3) and (U4) clearly break the skiplist layout of the data structure, and hence condition (U5) cannot hold. The key to make formula ψ_{UNSAT} unsatisfiable is that we are making null terminated a level lower than l at a cell that is not $tail$. This change prevents the set of addresses reachable from $head$ in the level immediately on top of level i to be included into the set of addresses reachable from $head$ at level i, which is one of the required conditions for a layout to have the shape of a skiplist.

We can now normalize formula ψ_{UNSAT}, obtaining $\psi^{\mathrm{norm}}_{\mathsf{UNSAT}}$:

$$\psi^{\mathrm{norm}}_{\mathsf{UNSAT}} \quad : \quad \begin{cases} \mathit{skiplist}(\mathit{heap}, \mathit{reg}, l, \mathit{head}, \mathit{tail}) \wedge l = 3 & \wedge \qquad \text{(UN1)} \\ n = \mathit{null} \wedge \mathit{head} \neq n \wedge \mathit{tail} \neq n \wedge \mathit{head} \neq \mathit{tail} & \wedge \qquad \text{(UN2)} \\ i < l \wedge \begin{pmatrix} a \neq \mathit{tail} \wedge \\ a \neq n \end{pmatrix} \wedge \begin{pmatrix} s = \mathit{addr2set}(\mathit{heap}, \mathit{head}, i) \\ \wedge \\ a \in s \end{pmatrix} & \wedge \qquad \text{(UN3)} \\ \begin{pmatrix} d = \mathit{heap}[a] \quad \wedge \\ d = \mathit{mkcell}(e_1, A, j_1) \wedge \\ c = \mathit{mkcell}(e_2, B, j_2) \wedge \\ B = A\{i \leftarrow n\} \end{pmatrix} \wedge m = \mathit{upd}(\mathit{heap}, a, c) & \wedge \qquad \text{(UN4)} \\ \mathit{skiplist}(m, \mathit{reg}, l, \mathit{head}, \mathit{tail}) & \qquad\quad \text{(UN5)} \end{cases}$$

Here (UN1), (UN2), (UN3), (UN4) and ,(UN5) correspond to the normalized version of (U1), (U2), (U3), (U4) and (U5) respectively. ⌟

Following the result of Lemma 8.1, the resulting formula after normalization can then be converted into its disjunctive normal form, what let us obtain the following result.

Lemma 8.2 (Normalized Literals):
Every TSL-formula is equivalent to a disjunction of conjunctions of normalized TSL-literals. ⌟

8.4 Decidability of TSL

We now proceed to present the decision procedure for quantifier-free TSL normalized formulas sketched in Fig. 8.7. This procedure reduces the decision problem of quantifier-free TSL formulas to the satisfiability of quantifier-free $\mathsf{TSL_K}$ formulas and quantifier-free Presburger arithmetic formulas.

Our procedure starts from a TSL formula φ expressed as a normalized conjunction of literals. The main idea is to guess a feasible arrangement between level variables, and then extract from φ the *relevant levels*, generating a $\mathsf{TSL_K}$ formula that uses only relevant levels and respects the guessed arrangement. The original TSL formula is satisfiable and compatible with the arrangement if and only if the generated $\mathsf{TSL_K}$ formula is satisfiable. To formally prove this, we proceed constructively populating the missing levels (after ignoring the literals $l = q$) by cloning intermediate levels as needed.

The main problem to perform this reduction is that literals of the form $l = q$ for some constant q (like $l = 3$ for example) preclude the removal of levels and the generation of the $\mathsf{TSL_K}$ formula. Our procedure includes a transformation that preservers satisfiability and allows to exclude literals of the form $l = q$ in the formula generated that is subsequently reduced to $\mathsf{TSL_K}$ (see Fig. 8.8).

We now proceed to describe each step of the decision procedure separately.

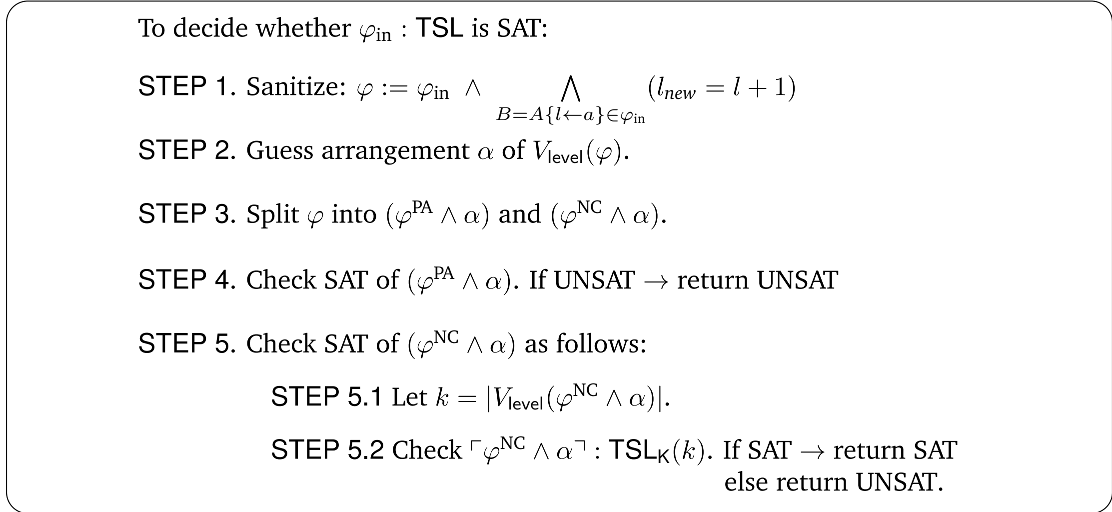


Figure 8.7: Steps of the decision procedure for the satisfiability of TSL formulas.

8.4.1 STEP 1: Sanitization

The decision procedure begins by sanitizing the normalized collection of literals received as input. The sanitization process guarantees that the satisfiability is preserved in the split step. Formally, this preservation will be proven by showing that a model of the original formula can be built from a model of the split formulas, populating the removed intermediate levels.

Definition 8.2 (Sanitized).
A conjunction of normalized literals is sanitized if for every literal $B = A\{l \leftarrow a\}$ there is a literal of the form $l_2 = l + 1$. ⌟

A formula can be easily sanitized by adding a fresh variable l_{new} and a literal $l_{new} = l + 1$ for every literal $B = A\{l \leftarrow a\}$ in case there is no literal of the form $l_2 = l + 1$ for some l_2 already in the formula. The fresh level variables in sanitized formulas will be later used in the proof of Theorem 8.1 to construct a proper model of the TSL formula by replicating level l_{new} instead of level l. Sanitization allows to show the existence of models with constants from models of sub-formulas without constants.

Sanitizing a formula does not affect its satisfiability because it only adds an arithmetic constraint $(l_{new} = l + 1)$ where l_{new} is a fresh new level variable. Hence, a model of φ (the

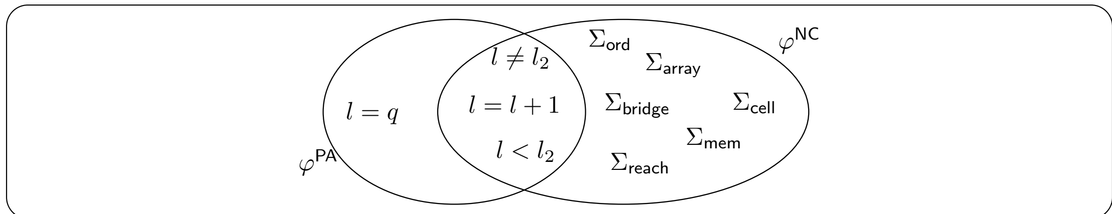


Figure 8.8: A split of φ obtained after STEP 1 into φ^{PA} and φ^{NC}.

sanitized formula) is a model for φ_{in} (the input formula), and from a model of φ_{in} one can immediately build a model of φ by computing the values of the fresh variables l_{new}.

Example 8.5
Consider again the normalized formulas $\psi^{\text{norm}}_{\mathsf{SAT}}$ and $\psi^{\text{norm}}_{\mathsf{UNSAT}}$ introduced in Example 8.4. After sanitizing these formulas we obtain $\psi^{\text{sanit}}_{\mathsf{SAT}}$ and $\psi^{\text{sanit}}_{\mathsf{UNSAT}}$:

$$\begin{array}{lcl} \psi^{\text{sanit}}_{\mathsf{SAT}} & : & \psi^{\text{norm}}_{\mathsf{SAT}} \wedge l_{new} = i+1 \\ \psi^{\text{sanit}}_{\mathsf{UNSAT}} & : & \psi^{\text{norm}}_{\mathsf{UNSAT}} \wedge l_{new} = i+1 \end{array}$$
⌟

8.4.2 STEP 2: Order Arrangements

A model of a TSL formula maps a level variable to a natural number. Hence, every two variables are either assigned the same value or their values are ordered by the usual order between natural numbers. That is, in a given model $\mathcal{M}$ of a formula φ, for every pair of level variables l_1 and l_2 occurring in φ, there is exactly one of the formulas $l_1 = l_2$, $l_1 < l_2$ and $l_2 < l_1$ that hold in $\mathcal{M}$. We call the arithmetic formulas that capture the relations between level variables an *order arrangement*.

Definition 8.3 (Order Arrangement).
Given a sanitized formula φ, an order arrangement is a collection of literals containing, for every pair of level variables $l_1, l_2 \in V_{\mathsf{level}}(\varphi)$, exactly one of the following:

$$l_1 = l_2 \qquad\qquad l_1 < l_2 \qquad\qquad l_2 < l_1$$
⌟

For instance, an order arrangement of $\psi^{\text{sanit}}_{\mathsf{SAT}}$ is $\{i < l_{new}, i < l, l_{new} < l\}$. Since there is a finite number of level variables in a formula φ, there is a finite number of order arrangements. Note also that a formula φ is satisfiable if and only if there is an order arrangement α such that $\varphi \wedge \alpha$ is satisfiable. STEP 2 of the decision procedure consists of guessing an order arrangement α for the given sanitized formula.

8.4.3 STEP 3: Split

This step of the decision procedure begins by splitting the sanitized formula φ into φ^{PA}, which contains precisely all those literals in the theory of arithmetic Σ_{level}, and φ^{NC} containing all literals from φ except those involving constants of sort level $(l = q)$. Clearly, φ is equivalent to $\varphi^{\text{NC}} \wedge \varphi^{\text{PA}}$.

Example 8.6
Consider the sanitized formulas $\psi^{\text{sanit}}_{\mathsf{SAT}}$ and $\psi^{\text{sanit}}_{\mathsf{UNSAT}}$ defined in Example 8.5. In the case of $\psi^{\text{sanit}}_{\mathsf{SAT}}$, the formula is split into $\psi^{\text{PA}}_{\mathsf{SAT}}$ and $\psi^{\text{NC}}_{\mathsf{SAT}}$:

$$\begin{array}{lcl} \psi^{\text{PA}}_{\mathsf{SAT}} & : & i = 0 \wedge l_{new} = i+1 \\ \psi^{\text{NC}}_{\mathsf{SAT}} & : & \begin{pmatrix} c = heap[head] \ \wedge \\ c = mkcell(e, A, l) \end{pmatrix} \wedge B = A\{i \leftarrow tail\} \wedge l_{new} = i+1 \end{array}$$

Similarly, formula $\psi^{\text{sanit}}_{\mathsf{UNSAT}}$ is split into $\psi^{\text{PA}}_{\mathsf{UNSAT}}$ and $\psi^{\text{NC}}_{\mathsf{UNSAT}}$:

$$\psi^{\text{PA}}_{\mathsf{UNSAT}} \quad : \quad l = 3 \wedge i < l \wedge l_{new} = i + 1$$

$$\psi^{\text{NC}}_{\mathsf{UNSAT}} \quad : \quad \begin{cases} skiplist(heap, reg, l, head, tail) & \wedge \\ n = null \wedge head \neq n \wedge tail \neq n \wedge head \neq tail & \wedge \\ \begin{pmatrix} i < l \ \wedge \\ l_{new} = i + 1 \end{pmatrix} \wedge \begin{pmatrix} a \neq tail \wedge \\ a \neq n \end{pmatrix} \wedge \begin{pmatrix} s = addr2set(heap, head, i) \\ \wedge \\ a \in s \end{pmatrix} & \wedge \\ \begin{pmatrix} d = heap[a] \ \wedge \\ d = mkcell(e_1, A, j_1) \wedge \\ c = mkcell(e_2, B, j_2) \wedge \\ B = A\{i \leftarrow n\} \end{pmatrix} \wedge m = upd(heap, a, c) & \wedge \\ skiplist(m, reg, l, head, tail) & \end{cases}$$

STEP 3 uses the order arrangement guessed in STEP 2 to reduce the satisfiability of a sanitized formula φ that follows an order arrangement α into:

1) the satisfiability of a Presburger arithmetic formula $(\varphi^{\text{PA}} \wedge \alpha)$, which will be checked later in STEP 4; and

2) the satisfiability of a sanitized formula without constants $(\varphi^{\text{NC}} \wedge \alpha)$, which will be checked in STEP 5.

We now show that φ is satisfiable if and only if for some arrangement α bot $(\varphi^{\text{PA}} \wedge \alpha)$ and $(\varphi^{\text{NC}} \wedge \alpha)$ are satisfiable.

An essential notion to show the correctness of this split and reduction is that of a *gap*, which is a level in a model that is not named explicitly by a level variable.

Definition 8.4 (Gap).
Let $\mathcal{A}$ be a model of φ. We say that a natural number n is a gap *in $\mathcal{A}$ if there are variables l_1, l_2 in $V_{\mathsf{level}}(\varphi)$ such that $l_1^{\mathcal{A}} < n < l_2^{\mathcal{A}}$, but there is no variable l in $V_{\mathsf{level}}(\varphi)$ with $l^{\mathcal{A}} = n$.*

Example 8.7
Consider the formula $\psi^{\text{sanit}}_{\mathsf{SAT}}$ introduced in Example 8.5, for which

$$V_{\mathsf{level}}(\psi^{\text{sanit}}_{\mathsf{SAT}}) = \{i, l_{new}, l\}.$$

A model $\mathcal{A}_{\psi^{\text{sanit}}_{\mathsf{SAT}}}$ of $\psi^{\text{sanit}}_{\mathsf{SAT}}$ that interprets variables i, l_{new} and l as 0, 1 and 3 respectively has a gap at 2.

A gap-less model is a model without gaps, either between two level variables or above any level variable.

Definition 8.5 (Gap-less model).
A model $\mathcal{A}$ of φ is a gap-less model whenever it has no gaps, and for every array C in the domain $\mathsf{array}^{\mathcal{A}}$*:*

$$C(n) = null$$

for every n with $n > l^{\mathcal{A}}$ for all $l \in V_{\mathsf{level}}(\varphi)$. ⌟

We prove now the existence of a gap-less model given that there is a model. We first need one last auxiliary notion to ease the construction of similar models, by setting a condition under which reachability at different levels is preserved.

Definition 8.6 (Sort agreement).
Two interpretations $\mathcal{A}$ and $\mathcal{B}$ of a formula φ agree *on sorts σ whenever all of the following hold:*

1. *$\mathcal{A}_\sigma = \mathcal{B}_\sigma$. That is, the domains for sorts σ coincide.*
2. *For every $v \in V_\sigma(\varphi)$, $v^{\mathcal{A}} = v^{\mathcal{B}}$. That is, the interpretations of variables coincide.*
3. *For every function symbol f with domain and co-domain from sorts in σ, $f^{\mathcal{A}} = f^{\mathcal{B}}$ and for every predicate symbol P with domain in σ, $P^{\mathcal{A}}$ iff $P^{\mathcal{B}}$.* ⌟

Lemma 8.3:
Let $\mathcal{A}$ and $\mathcal{B}$ be two interpretations of a sanitized formula φ that agree on sorts addr, elem, path, setaddr, setelem, *and such that for every level variable $l \in V_{\mathsf{level}}(\varphi)$, $m \in V_{\mathsf{mem}}(\varphi)$, and $a \in \mathcal{A}_{\mathsf{addr}}$:*

$$m^{\mathcal{A}}(a).arr^{\mathcal{A}}(l^{\mathcal{A}}) = m^{\mathcal{B}}(a).arr^{\mathcal{B}}(l^{\mathcal{B}}) \tag{8.11}$$

It follows that, for every $p \in V_{\mathsf{path}}(\varphi)$, $a_{init} \in V_{\mathsf{addr}}(\varphi)$ and $a_{end} \in V_{\mathsf{addr}}(\varphi)$:

$$reach^{\mathcal{A}}(m^{\mathcal{A}}, a^{\mathcal{A}}_{init}, a^{\mathcal{A}}_{end}, l^{\mathcal{A}}, p^{\mathcal{A}}) \quad \textit{if and only if} \quad reach^{\mathcal{B}}(m^{\mathcal{B}}, a^{\mathcal{B}}_{init}, a^{\mathcal{B}}_{end}, l^{\mathcal{B}}, p^{\mathcal{B}}).$$ ⌟

Proof. Let $\mathcal{A}$ and $\mathcal{B}$ be two interpretations of φ satisfying the conditions established in the statement of Lemma 8.3, and assume that $reach^{\mathcal{A}}(m^{\mathcal{A}}, a^{\mathcal{A}}_{init}, a^{\mathcal{A}}_{end}, l^{\mathcal{A}}, p^{\mathcal{A}})$ holds for some $a_{init}, a_{end} \in V_{\mathsf{addr}}(\varphi)$, $m \in V_{\mathsf{mem}}(\varphi)$ and $p \in V_{\mathsf{path}}(\varphi)$. Note that, by assumption, $a^{\mathcal{A}}_{init} = a^{\mathcal{B}}_{init}$, $a^{\mathcal{A}}_{end} = a^{\mathcal{B}}_{end}$ and $p^{\mathcal{A}} = p^{\mathcal{B}}$. We consider the cases for $p^{\mathcal{A}}$:

(1) If $p^{\mathcal{A}} = \epsilon$ then $a^{\mathcal{A}}_{init} = a^{\mathcal{A}}_{end}$. Consequently, $p^{\mathcal{B}} = \epsilon$ and $a^{\mathcal{B}}_{init} = a^{\mathcal{B}}_{end}$, so for interpretation $\mathcal{B}$, the predicate $reach^{\mathcal{A}}(m^{\mathcal{B}}, a^{\mathcal{B}}_{init}, a^{\mathcal{B}}_{end}, l^{\mathcal{B}}, p^{\mathcal{B}})$ also holds.

(2) The other case is: $p = [a_1, \ldots, a_n]$ with $a_1 = a_{init}$ and $m^{\mathcal{A}}(a_n).arr^{\mathcal{A}}(l^{\mathcal{A}}) = a_{end}$, and for every $r < n$, $m^{\mathcal{A}}(a_r).arr^{\mathcal{A}}(l^{\mathcal{A}}) = a_{r+1}$. It follows, by

$$m^{\mathcal{A}}(a).arr^{\mathcal{A}}(l^{\mathcal{A}}) = m^{\mathcal{B}}(a).arr^{\mathcal{B}}(l^{\mathcal{B}})$$

that $m^{\mathcal{B}}(a_n).arr^{\mathcal{B}}(l^{\mathcal{B}}) = a_{end}$, and for every $r < n$, $m^{\mathcal{B}}(a_r).arr^{\mathcal{B}}(l^{\mathcal{B}}) = a_{r+1}$. Hence, $reach^{\mathcal{A}}(m^{\mathcal{B}}, a^{\mathcal{B}}_{init}, a^{\mathcal{B}}_{end}, l^{\mathcal{B}}, p^{\mathcal{B}})$.

The other direction follows by symmetry. □

At the end of the split phase in STEP 3 we obtain a sanitized formula without constants. We show now that if a sanitized formula without constants has a model then it has a model without gaps.

Lemma 8.4 (Gap-reduction):
Let $\mathcal{A}$ be a model of a sanitized formula φ without constants, and let $\mathcal{A}$ have a gap at n. Then, there is a model $\mathcal{B}$ of φ such that, for every $l \in V_{\mathsf{level}}(\varphi)$:

$$l^{\mathcal{B}} = \begin{cases} l^{\mathcal{A}} & \text{if } l^{\mathcal{A}} < n \\ l^{\mathcal{A}} - 1 & \text{if } l^{\mathcal{A}} > n \end{cases}$$

That is, the number of gaps in $\mathcal{B}$ is one less than in $\mathcal{A}$. ⌟

Proof. Let $\mathcal{A}$ be a model of φ with a gap at n. We build a model $\mathcal{B}$ with the condition in the lemma as follows. The model $\mathcal{B}$ agrees with $\mathcal{A}$ on addr, elem, path, setaddr and setelem. In particular, $v^{\mathcal{B}} = v^{\mathcal{A}}$ for variables of these sorts. For the other sorts $\sigma \in \{\mathsf{level}, \mathsf{array}, \mathsf{cell}, \mathsf{mem}\}$, we let $\mathcal{B}_\sigma = \mathcal{A}_\sigma$. We define the following transformation maps that capture how the elements in the domains in $\mathcal{B}$ are built from elements in the corresponding domain in $\mathcal{A}$:

$$\beta_{\mathsf{level}}(j) = \begin{cases} j & \text{if } j < n \\ j-1 & \text{otherwise} \end{cases} \qquad \beta_{\mathsf{array}}(A)(i) = \begin{cases} A(i) & \text{if } i < n \\ A(i+1) & \text{if } i \geq n \end{cases}$$

$$\beta_{\mathsf{cell}}((e, A, l)) = (e, \beta_{\mathsf{array}}(A), \beta_{\mathsf{level}}(l)) \qquad \beta_{\mathsf{mem}}(m)(a) = \beta_{\mathsf{cell}}(m(a))$$

Now we are ready to define the valuations of variables $l \in V_{\mathsf{level}}(\varphi)$, $A \in V_{\mathsf{array}}(\varphi)$, $c \in V_{\mathsf{cell}}(\varphi)$ and $m \in V_{\mathsf{mem}}(\varphi)$:

$$l^{\mathcal{B}} = \beta_{\mathsf{level}}(l^{\mathcal{A}}) \qquad A^{\mathcal{B}} = \beta_{\mathsf{array}}(A^{\mathcal{A}}) \qquad c^{\mathcal{B}} = \beta_{\mathsf{cell}}(c^{\mathcal{A}}) \qquad m^{\mathcal{B}} = \beta_{\mathsf{mem}}(m^{\mathcal{A}})$$

The interpretation of all functions and predicates is preserved from $\mathcal{A}$ into $\mathcal{B}$.

The next step is to show that $\mathcal{B}$ is indeed a model of φ. All literals of the following form hold in $\mathcal{B}$ if and only if they hold in $\mathcal{A}$, because the valuations and interpretations of functions and predicates of the corresponding sorts are preserved:

$e_1 \neq e_2$	$a_1 \neq a_2$	$l_1 \neq l_2$
$e_1 \preceq e_2$	$a = null$	$c = error$
$l_1 < l_2$	$l = q$	
$s = \{a\}$	$s_1 = s_2 \cup s_3$	$s_1 = s_2 \setminus s_3$
$x = \{e\}_{\mathrm{E}}$	$x_1 = x_2 \cup_{\mathrm{E}} x_3$	$x_1 = x_2 \setminus_{\mathrm{E}} x_3$
$p_1 \neq p_2$	$p = [a]$	$ordList(m, p)$
$s = path2set(p)$	$append(p_1, p_2, p_3)$	$\neg append(p_1, p_2, p_3)$

Literals of the form $c = m[a]$ and $m_2 = upd(m_1, a, c)$ hold in $\mathcal{B}$ whenever they do in $\mathcal{A}$, because the same transformations are performed on both sides of the equation.

Finally, we show the remaining literals:

- Literal $c = mkcell(e, A, l)$. Assuming $c^{\mathcal{A}} = mkcell(e^{\mathcal{A}}, A^{\mathcal{A}}, l^{\mathcal{A}})$ we have that:

$$mkcell(e^{\mathcal{B}}, A^{\mathcal{B}}, l^{\mathcal{B}}) \quad = \quad mkcell(e^{\mathcal{A}}, \beta_{\mathsf{array}}(A^{\mathcal{A}}), \beta_{\mathsf{level}}(l^{\mathcal{A}})) \quad = \quad \beta_{\mathsf{cell}}(c^{\mathcal{A}}) \quad = \quad c^{\mathcal{B}}$$

- Literal $a = A[l]$. Assuming $a^{\mathcal{A}} = A^{\mathcal{A}}[l^{\mathcal{A}}]$, we have two possible cases for $l^{\mathcal{A}}$:

  1. Case $l^{\mathcal{A}} < n$. In this case, we have that:

$$A^{\mathcal{B}}[l^{\mathcal{B}}] \quad = \quad A^{\mathcal{A}}[l^{\mathcal{A}}] \quad = \quad a^{\mathcal{A}} \quad = \quad a^{\mathcal{B}}$$

  2. Case $l^{\mathcal{A}} > n$. In this case, we have that:

$$A^{\mathcal{B}}[l^{\mathcal{B}}] \quad = \quad A^{\mathcal{A}}[(l^{\mathcal{A}} - 1) + 1] \quad = \quad A^{\mathcal{A}}[l^{\mathcal{A}}] \quad = \quad a^{\mathcal{A}} \quad = \quad a^{\mathcal{B}}$$

  Note that it is critical that the removed level was not named by a variable (which is true because we are removing a gap), so the two cases above are exhaustive.

- Literal $B = A\{l \leftarrow a\}$. Assume that $B^{\mathcal{A}} = A^{\mathcal{A}}\{l^{\mathcal{A}} \leftarrow a^{\mathcal{A}}\}$ and consider an arbitrary $m \in \mathbb{N}$. We have three possible cases:

  1. Case $m = l^{\mathcal{B}}$. In this case, we have that:

$$(A^{\mathcal{B}}\{l^{\mathcal{B}} \leftarrow a^{\mathcal{B}}\})(m) \quad = \quad (\beta_{\mathsf{array}}(A^{\mathcal{A}})\{l^{\mathcal{B}} \leftarrow a^{\mathcal{B}}\})(m) \quad = \quad a^{\mathcal{B}}$$

  2. Case $m \neq l^{\mathcal{B}}$ and $m < n$. In this case, we have that:

$$\begin{aligned} (A^{\mathcal{B}}\{l^{\mathcal{B}} \leftarrow a^{\mathcal{B}}\})(m) &= (\beta_{\mathsf{array}}(A^{\mathcal{A}})\{l^{\mathcal{B}} \leftarrow a^{\mathcal{B}}\})(m) \\ &= (\beta_{\mathsf{array}}(A^{\mathcal{A}}))(m) \\ &= A^{\mathcal{A}}(m) \\ &= B^{\mathcal{A}}(m) \\ &= \beta_{\mathsf{array}}(B^{\mathcal{A}}(m)) \\ &= B^{\mathcal{B}}(m) \end{aligned}$$

  3. Case $m \neq l^{\mathcal{B}}$ and $m \geq n$. In this case, we have:

$$\begin{aligned} (A^{\mathcal{B}}\{l^{\mathcal{B}} \leftarrow a^{\mathcal{B}}\})(m) &= (\beta_{\mathsf{array}}(A^{\mathcal{A}})\{l^{\mathcal{B}} \leftarrow a^{\mathcal{B}}\})(m) \\ &= (\beta_{\mathsf{array}}(A^{\mathcal{A}}))(m) \\ &= A^{\mathcal{A}}(m+1) \\ &= B^{\mathcal{A}}(m+1) \\ &= \beta_{\mathsf{array}}(B^{\mathcal{A}})(m) \\ &= B^{\mathcal{B}}(m) \end{aligned}$$

  Again, the cases are exhaustive because the removed level is not named by any level variable, including l.

- Literals $s = addr2set(m, a, l)$ and $p = getp(m, a_1, a_2, l)$. We first prove that for all variables m and l, elements of the address domain a_{init} and a_{end} and paths p, $reach(m^{\mathcal{A}}, a_{init}, a_{end}, l^{\mathcal{A}}, p)$ if and only if $reach(m^{\mathcal{B}}, a_{init}, a_{end}, l^{\mathcal{B}}, p)$. Assume $reach(m^{\mathcal{A}}, a_{init}, a_{end}, l^{\mathcal{A}}, p)$, then either $a_{init} = a_{end}$ and $p = \epsilon$, in which case $reach(m^{\mathcal{B}}, a_{init}, a_{end}, l^{\mathcal{B}}, p)$, or there is a sequence of addresses $a_1, \ldots, a_k$ with

  (a) $p = [a_1 \ldots a_k]$

  (b) $a_1 = a_{init}$

  (c) $m^{\mathcal{A}}(a_r).arr^{\mathcal{A}}(l^{\mathcal{A}}) = a_{r+1}$, for $r < k$

  (d) $m^{\mathcal{A}}(a_k).arr^{\mathcal{A}}(l^{\mathcal{A}}) = a_{end}$

  Consider an arbitrary $r < k$. Either $l^{\mathcal{A}} < n$ or $l^{\mathcal{A}} > n$ (recall that $l^{\mathcal{A}}$ is either strictly under or strictly over the gap). In either case,

  $$m^{\mathcal{B}}(a_r).arr^{\mathcal{B}}(l^{\mathcal{B}}) \quad = \quad m^{\mathcal{A}}(a_r).arr^{\mathcal{A}}(l^{\mathcal{A}}) \quad = \quad a_{r+1}$$

  And also,

  $$m^{\mathcal{B}}(a_k).arr^{\mathcal{B}}(l^{\mathcal{B}}) \quad = \quad m^{\mathcal{B}}(a_k).arr^{\mathcal{B}}(l^{\mathcal{B}}) = a_{end}.$$

  Hence, conditions (a), (b), (c) and (d) hold for $\mathcal{B}$ and then $reach(m^{\mathcal{B}}, a_{init}, a_{end}, l^{\mathcal{B}}, p)$. Essentially, predicate $reach$ only depends on pointers at level l and all relevant connectivity properties at level l are preserved from interpretation $\mathcal{A}$ to interpretation $\mathcal{B}$. The other direction holds similarly. From the preservation of the $reach$ predicate it follows that, if $addr2set(m^{\mathcal{A}}, a^{\mathcal{A}}, l^{\mathcal{A}}) = s^{\mathcal{A}}$ then

  $$\begin{aligned} addr2set(m^{\mathcal{B}}, a^{\mathcal{B}}, l^{\mathcal{B}}) &= \{a' \mid \exists p \in \mathcal{B}_{\mathsf{path}} \,.\, (m, a, a', l, p \in reach^{\mathcal{B}}\} \\ &= \{a' \mid \exists p \in \mathcal{A}_{\mathsf{path}} \,.\, (m, a, a', l, p \in reach^{\mathcal{A}}\} \\ &= addr2set(m^{\mathcal{A}}, a^{\mathcal{A}}, l^{\mathcal{A}}) \\ &= s^{\mathcal{A}} \\ &= s^{\mathcal{B}} \end{aligned}$$

  Finally, assume that $p^{\mathcal{A}} = getp(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, l^{\mathcal{A}})$. If $(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, l^{\mathcal{A}}, p^{\mathcal{A}}) \in reach^{\mathcal{A}}$ then $(m^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, l^{\mathcal{B}}, p^{\mathcal{B}}) \in reach^{\mathcal{B}}$ and hence $p^{\mathcal{B}} = getp(m^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, l^{\mathcal{B}})$. The other possible case is that $\epsilon = getp(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, l^{\mathcal{A}})$ when

  $$\text{for no path } p, \qquad (m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, l^{\mathcal{A}}, p) \in reach^{\mathcal{A}}$$

  but then also

  $$\text{for no path } p, \qquad (m^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, l^{\mathcal{B}}, p) \in reach^{\mathcal{B}}$$

  and then $\epsilon = getp(m^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, l^{\mathcal{B}})$, as desired.

- Literal $skiplist(m, r, l, a_1, a_2)$. We assume $skiplist(m^{\mathcal{A}}, r^{\mathcal{A}}, l^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}})$. This implies:

i) $ordList^{\mathcal{A}}(m^{\mathcal{A}}, getp^{\mathcal{A}}(a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, 0))$.
Let p be an element of $\mathcal{A}_{\mathsf{path}}$ such that $p = getp^{\mathcal{A}}(a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, 0))$. As shown previously, $p = getp^{\mathcal{B}}(a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, 0))$. Then

$$ordList^{\mathcal{B}}(m^{\mathcal{B}}, getp^{\mathcal{B}}(a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, 0))$$

holds because

$$ordList^{\mathcal{A}}(m^{\mathcal{A}}, getp^{\mathcal{A}}(a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, 0))$$

holds.

ii) $r^{\mathcal{A}} = path2set^{\mathcal{A}}(getp^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, 0))$.
In this case, we have that

$$r^{\mathcal{B}} = path2set^{\mathcal{B}}(getp^{\mathcal{B}}(m^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, 0))$$

because we have that

$$getp^{\mathcal{B}}(m^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, 0) = getp^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, 0)$$

iii) for every $a \in r^{\mathcal{A}}$, $m^{\mathcal{A}}(a^{\mathcal{A}}).max^{\mathcal{A}} \leq l^{\mathcal{A}}$.
Since $r^{\mathcal{B}} = r^{\mathcal{A}}$ and $m^{\mathcal{B}}(a) = \beta_{\mathsf{cell}}(m^{\mathcal{A}}(a))$, it is enough to consider two cases:
1) $m^{\mathcal{A}}(a).max^{\mathcal{A}} = l^{\mathcal{A}}$, in which case $m^{\mathcal{B}}(a).max^{\mathcal{B}} = l^{\mathcal{B}}$.
2) $m^{\mathcal{A}}(a).max^{\mathcal{A}} < l^{\mathcal{A}}$, in which case $m^{\mathcal{A}}(a).max^{\mathcal{A}} \leq l^{\mathcal{A}}$.

iv) If $(0 = l^{\mathcal{A}})$ then $(0 = l^{\mathcal{B}})$.

v) If $(0 < l^{\mathcal{A}})$, and for all i within $0 < i < l^{\mathcal{A}}$:

$$\begin{array}{r}
m^{\mathcal{A}}(a_2^{\mathcal{A}}).arr^{\mathcal{A}}(i) = null^{\mathcal{A}} \\
a_2^{\mathcal{A}} \in addr2set^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, i) \\
path2set^{\mathcal{A}}(getp^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, i+1)) \subseteq path2set^{\mathcal{A}}(getp^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, i))
\end{array}$$

Note that $0 < l^{\mathcal{B}}$, because 0 is never removed. Then for all i within $0 < i < l^{\mathcal{B}}$:
1) Since $null^{\mathcal{B}} = null^{\mathcal{A}}$ and $m^{\mathcal{B}}(a_2) = \beta_{\mathsf{cell}}(m^{\mathcal{A}}(a_2))$, we have that

$$m^{\mathcal{B}}(a_2).arr^{\mathcal{B}}(i) \quad = \quad null^{\mathcal{B}}$$

2) If $a_2^{\mathcal{A}} \in addr2set^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, i)$, from what we have show before, we have that $addr2set^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, i) = addr2set^{\mathcal{B}}(m^{\mathcal{B}}, a_1^{\mathcal{B}}, i)$ and as $a_2^{\mathcal{A}} = a_2^{\mathcal{B}}$, we have that:

$$a_2^{\mathcal{B}} \in addr2set^{\mathcal{B}}(m^{\mathcal{B}}, a_1^{\mathcal{B}}, i)$$

3) If $path2set^{\mathcal{A}}(getp^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, i+1)) \subseteq path2set^{\mathcal{A}}(getp^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, i))$, from what we have shown before, we have that:

$$\begin{array}{rcll}
getp^{\mathcal{B}}(m^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, i+1) & = & getp^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, i+1) & \text{and} \\
getp^{\mathcal{B}}(m^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, i) & = & getp^{\mathcal{A}}(m^{\mathcal{A}}, a_1^{\mathcal{A}}, a_2^{\mathcal{A}}, i) &
\end{array}$$

Hence,

$$path2set^{\mathcal{B}}(getp^{\mathcal{B}}(m^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, i+1)) \subseteq path2set^{\mathcal{B}}(getp^{\mathcal{B}}(m^{\mathcal{B}}, a_1^{\mathcal{B}}, a_2^{\mathcal{B}}, i))$$

This concludes the proof. □

Example 8.8

Consider the formula $\psi_{\mathsf{SAT}}^{\mathrm{sanit}}$ introduced in Example 8.5:

$$\psi_{\mathsf{SAT}}^{\mathrm{sanit}} \quad : \quad i = 0 \wedge \begin{pmatrix} c = heap[head] \wedge \\ c = mkcell(e, A, l) \end{pmatrix} \wedge B = A\{i \leftarrow tail\} \wedge l_{new} = i + 1$$

Let $\mathcal{A}$ be the model $\mathcal{A}_{\psi_{\mathsf{SAT}}^{\mathrm{sanit}}}$ presented in Example 8.7, in which:

$$i^{\mathcal{A}} = 0 \qquad l_{new}^{\mathcal{A}} = 1 \qquad l^{\mathcal{A}} = 3$$

We can then construct a model $\mathcal{B}$ of $\psi_{\mathsf{SAT}}^{\mathrm{sanit}}$ by reducing one gap from $\mathcal{A}$ by stating that

$$i^{\mathcal{B}} = i^{\mathcal{A}} \qquad {l_{new}}^{\mathcal{B}} = {l_{new}}^{\mathcal{A}} \qquad l^{\mathcal{B}} = 2$$

completely ignoring arrays in model $\mathcal{A}$ at level 2. ⌟

With the previous Lemma we have shown that if a sanitized TSL formula without constants has a model, then it has a model without gaps. We now present a Lemma that allows to remove levels above any level mentioned in the formula.

Lemma 8.5 (Top-reduction):

Let $\mathcal{A}$ be a model of φ, and n a level such that $n > l^{\mathcal{A}}$ for all $l \in V_{\mathsf{level}}(\varphi)$. Let $A \in \mathcal{A}_{\mathsf{array}}$ be such that $A(n) \neq null$. Then the interpretation $\mathcal{B}$ obtained from $\mathcal{A}$ by replacing $A(n) = null$ is also a model of φ. ⌟

Proof. By a simple exhaustive case analysis on the literals of φ using Lemma 8.3. □

Thanks to the previous two Lemmas, we know that if a TSL formula φ has a model $\mathcal{A}$, then it is possible to construct a model $\mathcal{B}$ which preserves the satisfiability of φ and such that $\mathcal{B}$ has no gaps.

Corollary 8.1:

Let φ be a sanitized formula without constants. Then, φ has a model if and only if φ has a gap-less model. ⌟

We are now ready to show that the guess in STEP 2 and the split in STEP 3 preserve satisfiability. Theorem 8.1 below allows to reduce the satisfiability of φ to the satisfiability of a Presburger arithmetic formula and the satisfiability of a TSL formula without constants. We will then show how to decide this fragment of TSL.

Theorem 8.1:
A sanitized TSL formula φ is satisfiable if and only if for some order arrangement α, both $(\varphi^{PA} \wedge \alpha)$ and $(\varphi^{NC} \wedge \alpha)$ are satisfiable. ⌟

Proof. The "⇒" direction follows immediately, since a model of φ contains a model of its subformulas φ^{PA} and φ^{NC}, and a model of φ^{PA} induces a satisfying order arrangement α.

For "⇐", let α be an order arrangement for which both $(\varphi^{\text{PA}} \wedge \alpha)$ and $(\varphi^{\text{NC}} \wedge \alpha)$ are satisfiable, and let $\mathcal{N}$ be a model of $(\varphi^{\text{NC}} \wedge \alpha)$ and $\mathcal{P}$ be a model of $(\varphi^{\text{PA}} \wedge \alpha)$. By Corollary 8.1, we assume that $\mathcal{N}$ is a gap-less model. In particular, for all variables $l \in V_{\mathsf{level}}(\varphi)$, then $l^{\mathcal{N}} < \mathsf{K}$, where $\mathsf{K} = |V_{\mathsf{level}}(\varphi)|$, and for all cells $c \in \mathcal{N}_{\mathsf{cell}}$, with $c = (e, D, l)$, $l^{\mathcal{N}} < \mathsf{K}$. We construct the model $\mathcal{M}$ of φ forcing $\mathcal{M}$ to assign values to variables from $V_{\mathsf{level}}(\varphi)$ that are consistent with α. The obstacle is that the values for level variables in $\mathcal{N}$ and in $\mathcal{P}$ may be different, so the models cannot be immediately merged. We will build a model $\mathcal{M}$ of φ using $\mathcal{N}$ and $\mathcal{P}$.

In $\mathcal{M}$, all levels will receive $l^{\mathcal{M}} = l^{\mathcal{P}}$ and contents of cells will be filled using information from cells in $\mathcal{N}$. In particular, cells at level $l^{\mathcal{M}}$ are copied from the corresponding cells at level $l^{\mathcal{N}}$. The remaining issue is how to fill in cells in $\mathcal{M}$ at intermediate levels, not existing in $\mathcal{N}$. We show that these levels can be populated by cloning existing levels from $\mathcal{N}$, illustrated in Fig. 8.9(a). The two reasonable candidates to populate the necessary levels between $l_1^{\mathcal{M}}$ and $l_2^{\mathcal{M}}$, are level $l_1^{\mathcal{N}}$ and level $l_2^{\mathcal{N}}$, but without sanitization both options can lead to a predicate changing its truth value between models $\mathcal{N}$ and $\mathcal{M}$, as illustrated in Fig. 8.9(b) and Fig. 8.9(c). With sanitization, level l_{new} can be used to populate the intermediate levels, preserving the truth values of all predicates between models $\mathcal{N}$ and $\mathcal{M}$.

Let K^{PA} be the largest value assigned by $\mathcal{P}$ to any variable from $V_{\mathsf{level}}(\varphi)$. In the rest of this chapter we will use $[\mathsf{K}]$ as a short for the set $0 \ldots \mathsf{K}-1$. We start by defining the following maps:

$$\begin{array}{rcl} up : [\mathsf{K}] & \to & [\mathsf{K}^{\text{PA}}] \\ l^{\mathcal{N}} & \mapsto & l^{\mathcal{P}} \end{array} \qquad \begin{array}{rcl} \mathit{fill} : [\mathsf{K}^{\text{PA}}] & \to & [\mathsf{K}] \\ n & \mapsto & max\{k \in [\mathsf{K}] \mid up(k) \leq n\} \end{array}$$

Essentially, *fill* provides the level from $\mathcal{N}$ that will be used to fill the missing level in model $\mathcal{M}$. Some easy facts that follow from the choice of the definition of up and *fill* are that, for every variable l in $V_{\mathsf{level}}(\varphi)$, $\mathit{fill}(up(l^{\mathcal{N}})) = l^{\mathcal{N}}$.

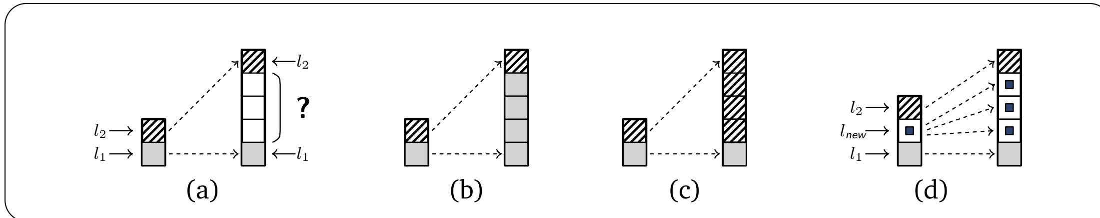


Figure 8.9: Pumping a model $\mathcal{N}$ of φ^{NC} (on the left of (a), (b), (c) and (d)) to a model $\mathcal{M}$ of φ (on the right of (a), (b), (c) and (d)) is allowed thanks to the fresh level l_{new}. In (b) the truth value of $C = D\{l_1 \leftarrow e\}$ is not preserved. In (c) $C = D\{l_2 \leftarrow e\}$ is not preserved. In (d) all predicates are preserved.

Also, every literal of the form:

$$C = D\{l \leftarrow a\} \qquad \text{satisfies that} \qquad \mathit{fill}(l+1) = \mathit{fill}(l) + 1 \tag{8.12}$$

because a sanitized formula φ contains a literal $l_{new} = l + 1$ for every literal $C = D\{l \leftarrow a\}$ that is present in φ.

We describe now how to build a model $\mathcal{M}$ of φ. The only literals missing in φ^{NC} with respect to φ are literals of the form $l = q$ for constant level q. The interpretation $\mathcal{M}$ agrees with $\mathcal{N}$ on sorts addr, elem, path, setaddr, setelem. Also, we set the domain $\mathcal{M}_{\mathsf{level}}$ to be the naturals with order, and

$$\begin{aligned} \mathcal{M}_{\mathsf{cell}} &= \mathcal{M}_{\mathsf{elem}} \times \mathcal{M}_{\mathsf{array}} \times \mathcal{M}_{\mathsf{level}} \\ \mathcal{M}_{\mathsf{mem}} &= \mathcal{M}_{\mathsf{cell}}^{\mathcal{M}_{\mathsf{addr}}} \end{aligned}$$

For level variables, we let $v^{\mathcal{M}} = v^{\mathcal{P}}$, where $v^{\mathcal{P}}$ is the interpretation of variable v in $\mathcal{P}$, the model of $(\varphi^{\text{PA}} \wedge \alpha)$. Note that $v^{\mathcal{M}} = v^{\mathcal{P}} = \mathit{up}(v^{\mathcal{N}})$. For arrays, we define $\mathcal{M}_{\mathsf{array}}$ to be the set of arrays of addresses indexed by naturals, and define the transformation $\beta_{\mathsf{array}} : \mathcal{N}_{\mathsf{array}} \to \mathcal{M}_{\mathsf{array}}$ defined as follows:

$$\beta_{\mathsf{array}}(D)(i) = D(\mathit{fill}(i))$$

Elements of sort cell $c : (e, D, l)$ are transformed into:

$$\beta_{\mathsf{cell}}(c) = (e, \beta_{\mathsf{array}}(D), \mathit{fill}(l))$$

Variables D of sort array and variables c of sort cell are interpreted as:

$$D^{\mathcal{M}} = \beta_{\mathsf{array}}(D^{\mathcal{N}}) \qquad c^{\mathcal{M}} = \beta_{\mathsf{cell}}(c^{\mathcal{N}})$$

Finally, heaps are transformed by returning the transformed cell. That is, for every $v \in V_{\mathsf{mem}}$, we let:

$$v^{\mathcal{M}}(a) = \beta_{\mathsf{cell}}(v^{\mathcal{N}})(a)$$

We only need to show that $\mathcal{M}$ is indeed a model of φ. Interestingly, all literals $l = q$ in $\mathcal{M}$ are immediately satisfied because $l^{\mathcal{M}} = l^{\mathcal{P}}$ and $q^{\mathcal{M}} = q^{\mathcal{P}}$, and the literal $(l = q)$ holds in the model $\mathcal{P}$ of $(\varphi^{\text{PA}} \wedge \alpha)$. The rest of the arithmetic literals also hold in $\mathcal{M}$ because they hold in $\mathcal{P}$ and the values of the terms involved are copied from $\mathcal{P}$ into $\mathcal{M}$: $l_1 < l_2$, $l_1 = l_2 + 1$ and $l_1 \neq l_2$. The following literals hold in $\mathcal{M}$ because they hold in $\mathcal{N}$ and their subformulas either receive the same values in $\mathcal{M}$ as in $\mathcal{N}$ or the transformations are the same on both sides:

$e_1 \neq e_2$	$a_1 \neq a_2$	
$e_1 \preceq e_2$	$a = \mathit{null}$	$c = \mathit{error}$
$c = \mathit{mkcell}(e, A, l)$	$m_2 = \mathit{upd}(m_1, a, c)$	$c = m[a]$
$l_1 < l_2$	$l = q$	$l_1 = s(l_2)$
$s = \{a\}$	$s_1 = s_2 \cup s_3$	$s_1 = s_2 \setminus s_3$
$x = \{e\}_{\mathrm{E}}$	$x_1 = x_2 \cup_{\mathrm{E}} x_3$	$x_1 = x_2 \setminus_{\mathrm{E}} x_3$

$p_1 \neq p_2$	$p = [a]$	$ordList(m, p)$
$s = path2set(p)$	$append(p_1, p_2, p_3)$	$\neg append(p_1, p_2, p_3)$

Finally, observe that $(s = addr2set(m, a, l))$ and $(p = getp(m, a_1, a_2, l))$ hold in $\mathcal{M}$ whenever they hold in $\mathcal{N}$, directly from Lemma 8.3. The remaining literals are:

- Literal $a = D[l]$. Assume $a^{\mathcal{N}} = D^{\mathcal{N}}[l^{\mathcal{N}}]$. Then, in $\mathcal{M}$, $a^{\mathcal{M}} = a^{\mathcal{N}}$ and

$$\begin{aligned} D^{\mathcal{M}}[l^{\mathcal{M}}] &= \beta_{\mathsf{array}}(D^{\mathcal{N}})[l^{\mathcal{M}}] \\ &= D^{\mathcal{N}}(\mathit{fill}(l^{\mathcal{P}})) \\ &= D^{\mathcal{N}}(\mathit{fill}(up(l^{\mathcal{N}}))) \\ &= D^{\mathcal{N}}(l^{\mathcal{N}}) \\ &= a^{\mathcal{N}} \\ &= a^{\mathcal{M}} \end{aligned}$$

- Literal $C = D\{l \leftarrow a\}$. We distinguish the two possible cases:

  1) First, let $n = l^{\mathcal{M}}$. Then,

  $$C^{\mathcal{M}}\{l^{\mathcal{M}} \leftarrow a\}(n) \;=\; C^{\mathcal{M}}\{l^{\mathcal{M}} \leftarrow a\}(l^{\mathcal{M}}) \;=\; a$$

  and

  $$D^{\mathcal{M}}(n) \;=\; D^{\mathcal{M}}(l^{\mathcal{M}}) \;=\; D^{\mathcal{N}}(\mathit{fill}(l^{\mathcal{M}})) \;=\; D^{\mathcal{N}}(\mathit{fill}(up(l^{\mathcal{N}}))) \;=\; D^{\mathcal{N}}(l^{\mathcal{N}}) \;=\; a$$

  2) The second case is $n \neq l^{\mathcal{M}}$. Then,

  $$C^{\mathcal{M}}\{l^{\mathcal{M}} \leftarrow a\})(n) \;=\; C^{\mathcal{M}}(n) \;=\; C^{\mathcal{N}}(\mathit{fill}(n))$$

  and

  $$D^{\mathcal{M}}(n) \;=\; D^{\mathcal{N}}(\mathit{fill}(n))$$

  Now, we have that $\mathit{fill}(n) \neq l^{\mathcal{N}}$. To show this we consider the two cases for $n \neq l^{\mathcal{M}}$:

  - If $n < l^{\mathcal{M}}$ then, since $\mathit{fill}(n) = max\{k \in [\mathsf{K}] | up(k) \leq n\}$ by definition, $up(l^{\mathcal{N}}) = l^{\mathcal{M}} > n$ and $\mathit{fill}(n) < l^{\mathcal{N}}$ which implies that $\mathit{fill}(n) \neq l^{\mathcal{N}}$.
  - If $n > l^{\mathcal{M}}$ then $n \geq l^{\mathcal{M}} + 1$. By (8.12) above, there is a different literal $l_{new} = l + 1$ for which $\mathit{fill}(n) \geq \mathit{fill}(l^{\mathcal{M}}_{new}) > \mathit{fill}(l^{\mathcal{M}}) = l^{\mathcal{N}}$

  Now, since in both cases $\mathit{fill}(n) \neq l^{\mathcal{N}}$, then

  $$\begin{aligned} D^{\mathcal{M}}(n) &= D^{\mathcal{N}}(\mathit{fill}(n)) \\ &= C^{\mathcal{N}}(\mathit{fill}(n)) \\ &= C^{\mathcal{N}}\{l^{\mathcal{N}} \leftarrow a\}(\mathit{fill}(n)) \\ &= C^{\mathcal{M}}\{l^{\mathcal{M}} \leftarrow a\}(n) \end{aligned}$$

Essentially, the introductions of a variable $l_{new} = l + 1$ restricts the replication of identical

levels to only the level l in $D = C\{l \leftarrow a\}$. All higher and lower levels are replicas of levels different than l (where C and D agree as in model $\mathcal{N}$).

- Literals $s = addr2set(m, a, l)$ and $p = getp(m, a_1, a_2, l)$. By induction on the length of paths, we have that for all $l^{\mathcal{N}}$:

$$(m^{\mathcal{N}}, a^{\mathcal{N}}, b^{\mathcal{N}}, l^{\mathcal{N}}, p^{\mathcal{N}}) \in reach^{\mathcal{N}} \quad \textit{iff} \quad (m^{\mathcal{M}}, a^{\mathcal{M}}, b^{\mathcal{M}}, l^{\mathcal{M}}, p^{\mathcal{M}}) \in reach^{\mathcal{M}} \tag{8.13}$$

It follows that $s^{\mathcal{N}} = addr2set(m^{\mathcal{N}}, a^{\mathcal{N}}, l^{\mathcal{N}})$ implies $s^{\mathcal{M}} = addr2set(m^{\mathcal{M}}, a^{\mathcal{M}}, l^{\mathcal{M}})$. Also $p^{\mathcal{N}} = getp(m^{\mathcal{N}}, a_1^{\mathcal{N}}, a_2^{\mathcal{N}}, l^{\mathcal{N}})$ implies that $p^{\mathcal{M}} = getp(m^{\mathcal{M}}, a_1^{\mathcal{M}}, a_2^{\mathcal{M}}, l^{\mathcal{M}})$. Essentially, since level $l^{\mathcal{M}}$ in $\mathcal{M}$ is a replica of level $l^{\mathcal{N}}$ in $\mathcal{N}$, the transitive closure of following pointers is the same paths (for $getp$) and also the same sets (for $addr2set$).

- $skiplist(m, r, l, a_1, a_2)$. We assume $skiplist(m^{\mathcal{N}}, r^{\mathcal{N}}, l^{\mathcal{N}}, a_1^{\mathcal{N}}, a_2^{\mathcal{N}})$. This implies all of the following in $\mathcal{N}$:

  i) $ordList^{\mathcal{N}}(m^{\mathcal{N}}, getp^{\mathcal{N}}(a_1^{\mathcal{N}}, a_2^{\mathcal{N}}, 0))$.
  Let p be such that $p = getp^{\mathcal{N}}(a_1^{\mathcal{N}}, a_2^{\mathcal{N}}, 0))$. As a consequence of (**??**) we have that $p = getp^{\mathcal{M}}(a_1^{\mathcal{M}}, a_2^{\mathcal{M}}, 0))$, and then:

$$ordList^{\mathcal{N}}(m^{\mathcal{N}}, getp^{\mathcal{N}}(a_1^{\mathcal{N}}, a_2^{\mathcal{N}}, 0)) \qquad \text{implies} \qquad ordList^{\mathcal{M}}(m^{\mathcal{M}}, getp^{\mathcal{M}}(a_1^{\mathcal{M}}, a_2^{\mathcal{M}}, 0))$$

  ii) $r^{\mathcal{N}} = path2set^{\mathcal{N}}(getp^{\mathcal{N}}(m^{\mathcal{N}}, a_1^{\mathcal{N}}, a_2^{\mathcal{N}}, 0))$.
  Because $getp^{\mathcal{M}}(m^{\mathcal{M}}, a_1^{\mathcal{M}}, a_2^{\mathcal{M}}, 0) = getp^{\mathcal{N}}(m^{\mathcal{N}}, a_1^{\mathcal{N}}, a_2^{\mathcal{N}}, 0)$, we also have that:

$$r^{\mathcal{M}} = path2set^{\mathcal{M}}(getp^{\mathcal{M}}(m^{\mathcal{M}}, a_1^{\mathcal{M}}, a_2^{\mathcal{M}}, 0))$$

  iii) $0 \leq l^{\mathcal{N}}$, which implies $0 \leq l^{\mathcal{M}}$.

  iv) For all $a \in r^{\mathcal{N}}$, the following holds $m^{\mathcal{N}}(a^{\mathcal{N}}).max^{\mathcal{N}} \leq l^{\mathcal{N}}$. Since $r^{\mathcal{M}} = r^{\mathcal{N}}$ and $m^{\mathcal{M}}(a) = \beta_{\mathsf{cell}}(m^{\mathcal{N}}(a))$ it is enough to consider two cases:
  1) $m^{\mathcal{N}}(a).max^{\mathcal{N}} = l^{\mathcal{N}}$, in which case $m^{\mathcal{M}}(a).max^{\mathcal{M}} = l^{\mathcal{M}}$.
  2) $m^{\mathcal{N}}(a).max^{\mathcal{N}} < l^{\mathcal{N}}$, in which case $m^{\mathcal{N}}(a).max^{\mathcal{N}} \leq l^{\mathcal{N}}$.

  v) If $(0 = l^{\mathcal{N}})$ then $(0 = l^{\mathcal{M}})$.

  vi) If $(0 < l^{\mathcal{N}})$, then $0 < l^{\mathcal{M}}$. Also, for all i from 0 to $l^{\mathcal{N}}$:

$$\begin{array}{r}
m^{\mathcal{N}}(a_2^{\mathcal{N}}).arr^{\mathcal{N}}(i) = null^{\mathcal{N}} \\
a_2^{\mathcal{N}} \in addr2set^{\mathcal{N}}(m^{\mathcal{N}}, a_1^{\mathcal{N}}, i) \\
path2set^{\mathcal{N}}(getp^{\mathcal{N}}(m^{\mathcal{N}}, a_1^{\mathcal{N}}, a_2^{\mathcal{N}}, i+1)) \subseteq path2set^{\mathcal{N}}(getp^{\mathcal{N}}(m^{\mathcal{N}}, a_1^{\mathcal{N}}, a_2^{\mathcal{N}}, i))
\end{array}$$

  Consider an arbitrary i between 0 and $l^{\mathcal{M}}$. It follows that $\mathit{fill}(i) \leq \mathit{fill}(l^{\mathcal{M}})$ so $\mathit{fill}(i) \leq l^{\mathcal{N}}$ and then:
  1) $m^{\mathcal{M}}(a_2).arr^{\mathcal{M}}(i) = m^{\mathcal{M}}(a_2).arr^{\mathcal{N}}(\mathit{fill}(i)) = null^{\mathcal{N}} = null^{\mathcal{M}}$
  2) As $a_2^{\mathcal{N}} \in addr2set^{\mathcal{N}}(m^{\mathcal{N}}, a_1^{\mathcal{N}}, i)$ and $a_2^{\mathcal{M}} = a_2^{\mathcal{N}}$, then:

$$a_2^{\mathcal{M}} = a_2^{\mathcal{N}} \in addr2set^{\mathcal{N}}(m^{\mathcal{N}}, a_1^{\mathcal{N}}, \mathit{fill}(i)) = addr2set^{\mathcal{M}}(m^{\mathcal{M}}, a_1^{\mathcal{M}}, i)$$

3) For *path2set* we have:

$$\begin{array}{ll} path2set^{\mathcal{M}}(getp^{\mathcal{M}}(m^{\mathcal{M}}, a_1^{\mathcal{M}}, a_2^{\mathcal{M}}, i+1)) & = \\ path2set^{\mathcal{N}}(getp^{\mathcal{N}}(m^{\mathcal{N}}, a_1^{\mathcal{N}}, a_2^{\mathcal{N}}, fill(i+1))) & \subseteq \\ path2set^{\mathcal{N}}(getp^{\mathcal{N}}(m^{\mathcal{N}}, a_1^{\mathcal{N}}, a_2^{\mathcal{N}}, fill(i))) & = \\ path2set^{\mathcal{M}}(getp^{\mathcal{M}}(m^{\mathcal{M}}, a_1^{\mathcal{M}}, a_2^{\mathcal{M}}, i)) & \end{array}$$

This concludes the proof. □

8.4.4 STEP 4: Presburger Constraints

At this point, the formula $(\varphi^{\text{PA}} \wedge \alpha)$ contains only literals of the form $l_1 = q$, $l_1 \neq l_2$, $l_1 = l_2 + 1$, and $l_1 < l_2$ for integer variables l_1 and l_2 and integer constant q. The satisfiability of this kind of formulas can be easily decided with off-the-shelf SMT solvers or other decision procedures for Presburger arithmetic. If $(\varphi^{\text{PA}} \wedge \alpha)$ is unsatisfiable, then the original formula (for the guessed order arrangement) is also unsatisfiable. If $(\varphi^{\text{PA}} \wedge \alpha)$ is satisfiable, then we need to check the satisfiability of $(\varphi^{\text{NC}} \wedge \alpha)$ in the next step of our decision procedure.

8.4.5 STEP 5: Deciding Satisfiability of Formulas Without Constants

In steps 1 to 4 we have reduced the satisfiability check of a given TSL formula into guessing an arrangement, checking the satisfiability of a Presburger arithmetic formula and checking the existence of a gap-less model of a sanitized formula with no constants.

We show here a reduction of the satisfiability of a sanitized formula without constants to the satisfiability of a formula in the decidable theory $\mathsf{TSL_K}$ for a sufficiently large K. That is, we detail how to generate from a sanitized formula without constants ψ (formula $(\varphi^{\text{NC}} \ \wedge \ \alpha)$ in Fig. 8.7 and Fig. 8.8) an equisatisfiable $\mathsf{TSL_K}$ formula $\ulcorner\psi\urcorner$ for a finite value K computed from the formula. The bound for K is the number of equivalence classes within the arrangement α, which is certainly lower or equal than $|V_{\mathsf{level}}(\psi)|$. This bound limits the number of levels required in the reasoning.

Example 8.9
For the formula $\psi_{\mathsf{SAT}}^{\text{sanit}}$ defined in Example 8.5 we have that $V_{\mathsf{level}}(\psi_{\mathsf{SAT}}^{\text{sanit}}) = \{i, l, l_{new}\}$ and thus we need to construct a formula in, at most, TSL_3. For example, if for such formula we would have guessed the arrangement $\{i < l_{new}, i < l, l_{new} = l\}$, we would require to construct a formula in TSL_2. On the other hand, for formula $\psi_{\mathsf{UNSAT}}^{\text{sanit}}$ we have that $V_{\mathsf{level}}(\psi_{\mathsf{UNSAT}}^{\text{sanit}}) = \{i, l, l_{new}, j_1, j_2\}$ and hence we need to construct a formula in, at most, TSL_5. ⌟

The translation from ψ into $\ulcorner\psi\urcorner$ works as follows. For every variable D of sort array appearing in some literal in ψ we introduce K fresh new variables $v_{D[0]}, \ldots, v_{D[\mathsf{K}-1]}$ of sort addr. These variables correspond to the addresses from D that the decision procedure for $\mathsf{TSL_K}$ needs to reason about. All literals from ψ are left unchanged in $\ulcorner\psi\urcorner$ except $(c = mkcell(e, D, l))$, $(a = D[l])$, $(C = D\{l \leftarrow a\})$ and $skiplist(m, s, l, a_1, a_2)$ that are changed as follows:

- Literal $c = mkcell(e, D, l)$ is transformed into:

$$c \quad = \quad (e, v_{D[0]}, \ldots, v_{D[K-1]})$$

- Literal $a = D[l]$ gets translated into:

$$\bigwedge_{i \in [\mathsf{K}]} l = i \rightarrow a = v_{D[i]}$$

- Literal $C = D\{l \leftarrow a\}$ is translated into:

$$\Big(\bigwedge_{i \in [\mathsf{K}]} l = i \rightarrow a = v_{C[i]}\Big) \quad \wedge \quad \Big(\bigwedge_{j \in [\mathsf{K}]} l \neq j \rightarrow v_{C[j]} = v_{D[j]}\Big) \tag{8.14}$$

- Literal $skiplist(m, r, l, a_1, a_2)$ gets translated into:

$$\left[\begin{array}{lll} ordList(m, getp(m, a_1, a_2, 0)) & \wedge \quad r = path2set(getp(m, a_1, a_2, 0)) & \wedge \\ \bigwedge_{i \in [\mathsf{K}]} m[a_2].arr[i] = null & & \wedge \\ \bigwedge_{i \in [\mathsf{K}]} a_2 \in addr2set(m, a_1, i) & & \wedge \\ \bigwedge_{i \in [\mathsf{K}-1]} path2set(getp(m, a_1, a_2, i+1)) \subseteq path2set(getp(m, a_1, a_2, i)) & & \end{array}\right] \tag{8.15}$$

Note that a conjunct of the form $\bigwedge_{i \in [\mathsf{K}]}$ is simply a notation for a finite collection of conjuncts for the previously fixed value K. For example, for $\mathsf{K} = 3$,

$$\bigwedge_{i \in [\mathsf{K}]} l = i \rightarrow a = v_{A[i]}$$

is simply

$$\big(l = 0 \rightarrow a = v_{A[0]}\big) \;\wedge\; \big(l = 1 \rightarrow a = v_{A[1]}\big) \;\wedge\; \big(l = 2 \rightarrow a = v_{A[2]}\big)$$

The formula $\ulcorner \varphi \urcorner$ obtained using these translations belongs to the theory TSL_K.

Example 8.10

For instance, if we consider the formula $\psi^{\mathrm{NC}}_{\mathsf{SAT}}$ presented in Example 8.6, we have:

$$\ulcorner \psi^{\mathrm{NC}}_{\mathsf{SAT}} \urcorner : \left[\begin{array}{llllll} i = 0 \rightarrow tail = v_{B[0]} & \wedge & i = 1 \rightarrow tail = v_{B[1]} & \wedge & i = 2 \rightarrow tail = v_{B[2]} & \wedge \\ i \neq 0 \rightarrow v_{B[0]} = v_{A[0]} & \wedge & i \neq 1 \rightarrow v_{B[1]} = v_{A[1]} & \wedge & i \neq 2 \rightarrow v_{B[2]} = v_{A[2]} & \wedge \\ c = heap[head] & \wedge & c = mkcell(e, v_{A[0]}, v_{A[1]}, v_{A[2]}) \wedge l_{new} = i + 1 & & & \end{array}\right]$$

⌟

The following Lemma establishes the correctness of the translation.

Lemma 8.6:
Let ψ be a sanitized TSL formula with no constants. Then, ψ is satisfiable if and only if $\ulcorner\psi\urcorner$ is also satisfiable. ⌟

Proof. Directly from Lemmas 8.7 and 8.8 below. □

Lemma 8.7:
Let φ be a normalized set of TSL literals with no constants. Then, if φ is satisfiable then $\ulcorner\varphi\urcorner$ is also satisfiable. ⌟

Proof. Assume φ is satisfiable. By Corollary 8.1, φ has a gap-less model $\mathcal{A}$. This model $\mathcal{A}$ satisfies that for every natural i from 0 to $\mathsf{K}-1$ there is a level $l \in V_{\mathsf{level}}(\varphi)$ with $l^{\mathcal{A}} = i$.

Building a Model $\mathcal{B}$. We now construct a model $\mathcal{B}$ of $\ulcorner\varphi\urcorner$. For the domains:

$$\mathcal{B}_{\mathsf{addr}} = \mathcal{A}_{\mathsf{addr}} \quad \mathcal{B}_{\mathsf{elem}} = \mathcal{A}_{\mathsf{elem}} \quad \mathcal{B}_{\mathsf{path}} = \mathcal{A}_{\mathsf{path}} \quad \mathcal{B}_{\mathsf{setaddr}} = \mathcal{A}_{\mathsf{setaddr}} \quad \mathcal{B}_{\mathsf{setelem}} = \mathcal{A}_{\mathsf{setelem}}$$

and:

$$\mathcal{B}_{\mathsf{level}} = [\mathsf{K}] \qquad \mathcal{B}_{\mathsf{cell}} = \mathcal{B}_{\mathsf{elem}} \times \mathcal{B}_{\mathsf{addr}}^{\mathsf{K}} \qquad \mathcal{B}_{\mathsf{mem}} = \mathcal{B}_{\mathsf{cell}}^{\mathcal{B}_{\mathsf{addr}}}$$

For the variables, we let $v^{\mathcal{B}} = v^{\mathcal{A}}$ for sorts addr, elem, path, setaddr and setelem. For level, we assign $l^{\mathcal{B}} = l^{\mathcal{A}}$, which is guaranteed to be within 0 and $\mathsf{K}-1$. For cell, let $c = (e, D, l)$ be an element of $\mathcal{A}_{\mathsf{cell}}$. The following function δ_{cell} maps c into an element of $\mathcal{B}_{\mathsf{cell}}$:

$$\delta_{\mathsf{cell}}(e, D, l) \quad = \quad (e, D(0), \ldots, D(\mathsf{K}-1))$$

Essentially, cells in $\mathcal{B}$ only record information of relevant levels from the corresponding cells in $\mathcal{A}$, which are those levels for which there is a level variable. All upper levels are ignored. Every variable v of sort cell is interpreted as $v^{\mathcal{B}} = \delta_{\mathsf{cell}}(v^{\mathcal{A}})$. A variable v of sort mem is interpreted as a function that maps an element a of $\mathcal{B}_{\mathsf{addr}}$ into $\delta_{\mathsf{cell}}(v^{\mathcal{A}}(a))$, essentially mapping addresses to the corresponding transformed cells. Finally, for all array variables D in the formula φ, we assign $v^{\mathcal{B}}_{D[i]} = D^{\mathcal{A}}(i)$.

Checking the Model $\mathcal{B}$. We are ready to show, by case analysis on the literals of the original formula φ, that $\mathcal{B}$ is indeed a model of $\ulcorner\varphi\urcorner$. The following literals hold in $\mathcal{B}$, directly from the choice of assignments in $\mathcal{B}$ because the corresponding literals hold in $\mathcal{A}$:

$e_1 \neq e_2$	$a_1 \neq a_2$	$l_1 \neq l_2$
$e_1 \preceq e_2$	$a = null$	$c = error$
	$m_2 = upd(m_1, a, c)$	$c = m[a]$
$l_1 < l_2$	$l = q$	
$s = \{a\}$	$s_1 = s_2 \cup s_3$	$s_1 = s_2 \setminus s_3$
$x = \{e\}_{\mathrm{E}}$	$x_1 = x_2 \cup_{\mathrm{E}} x_3$	$x_1 = x_2 \setminus_{\mathrm{E}} x_3$
$p_1 \neq p_2$	$p = [a]$	$ordList(m, p)$
$s = path2set(p)$	$append(p_1, p_2, p_3)$	$\neg append(p_1, p_2, p_3)$

The remaining literals are:

- Literal $c = mkcell(e, D, l)$. Clearly the data field of $c^{\mathcal{B}}$ and the translation of $mkcell^{\mathcal{B}}(e, \ldots)$ coincide. Similarly, by the δ_{cell} map for elements of $\mathcal{B}_{\mathsf{cell}}$, the array entries coincide with the values of the variables $v_{D[i]}$. Hence, $c = mkcell(e, v_{D[0]}, \ldots, v_{D[K-1]})$ holds in $\mathcal{B}$.
- Literal $a = D[l]$. Our choice of $v^{\mathcal{B}}_{D[i]}$ makes, for $i = l^{\mathcal{B}}$:

$$v^{\mathcal{B}}_{D[i]} \quad = \quad D^{\mathcal{A}}(l^{\mathcal{B}}) \quad = \quad D^{\mathcal{A}}(l^{\mathcal{A}}) \quad = \quad a^{\mathcal{A}} \quad = \quad a^{\mathcal{B}}$$

  so the clause generated from $a = D[l]$ in $\ulcorner\varphi\urcorner$ holds in $\mathcal{B}$.
- Literal $C = D\{l \leftarrow a\}$. In this case, for $j = l^{\mathcal{A}} = l^{\mathcal{B}}$, $v^{\mathcal{B}}_{C[j]} = C^{\mathcal{A}}(l^{\mathcal{A}}) = a^{\mathcal{A}} = a^{\mathcal{B}}$. Moreover, for all other indices i:

$$v^{\mathcal{B}}_{C[i]} \quad = \quad C^{\mathcal{A}}(i) \quad = \quad D^{\mathcal{A}}(i) \quad = \quad v^{\mathcal{B}}_{D[i]}$$

  so the clause (8.14) generated from $C = D\{l \leftarrow a\}$ in $\ulcorner\varphi\urcorner$ holds in $\mathcal{B}$.
- Literal $s = addr2set(m, a, l)$. By induction on the length of paths for all $l^{\mathcal{A}}$:

$$(m^{\mathcal{A}}, a^{\mathcal{A}}, b^{\mathcal{A}}, l^{\mathcal{A}}, p^{\mathcal{A}}) \in reach^{\mathcal{A}} \quad \textit{iff} \quad (m^{\mathcal{B}}, a^{\mathcal{B}}, b^{\mathcal{B}}, l^{\mathcal{B}}, p^{\mathcal{B}}) \in reach^{\mathcal{B}} \tag{8.16}$$

  It follows that:

$$s^{\mathcal{A}} \quad = \quad addr2set(m^{\mathcal{A}}, a^{\mathcal{A}}, l^{\mathcal{A}}) \qquad \text{implies} \qquad s^{\mathcal{B}} = addr2set(m^{\mathcal{B}}, a^{\mathcal{B}}, l^{\mathcal{B}})$$

- Literal $p = getp(m, a_1, a_2, l)$. Fact (8.16) also implies immediately that if literal $p = getp(m, a_1, a_2, l)$ holds in $\mathcal{A}$ then $p = getp(m, a_1, a_2, l)$ holds in $\mathcal{B}$.
- Literal $skiplist(m, s, l, a_1, a_2)$. Following (8.15), the five disjuncts (1) the lowest level is ordered, (2) the region contains exactly all low level, (3) the last sentinel cell has null successors, (4) the last sentinel cell is reachable from the initial cell at all levels, and (5) each level is a subset of the lower level, hold in $\mathcal{B}$, because they corresponding disjunct holds in $\mathcal{A}$.

This shows that $\mathcal{B}$ is a model of $\ulcorner\varphi\urcorner$ and therefore $\ulcorner\varphi\urcorner$ is satisfiable. □

Lemma 8.8:
Let φ be a normalized set of TSL literals with no constants. If $\ulcorner\varphi\urcorner$ is satisfiable, then φ is also satisfiable. ⌟

Proof. We start from a $\mathsf{TSL_K}$ model $\mathcal{B}$ of $\ulcorner\varphi\urcorner$ and we construct a model $\mathcal{A}$ of φ.

Building a Model $\mathcal{A}$. We now proceed to show that φ is satisfiable by building a model $\mathcal{A}$. For the domains, we let:

$$\mathcal{A}_{\mathsf{addr}} = \mathcal{B}_{\mathsf{addr}} \qquad \mathcal{A}_{\mathsf{elem}} = \mathcal{B}_{\mathsf{elem}} \qquad \mathcal{A}_{\mathsf{path}} = \mathcal{B}_{\mathsf{path}} \qquad \mathcal{A}_{\mathsf{setaddr}} = \mathcal{B}_{\mathsf{setaddr}} \qquad \mathcal{A}_{\mathsf{setaddr}} = \mathcal{B}_{\mathsf{setelem}}$$

Also, $\mathcal{A}_{\mathsf{level}}$ is the naturals with order, and

$$\mathcal{A}_{\mathsf{cell}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{array}} \times \mathcal{A}_{\mathsf{level}} \qquad\qquad \mathcal{A}_{\mathsf{mem}} = \mathcal{A}_{\mathsf{cell}}^{\mathcal{A}_{\mathsf{addr}}}$$

For the variables, we let $v^{\mathcal{A}} = v^{\mathcal{B}}$ for sorts addr, elem, path, setaddr and setelem. For level, we also assign $l^{\mathcal{A}} = l^{\mathcal{B}}$. For cell, let $c = (e, a_0, \ldots, a_{\mathsf{K}-1})$ be an element of $\mathcal{B}_{\mathsf{cell}}$. Then the following function γ_{cell} maps $c : (e, a_0, \ldots, a_{K-1})$ into an element of $\mathcal{A}_{\mathsf{cell}}$:

$$\gamma_{\mathsf{cell}}((e, a_0, \ldots, a_{\mathsf{K}-1})) \quad = \quad (e, D, l) \tag{8.17}$$

where:

$$\begin{aligned} l &= \mathsf{K} \\ D(i) &= \begin{cases} a_i & \text{if } 0 \leq i < l \\ null & \text{if } i \geq l \end{cases} \end{aligned}$$

Every variable v of sort cell is interpreted as $v^{\mathcal{A}} = \gamma_{\mathsf{cell}}(v^{\mathcal{B}})$. Finally, a variable v of sort mem is interpreted as a function that maps an element a of $\mathcal{A}_{\mathsf{addr}}$ into $\gamma_{\mathsf{cell}}(v^{\mathcal{B}}(a))$, mapping addresses to transformed cells.

Finally, for all arrays variables D in the original formula φ, we assign:

$$D^{\mathcal{A}}(i) = \begin{cases} v^{\mathcal{B}}_{D[i]} & \text{if } i < \mathsf{K} \\ null & \text{otherwise} \end{cases} \tag{8.18}$$

Checking the Model $\mathcal{A}$. We are ready to show, by cases on the literals of the original formula φ that $\mathcal{A}$ is indeed a model of φ. The following literals hold in $\mathcal{A}$ because the corresponding literals hold in $\mathcal{B}$:

$e_1 \neq e_2$	$a_1 \neq a_2$	$l_1 \neq l_2$
$e_1 \preceq e_2$	$a = null$	$c = error$
	$m_2 = upd(m_1, a, c)$	$c = m[a]$
$l_1 < l_2$	$l = q$	
$s = \{a\}$	$s_1 = s_2 \cup s_3$	$s_1 = s_2 \setminus s_3$
$x = \{e\}_{\mathrm{E}}$	$x_1 = x_2 \cup_{\mathrm{E}} x_3$	$x_1 = x_2 \setminus_{\mathrm{E}} x_3$
$p_1 \neq p_2$	$p = [a]$	$ordList(m, p)$
$s = path2set(p)$	$append(p_1, p_2, p_3)$	$\neg append(p_1, p_2, p_3)$

The remaining literals are:

- Literal $c = mkcell(e, D, l)$. Clearly the data field of $c^{\mathcal{A}}$ and the translation of $mkcell^{\mathcal{A}}(e, \ldots)$ given by (8.17) coincide. By the choice of array variables $D^{\mathcal{A}}(i) = v^{\mathcal{B}}_{D[i]} = a_i$, so A and the array part of c coincide at all positions. For $l^{\mathcal{A}}$ we choose K for all cells.
- Literal $a = D[l]$. It holds since

$$a^{\mathcal{A}} \quad = \quad a^{\mathcal{B}} \quad = \quad v^{\mathcal{B}}_{D[l^{\mathcal{B}}]} \quad = \quad D^{\mathcal{A}}(l^{\mathcal{B}}) \quad = \quad D^{\mathcal{A}}(l^{\mathcal{A}})$$

- Literal $B = D\{l \leftarrow a\}$. We have that the translation of $C = D\{l \leftarrow a\}$ for $\ulcorner\varphi\urcorner$ given by (8.14) holds in $\mathcal{B}$. Consider an arbitrary level $m < \mathsf{K}$. If $m = l^{\mathcal{B}} = l^{\mathcal{A}}$ then $a = v_{C[m]} = C^{\mathcal{A}}(m)$. If $m \neq l^{\mathcal{B}}$ then $v_{C[m]} = v_{D[m]}$ and hence $C^{\mathcal{A}}(m) = v_{C[m]} = v_{D[m]} = D^{\mathcal{A}}(m)$.
- Literal $A = B$. The clause (8.18) generated from $A = B$ in $\ulcorner\varphi\urcorner$ holds in $\mathcal{B}$, by assumption. For an arbitrary j from [K]:

$$A^{\mathcal{A}}(j) \quad = \quad v^{\mathcal{B}}_{A[j]} \quad = \quad v^{\mathcal{B}}_{B[j]} \quad = \quad B^{\mathcal{A}}(j)$$

Moreover, for $j \geq \mathsf{K}$, then $A^{\mathcal{A}}(j) = null = B^{\mathcal{A}}(j)$ and consequently $A^{\mathcal{A}} = B^{\mathcal{A}}$ as desired.

- Literal $s = addr2set(m, a, l)$. By induction on the length of paths, for all $l^{\mathcal{A}}$:

$$(m^{\mathcal{A}}, a^{\mathcal{A}}, b^{\mathcal{A}}, l^{\mathcal{A}}, p^{\mathcal{A}}) \in reach^{\mathcal{A}} \quad \textit{iff} \quad (m^{\mathcal{B}}, a^{\mathcal{B}}, b^{\mathcal{B}}, l^{\mathcal{B}}, p^{\mathcal{B}}) \in reach^{\mathcal{B}} \tag{8.19}$$

  It follows that $s^{\mathcal{A}} = addr2set(m^{\mathcal{A}}, a^{\mathcal{A}}, l^{\mathcal{A}})$ implies $s^{\mathcal{A}} = addr2set(m^{\mathcal{A}}, a^{\mathcal{A}}, l^{\mathcal{A}})$.

- Literal $p = getp(m, a_1, a_2, l)$. Fact (8.19) also implies immediately that if literal $p = getp(m, a_1, a_2, l)$ holds in $\mathcal{A}$ then $p = getp(m, a_1, a_2, f(l))$ holds in $\mathcal{B}$.

- Literal $skiplist(m, s, l, a_1, a_2)$. Following (8.15) and the way arrays are translated from a model $\mathcal{A}$ into a model $\mathcal{B}$, we have that the five disjuncts (1) the lowest level is ordered, (2) the region contains exactly all low addresses in the lowest level, (3) the last sentinel cell has null successors, (4) the last sentinel cell is reachable from the initial sentinel cell at all levels, and (5) each level is a subset of the lower level, hold in $\mathcal{A}$, because their corresponding disjunct holds in $\mathcal{B}$.

This shows that $\mathcal{A}$ is a model of φ and therefore φ is satisfiable. □

The main result of this section is the following decidability theorem, which follows from Lemma 8.6, Theorem 8.1, the fact that every formula can be normalized and sanitized, and the decidability of $\mathsf{TSL_K}$ formulas.

Theorem 8.2:
The satisfiability problem of quantifier-free TSL-formulas is decidable. ⌟

8.5 Summary

In this chapter we have presented TSL, the *Theory of Skiplists with Unbounded Levels*, a theory which is capable of reasoning about skiplists with arbitrarily many levels. TSL is powerful enough to reason about memory, cells, pointers, regions and reachability, ordered single-linked lists and sublists, allowing the description of the skiplist property, and the representation of memory modifications introduced by the execution of program statements. The main novelty of TSL, contrary to $\mathsf{TSL_K}$ presented in Chapter 7, is that it is not limited to skiplists with a fixed number of levels.

In this chapter we showed that the quantifier-free TSL fragment is decidable by reducing its satisfiability problem to $\mathsf{TSL_K}$. The complexity of deciding TSL is NP-complete, as one can guess a model of polynomial size. Also, as part of the proposed decision procedure for TSL, we described how to reduce a formula in TSL to a equisatisfiable formula in $\mathsf{TSL_K}$. Our reduction illustrates that a decision procedure for TSL only needs to reason about those levels explicitly mentioned in the (sanitized) formula.

The decision procedure presented in this chapter has been implemented as part of LEAP, a prototype of theorem prover which is presented in Chapter 9. Later, in Chapter 10 we will show that theory TSL and its decision procedure are useful for automatically proving the verification conditions generated during the practical verification of skiplist implementations, including the skiplist implementation presented in Section 8.1 in this chapter and a skiplist implementation [123] which is part of the KDE library [197].

Part III

Implementation and Experimental Results

9

LEAP: *A Verification Tool for Parametrized Datatypes*

> *That's one small step for a man,*
> *one giant leap for mankind*
>
> Neil Armstrong

In this chapter we present LEAP, a tool for the verification of parametrized systems and concurrent data types that store both finite and infinite data. LEAP implements the parametrized invariance techniques presented in Chapter 3 and the parametrized verification diagrams introduced in Chapter 4. Additionally, LEAP implements some decision procedures, including the decision procedures for lists and skiplists described in Part II of this work. All decision procedures implemented by LEAP are constructed on top of state-of-the-art SMT solvers, such as Z3 [63], Yices [69] and CVC4 [15, 17].

The rest of this chapter is structured as follows. Section 9.1 presents a general description of LEAP. Section 9.2 gives a brief introduction to the main features offered by LEAP. Section 9.3 describes how to use LEAP in order to verify some of the concurrent data types presented in this work. Finally, Section 9.4 gives a summary of what is presented in this chapter.

9.1 General Overview

LEAP is a prototype theorem prover, currently under development at the IMDEA Software Institute, which implements the ideas discussed in this work. LEAP is implemented in Ocaml [198] and consists of about 70,000 lines of code. The source code is currently not accessible, but its preliminary version and a set of application examples can be downloaded from its web site: `http://software.imdea.org/leap`. Alternatively, there exists an online version of LEAP accessible at: `http://ares.software.imdea.org/leap`.

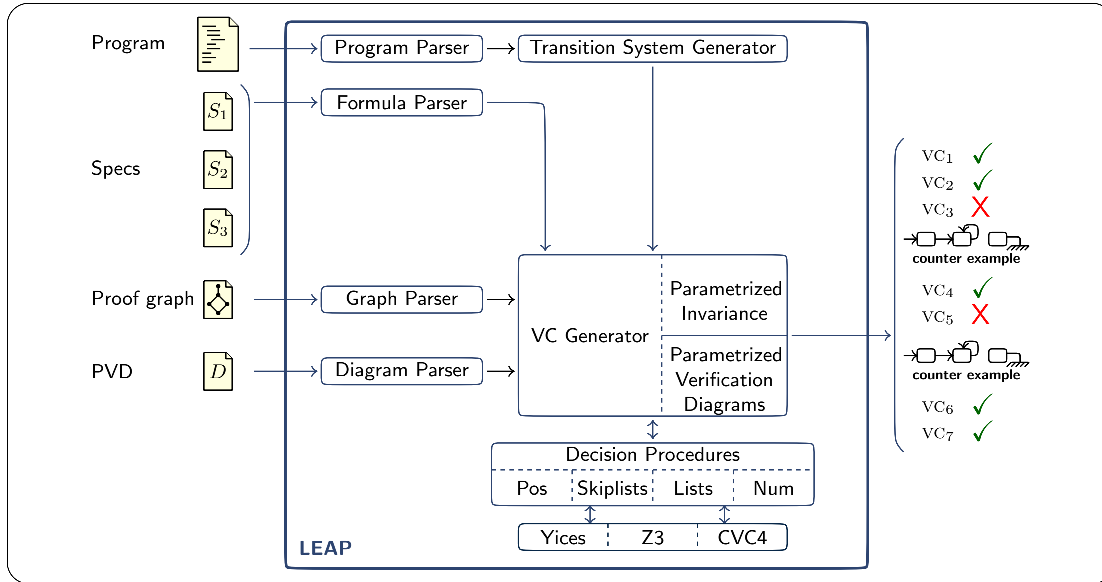


Figure 9.1: Schematic design of LEAP.

LEAP implements:

- The parametrized invariance proof rules for safety properties presented in Chapter 3.
- The parametrized verification diagrams, presented in Chapter 4, which enable the verification of liveness properties.
- The decision procedure for TL3, presented in Chapter 6, which tackles the satisfiability of data structures in the heap with the shape of single-linked lists.
- The decision procedure for TSL_K, described in Chapter 7, which enables checking the satisfiability of formulas describing concurrent skiplists with a bounded number of levels.
- The decision procedure for TSL, presented in Chapter 8, which tackles the satisfiability problem of skiplists with an unbounded number of levels.

As we mentioned before, we follow a deductive approach based on the ideas of Manna-Pnueli [144]. Because of that, we are willing to sacrifice full automation in exchange of being able to deal with programs that manipulate complex data structures. Our target with LEAP is wide applicability, while improving automation is an important secondary goal.

Fig. 9.1 shows a schematic representation of the structure of LEAP. LEAP receives as input a program which is assumed to be executed concurrently by an unbounded number of threads. The input program needs to be written in LEAP's own programming language. The language accepted by LEAP is very close to SPL, which is presented in Section 2.1, and includes conditionals, loops, atomic sections, pointer manipulation, ghost code notation and line labeling among other features. Appendix B.2.1 contains a detailed description of the syntax for programs accepted by LEAP.

In addition to the input program description, LEAP can receive further input files depending on the kind of verification to be carried out:

- For **safety** properties, LEAP requires to receive as input:
    1. A folder containing the set of files which describe the specifications of the invariant candidates. Appendix B.2.2 contains a full description of the syntax that these specifications need to follow.
    2. A file describing the proof graph. The idea of a proof graph was introduced in Section 3.2.4. Basically, a proof graph describes the relations and dependencies between the invariant candidates that take part in the verification process. Additionally, a proof graph contains extra information such as specialized tactics and heuristics to follow in order to ease the verification process. Appendix B.2.3 contains a more detailed explanation of the syntax accepted by LEAP for describing proof graphs.
- For **liveness** properties, LEAP needs to receive as input:
    1. A description of the parametrized verification diagram. The idea of parametrized verification diagrams was presented in Section 4.2. Basically, a parametrized verification diagram describes the proof that a parametrized system executing the input program satisfies a given temporal liveness property. Appendix B.2.4 presents a detailed description of the syntax that needs to be followed by a file describing a temporal parametrized verification diagram.
    2. In general, a parametrized verification diagram will require some assumptions in order to prove a temporal liveness property. Assumptions are system invariants described in specification files that follow the same syntax as for invariant candidates. The folder containing these assumptions can also be passed to LEAP as an input argument.
    3. Usually, the verification process results easier with the assistance of tactics and heuristics. This information can be provided on a separate input file. This file can optionally describe which auxiliary support invariant is required in particular cases. Appendix B.2.3 contains the syntax accepted by LEAP for this auxiliary input file.

Example 9.1
Fig. 9.2 presents an example of a program written in the programming language accepted by LEAP. The program contains part of the algorithm for lock-coupling single linked lists introduced in Section 6.1.1. The program in Fig. 9.2 includes two procedure. Procedure `main` corresponds to the most general client while procedure `insert` corresponds to procedure INSERT. For simplicity, we omit from this example the procedures SEARCH and REMOVE. ⌟

Example 9.1 makes evident some aspects of the input program. As can be seen, the declaration of global variables begins with the **global** keyword. The `ghost` modifier is used to declare ghost variables. To specify some initial assumptions we can use the **assume** keyword just before the procedure declaration. The initial assumptions are expressed as a LEAP formula. In the example above the initial assumptions state, among other things, that initially `region` contains only `head`, `tail` and `null`.

Every LEAP program must contain a `main` procedure which serves a special purpose, as it is the first procedure to be called when a program execution begins. Therefore, in the program described in Fig. 9.2 we use `main` as the most general client that any thread executing will run.

```
    global
      addr head
      addr tail
      ghost addrSet region

    assume
      region = union(union({head}, {tail}), {null})
      rd(heap, head).data = lowestElem /\
      rd(heap, tail).data = highestElem /\
      head != tail /\
      head != null /\
      tail != null /\
      head->next = tail /\
      tail->next = null

    procedure main()
      elem e
    begin
    :main_body[
1:    while (true) do
2:      e := havocListElem();
3:      choice
4:        call search(e);
5:      _or_
6:        call insert(e);
7:      _or_
8:        call remove(e);
9:      endchoice
10:   endwhile
11:   return();
    :main_body]
    end

    procedure insert(e:elem)
      addr prev
      addr curr
      addr aux
    begin
12:   prev := head;
13:   prev->lock;
14:   curr := prev->next;
15:   curr->lock;
16:   while (curr->data < e) do
17:     aux := prev;
18:     prev := curr;
19:     aux->unlock;
20:     curr := curr->next;
21:     curr->lock;
22:   endwhile
23:   if (curr != null /\ curr->data > e) then
24:     aux := malloc(e,null,#);
25:     aux->next := curr;
    :connect
26:     prev->next := aux
        $region := union(region, {aux});$
27:   endif
28:   prev->unlock;
29:   curr->unlock;
30:   return();
    end
```

Figure 9.2: Example of a LEAP input program.

Once the program input file is parsed, LEAP internally constructs a representation of a parametrized transition system in which all threads execute the same input program. Additionally, the auxiliary input files which contain the invariant candidates, the proof graphs and the parametrized verification diagrams are parsed and passed to a verification condition generator. The verification condition generator internally implements the proof rules and formalisms described in Chapter 3 and Chapter 4. As output, the verification condition generator produces a collection of verification conditions whose validity needs to be checked.

Each verification condition corresponds to a small step in the execution. All verification conditions generated by LEAP are quantifier free as long as the specification is quantifier free. So far, LEAP does not support the use of quantifiers. The generated verification conditions are then discharged to specialized decision procedures which automatically decide their validity. In order to check the validity of a given verification condition, LEAP transforms the verification condition into its negation and checks its satisfiability using an appropriate decision procedure. If the decision procedures conclude that all verification conditions are valid, then LEAP indicates that the parametrized program satisfies the analyzed safety or liveness property. If a verification condition is not valid, then the decision procedure generates a counter-model which corresponds to an offending small step of the system that leads to a violation of the safety or liveness property. The programmer can use this counter-model to either identify a bug or instrument the program with intermediate invariants.

Currently, LEAP implements 6 decision procedures for different data types. Each of these decision procedures are implemented on top of SMT solvers such as Yices, Z3 and CVC4. The implemented decision procedures include:

- A simple decision procedure capable of reasoning about **program locations**. In general, many of the generated verification conditions can be checked by simply reasoning about program locations, abstracting the rest of literals that may deal with complex data manipulation. Because of that, LEAP includes a simple decision procedure that reasons about program locations only. Usually, when a verification condition needs to be checked, LEAP first tries with this simple decision procedure which is fast and in many cases sufficient to determine the validity of a given verification condition.

- A decision procedure for **Presburger arithmetic with finite sets of integers**, where sets support operations such as union and intersection. This decision procedure is useful for programs that manipulate integers and set of integers, such as the mutual exclusion protocols presented in Example 2.2.

- A decision procedure for **Presburger arithmetic with finite sets and pairs of integers and thread identifiers** which is useful for the cases in which a numeric program needs to reason about pairs of integers and thread identifiers.

- A decision procedure for **concurrent single-linked lists**, inspired by the decision procedure for TL3 presented in Chapter 6.

- A decision procedure for **concurrent skiplists of unbounded width and bounded height**, which corresponds to the decision procedure for the TSL_K family presented in Chapter 7.

- A decision procedure for **skiplists of unbounded width and height**, motivated by the decision procedure for TSL described in Chapter 8.

Using some of LEAP's command line options, it is possible to instruct LEAP on which decision procedure to use. Appendix B.1 provides a more detailed explanation of all available LEAP command line options.

9.2 Features

LEAP implements some features with the purpose of easing the verification process. These features include notation and ghost code which can be added to an input program as well as tactics and heuristics to improve the verification process.

9.2.1 Programs

LEAP input programs are very similar to SPL, presented in Section 2.1, extended with some extra features. One of these features is program labeling. LEAP allows to assign a label to a program line, so that later it can be referred in a specification using this label. There are two kind of program labels:

1. A simple program label which refers a single program line. This kind of labels are declared using the "`:`" symbol followed by the label name.

2. A more complex program label which refers to a set of consecutive program lines. To do this, we need to insert the same label identifier at two different position within the program. All lines between both occurrences of the label identifier can then be referred using this identifier. The first occurrence of the label identifier must begin with the "`:`" symbol followed by the name of the label identifier and end with the "`[`" symbol. The second occurence of the label identifier must begin with the "`:`" symbol followed by the name of the label identifier, but in this case must end with the "`]`" symbol.

Example 9.2
Consider once again the program described in Fig. 9.2. In this program, label `connect` is a single line label which refers program line 26. This means that we can later use "`@connect(i).`" in a specification as a synonym for program line 26 for thread `i`.

On the other hand, label `main_body` is a multi-line label which denotes the program lines between 1 and 11. This means that we can use "`@main_body(i).`" in a specification as a synonym for the set of program lines that goes from line 1 to line 11 when run by thread `i`. ⌟

LEAP input programs can also contain ghost code. As we mentioned above, ghost variables are declared using the `ghost` modifier. Ghost code is added between dollar symbols "`$`" and can contain sequences of assignments and conditional statements.

Another feature of LEAP input programs is atomic section. Atomic sections describe a set of program instructions that are executed as a single program statement. These atomic sections are written between braces: "`{`" and "`}`".

```
{
    ticket := avail;
    avail := avail + 1;
    bag := iunion (bag, isingle(avail));
}
```

Figure 9.3: Example of LEAP atomic statement.

Example 9.3
Consider once again the mutual exclusion protocol presented in Example 2.2. In this example, line 3 consists of a statement which, atomically:

- assigns to $ticket$ the value of the integer variable $avail$;
- increases the value of $avail$ by one; and
- adds to the set bag the value of $ticket$.

Fig. 9.3 shows how we can express all these changes as a single LEAP atomic statement. In this statement, `isingle` is the constructor for singleton sets of integers which contain a single integer value and `iunion` is the union operator for set of integers. ⌟

Other feature offered by LEAP is the possibility to add some initial assumptions about the program variables. This feature results very useful in cases in which the user needs to assume some initial conditions, like for instance the fact that the memory contains a data structure of the shape of a list or skiplist, before the program is executed.

9.2.2 Specifications

Specifications can be used to define invariant candidates, when performing the verification of a safety property, or for describing supporting invariants, when performing the verification a liveness property.

A requirement imposed by LEAP is that all specifications must be written in files with the "`.inv`" extension. Each of these specification files contains two sections. The first section starts with the `vars` keyword and it is used for declaring all the variables that parametrize the specification. In general, specification parameters are some variables of sort thread identifier, however, it is possible to use variables of other sorts as parameters of a specification. The second section starts with the `invariant` keyword followed by an identifier between brackets. This identifier corresponds to the name that identifies the invariant candidate and which can later be used to referred the specification in a proof graph or a parametrized verification diagram. Following the specification name, it comes the formula which describes the invariant candidate itself.

Example 9.4
Consider again the mutual exclusion protocol presented in Example 2.2. Fig. 9.4 contains the LEAP specification which states the mutual exclusion property. This property states that 2 different

```
vars:
  tid i
  tid j

invariant [mutex] :
  (i != j) -> ~ (@crit(i). /\ @crit(j).)
```

Figure 9.4: Example of a LEAP specification.

threads cannot be in the critical section at the same time. In Fig. 9.4 we use the label `crit` to denote program lines 5 and 6, which correspond to the critical section of the program.

In the figure above, we declare an invariant candidate named `mutex` which is parametrized by two thread identifiers `i` and `j`. The formula for specification `mutex` states that if thread `i` and `j` are not the same, then it is not possible that both threads are in the critical section at the same time. ⌟

In a specification formula it is only possible to use variables that have been declared as part of the analyzed LEAP input program or as parameter of the current specification. Inside a LEAP specification, global variables are accessed using their name. On the other hand, local variables need to specify the procedure they belong to using the "`::`" operator.

Example 9.5
Fig. 9.5 presents another example of a LEAP specification, named `activeLow`. This specification states that when a thread is at lines 4, 5 or 6, its *ticket* variable is lower than the global variable *avail*. In this case, instead of using a label as we did in previous examples, we are referring the program lines using their actual values (4, 5 and 6). Finally, note that as *ticket* is a local variable of procedure `setMutex`, we need to write `setMutex::ticket`. ⌟

A LEAP specification file can contain many formulas. In this case, LEAP considers all of them to be part of a conjunction. It is also possible to label a specific formula, so that it can be used later in isolation as part of a proof graph or a parametrized verification diagram without considering the rest of the formulas declared in the same specification file. We can use the "#" symbol to label a formula inside a specification file. Later, it is possible to use the "`::`" symbol to access the formula associated to a label. This means that `spec::inv` refers to the formula labeled by `inv` which is declared inside the specification identified with `spec`. It is also possible to use braces to describe a collection of formulas within the same specification. This means that `spec::{inv1,inv2,inv3}` refers to the formula constructed by the conjunction of the

```
vars:
  tid i

invariant [activeLow] :
  (@4(i). \/ @5(i). \/ $6(i).) -> setMutex::ticket(i) < avail
```

Figure 9.5: Example of a LEAP specification.

```
vars:
  tid i
  tid j

invariant [combined] :

#mutex:
  (i != j) -> ~ (@crit(i). /\ @crit(j).)


#activeLow:
  ( @4(i). \/ @5(i). \/ $6(i). ) -> setMutex::ticket(i) < avail

#others:
  @crit(i). ->
    ( i = tid_of(spmin(bag)) /\
      int_of(spmin(bag)) = setMutex::ticket(i)) /\

    intidpair(i, bag) = @active(i).
```

Figure 9.6: Example of a combined LEAP specification.

formulas labeled with `inv1`, `inv2` and `inv3` which are declared inside the specification named `spec`.

Example 9.6
Fig. 9.6 presents an example of a LEAP specification named `combined`, which is the result of the combination of specifications `mutex`, `activeLow` and a couple of extra new formulas.

This means that we can use `combined::mutex` to refer to the mutual exclusion formula, `combined::activeLow` to refer to the formula that states that each *ticket* is lower than *avail* and finally `combined::others` to refer to the conjunction of the last two formulas in the specification. Similarly, the formula resulting from the conjunction of all the formulas declared within specification `combined` can be described by `combined::{mutex,activeLow,others}`. ⌟

9.2.3 Domain Cut-offs

The decision procedures implemented by LEAP, in general, are based on an exhaustive search of the domain space. The bounds for a large enough domain are computed by each decision procedure following the guidances given in Chapters 6, 7 and 8. However, depending on the characteristics of the analyzed formula, the user may prefer to compute the cut-off for domain bounds using different approaches. In general, a technique that computes a more exact domain bound has the advantage of searching a smaller domain space at the price of a more exhaustive analysis of the verification conditions, which may be time consuming. On the other hand, a technique that computes a not so exact domain bound will be forced to search a bigger domain space but requires less time to analyze the verification condition in order to compute such bound. Independently of the selected cut-off method, LEAP implements some techniques for breaking variable symmetry [19], which help in reducing the search in the domain space. LEAP currently implements 3 techniques for computing the domain cut-offs:

DNF: by computing the disjunctive normal form of the formula contained in the verification condition it is possible to compute an exact bound for each domain, which results in a smaller domain space. However, this cut-off technique may present an inconvenience. In some situations, computing the disjunctive normal form of a formula results in a time consuming task.

The worst case scenario presents when we have to compute the disjunctive normal form of a formula that is in conjunctive normal formal. For example, consider the formula:

$$(A_1 \vee A_2 \vee A_3 \vee A_4) \wedge (B_1 \vee B_2 \vee B_3 \vee B_4)$$

In order to compute the disjunctive normal form of this formula, we need to distribute all disjunctions over the conjunction, obtaining:

$$\begin{array}{ccccccc}
A_1 \wedge B_1 & \vee & A_1 \wedge B_2 & \vee & A_1 \wedge B_3 & \vee & A_1 \wedge B_4 \\
A_2 \wedge B_1 & \vee & A_2 \wedge B_2 & \vee & A_2 \wedge B_3 & \vee & A_2 \wedge B_4 \\
A_3 \wedge B_1 & \vee & A_3 \wedge B_2 & \vee & A_3 \wedge B_3 & \vee & A_3 \wedge B_4 \\
A_4 \wedge B_1 & \vee & A_4 \wedge B_2 & \vee & A_4 \wedge B_3 & \vee & A_4 \wedge B_4
\end{array}$$

Pruning: the problem with computing the disjunctive normal form is the big number of operations between conjunctions and disjunctions. In the methods for computing domain bounds for each of the theories presented in Part II, not all literals are required in order to compute bounds. An alternative method which reduces the overhead of computing the disjunctive normal form of a formula consists of removing from the formula all literals that are not required for computing the domain bounds. This is what the pruning method does.

This pruning method traverses the formula and removes from the formula all literals that are not required for the computation of domain bounds. The result is a formula with less literals and consequently an easier disjunctive normal form. The pruning method proceeds as follows. First, it computes the negation normal form of the formula, by pushing all negations to the leaves of the formula. Then, all literals that are irrelevant for the computation of the cut-off are removed from the formula. Finally, the disjunctive normal form of the resulting formula is computed and the domain bounds are obtained as a result. It is important to note that literals are removed only for computing the domain cut-offs, but they are preserved in the formula on which satisfiability is checked.

Example 9.7
Consider the following formula:

$$\begin{array}{cccccccccc}
(& \neg b & \vee & pc(i) = 4 & \vee & pc(j) = 5 & \vee & p \neq q &) & \wedge \\
(& b & \vee & pc(i) = 8 & \vee & pc(j) = 9 & \vee & p = getp(heap, head, null) &) &
\end{array}$$

It is clear that neither the value of the Boolean variable b or the values of the program counters do not affect the computation of the domain cut-offs for the decision procedures

described in Part II. Therefore, we can get rid of these unnecessary literals using pruning. Then, we obtain the formula:

$$p \neq q \quad \vee \quad p = getp(heap, head, null)$$

which, in this case, happens to be already in its disjunctive normal form. ⌟

Union: this method avoids computing the disjunctive normal form at all. Instead, it traverses the formula and tries to guess which would be the maximum size of the domains in case all literals in the formula are present in the same conjunct. To do so, the union method begins by considering all variables of sort addr, elem and tid occurring in the formula. Then, the method considers all special cases, according to what is described in Theorem 6.1 for TL3 and in Theorem 7.1 for $\mathsf{TSL_K}$. The bounds computed following this method correspond to the scenario in which, after computing the disjunctive normal form of the formula, all variables and special literals are within one of the disjunctions. The cut-off obtained using this method may be greater than the one obtained using pruning or DNF, but it is in general faster to compute.

Example 9.8
Consider the following formula:

$$(head = null \vee tail = null) \wedge p \neq q \tag{9.1}$$

Following the DNF technique, we would first require to compute the disjunctive normal form of the formula, obtaining:

$$(head = null \wedge p \neq q) \quad \vee \quad (tail = null \wedge p \neq q)$$

and then, we would require to analyze each disjunction, concluding that we would require a domain with 3 addresses, as:

- for the first disjunction we require 1 address for $head$, 1 address for $null$ and 1 address to act as the witness of the inequality between paths p and q.
- for the second disjunction we require 1 address for $tail$, 1 address for $null$ and 1 address to act as the witness of the inequality between paths p and q.

On the contrary, union would just traverse formula 9.1 and would determine that it is necessary a domain with 4 addresses: 1 for variable $head$, 1 for variable $tail$, 1 for $null$ and 1 to act as witness of the inequality of p and q. Note how, using union instead of DNF, we can conclude that we require a domain with 4 addresses instead of 3. ⌟

9.2.4 Tactics

An important feature of LEAP are tactics. Tactics represent heuristics that can ease the verification process by simplifying the verification conditions under analysis.

Tactics start from a verification condition and guide the process of converting the verification condition into queries to the decision procedures. A verification condition, as implemented in LEAP, consists on three main components. These are:

- **Support:** A verification condition keeps a copy of all the formulas that it may require as support.
- **Transition relation:** This component stores the transition relation of the statement under analysis. That is, the relation between the variables in the pre and post states.
- **Goal:** As its name suggests it, this is the formula that the verification condition is trying to prove in the post state assuming the conditions given by the support and the transformations described by the transition relation.

Example 9.9
Consider once again program SETMUTEX presented in Example 2.2 and invariant candidates `activeLow` and `others` presented in Example 9.5. Imagine we want to check whether when thread k executes line 5 of program SETMUTEX using `others` as support, `activeLow` is preserved. Fig. 9.7 describes the components of a LEAP verification condition for this case.

Note that the formulas in the support are parametrized by arbitrary thread identifiers. The final real parameters are instantiated later, during the second stage of tactics application. ⌟

In general, applying a tactic to a verification condition does not result on an equivalent verification condition. However, soundness follows if the validity of the verification condition obtained by the application of a tactic implies the validity of the original verification condition.

There are 4 different stages of transformation that goes from the original verification condition to the final queries passed to the decision procedures. These stages are:

Verification condition split: The first stage consists on splitting the original verification condition into simpler verification conditions. Currently there exists only one tactic implemented for verification condition splitting:

Support:

```
@active(i1). -> setMutex::ticket(i1) < avail
intidpair(i2, bag) = @active(i2).
```

Transition relation:

```
pc(k) = 4 /\ pc'(k) = 5
```

Goal:

```
@active(k). -> setMutex::ticket(k) < avail
```

Figure 9.7: Example of a LEAP verification condition representation.

- **split-goal:** This tactic receives a verification condition and analyzes the goal. If the goal is a conjunction, then it generates as many verification conditions as elements in the conjunction. Clearly, if all the verification conditions generated are valid, then the original verification condition is also valid.

Support generation: The second stage is in charge of instantiating the support, transforming the verification condition into a formula with the shape of an implication. The generated implication contains, in the antecedent, the instantiated support and the transition relation. Meanwhile, the consequent of the generated implication is simply the goal of the verification condition. Currently, LEAP implements 4 different tactics for support generation:

- **full:** This tactic instantiates the support considering all partial substitutions from parameters of the support to the vocabulary of the transition relation and the goal. Remember that the vocabulary of a formula, according to Definition 2.4 is the set of free variables of type tid appearing in the formula.
- **reduce:** This tactic proceeds as tactic full, but it only considers complete substitutions. In practice, this leads to a reduced number of formulas as support, as it only considers some of the generated thread identifier substitutions.
- **reduce2:** This tactic is similar to reduce, except that it removes duplicated formulas when full support is applied.
- **identity:** This tactic does not perform any instantiation of the support, leaving the original thread identifiers used in the support formulas. This is equivalent as considering only the empty thread identifier substitutions.

Example 9.10
Consider once again the verification condition described in Example 9.9. In this verification condition, the support is parametrized by the thread identifiers `i1` and `i2`. Similarly, the vocabulary of the transition relation and the goal in this verification condition is just $\{\texttt{k}\}$. Table 9.1 shows the instantiated support that is obtained when using tactic full.

ID	Resulting formula	Substitution
$(S1)$	`@active(i1). -> setMutex::ticket(i1) < avail` `intidpair(i2, bag) = @active(i2).`	$\{\}$
$(S2)$	`@active(k). -> setMutex::ticket(k) < avail` `intidpair(i2, bag) = @active(i2).`	$\{\texttt{k} \leftarrow \texttt{i1}\}$
$(S3)$	`@active(i1). -> setMutex::ticket(i1) < avail` `intidpair(k, bag) = @active(k).`	$\{\texttt{k} \leftarrow \texttt{i2}\}$
$(S4)$	`@active(k). -> setMutex::ticket(k) < avail` `intidpair(k, bag) = @active(k).`	$\{\texttt{k} \leftarrow \texttt{i1}, \texttt{k} \leftarrow \texttt{i2}\}$

Table 9.1: Example of support generated using tactic full.

Consider the formulas $(S1)$, $(S2)$, $(S3)$ and $(S4)$ generated using tactic full in Table 9.1. If instead of tactic full we use tactic reduce, then only formula $(S4)$ is generated. Finally, if we use tactic identity, then only formula $(S1)$ is generated. ⌟

Formula split: The third stage is in charge of splitting the implications generated in the previous stage into simpler implications. The idea is to simplify the work for the decision procedures that will later check the validity of these implications. Currently, LEAP implements 2 different tactics for this stage:

- **split-consequent:** This tactic receives as input an implication and produces a new set of implications by splitting the consequent of the input implication. To do so, the consequent of the input implication needs to be a conjunction. For the generated implications, the antecedent remains untouched but each conjunction in the consequent is used as a consequent for each of the new generated implications. That is, given an implication of the form:

  $$A \rightarrow C_1 \wedge C_2 \wedge C_3$$

  Applying tactic split-consequent generates three new implications of the form:

  $$\begin{aligned} A &\rightarrow C_1 \\ A &\rightarrow C_2 \\ A &\rightarrow C_3 \end{aligned}$$

- **split-antecedent-pc:** This tactic analyzes the antecedent of an implication searching for a disjunction of program locations. If the implication contains a disjunction of program locations in the antecedent, then new implications are created, each of which contains a single program location in the antecedent.

Example 9.11
Consider the following implication:

$$A \wedge (pc = 1 \vee pc = 2) \rightarrow C$$

The tactic split-antecedent-pc generates the following implications:

$$\begin{aligned} A \wedge pc = 1 &\rightarrow C \\ A \wedge pc = 2 &\rightarrow C \end{aligned}$$

⌟

Formula simplification: The forth and final stage is to simplify the implications before passing them to the decision procedures.

Currently, LEAP implements 6 different tactics for formula simplification:

- **simplify-pc:** This tactic analyzes the antecedent of the implication searching for program location equalities of the form $pc = n$. If it finds some contradiction, then assumes the implication to be $true$. On the other hand, if there is no contradiction, then it propagates these equalities to the whole formula. The effect of this tactic is to remove from the implication subformulas which are not required except for program locations.
- **simplify-pc-plus:** This tactic operates in two steps. First, it applies tactic simplify-pc and then it analyzes the resulting implication. If the consequent of the implication resulting from applying tactic simplify-pc is of the form $A \rightarrow (B \rightarrow C)$ and B is a restriction over program locations, then this tactic further uses B as a fact and propagates facts in B about program locations through A.
- **propositional-propagate:** This tactic analyzes the antecedent of the implication. If the antecedent is a conjunction, then each of the conjuncts are considered to be facts. Each of these facts are propagated to the whole formula.
- **filter-strict:** This tactic analyzes the implication and eliminates from the antecedent all literals that do not have variables in common with literals from the consequent.
- **propagate-disj-conseq-fst:** This tactic simplifies the implication by using facts from the consequent, following an approach similar to tactic simplify-pc-plus. If the implication is of the form $A \rightarrow (l \rightarrow C)$, where l is a literal, then it adds l to the consequent and uses it to further simplify A and C.
- **propagate-disj-conseq-fst:** This tactic is very similar to tactic propagate-disj-conseq-fst, with the difference that it considers implications of the form $A \rightarrow (B \rightarrow l)$. In this case, assumes $\neg l$ as a fact and propagates it in the whole implication.

Example 9.12
Consider the following implication:

$$\begin{pmatrix} pc = 2 \ \wedge \ pc' = 3 \ \wedge \ A_1 & \wedge \\ (pc = 2 \rightarrow A_2) & \wedge \\ (pc = 6 \rightarrow A_3) & \wedge \\ (pc = 8 \vee pc = 9) \rightarrow A_4) & \end{pmatrix} \rightarrow \begin{pmatrix} C_1 & \wedge \\ (pc' = 3 \rightarrow C_2) & \wedge \\ (pc' = 6 \rightarrow C_3) & \end{pmatrix}$$

Analyzing the antecedent, the tactic determines that $pc = 2$ and $pc' = 3$ are two facts. Then, propagating them results in the following simplified implication:

$$A_1 \wedge A_2 \quad \rightarrow \quad C_1 \wedge C_2$$

⌟

Example 9.13
Consider the following implication:

$$\Big(a = null \wedge a \neq b\Big) \quad \rightarrow \quad \begin{pmatrix} a = null \rightarrow reg = \emptyset & \wedge \\ a = b \rightarrow elems = \emptyset & \end{pmatrix}$$

In this case, from the antecedent we can learn that $a = null$ and $a \neq b$. Hence, applying tactic propositional-propagate with the facts stated above, we can reduce the original implication to $(a = null \wedge a \neq b \wedge reg = \emptyset)$. ⌟

9.3 Verification using LEAP

As LEAP implements the parametrized verification techniques presented in this work, it can be used for the verification of both safety and liveness properties. Safety properties are verified with the assistance of proof graphs, while liveness properties are verified using parametrized verification diagrams. We now illustrate with a simple example the verification of safety and liveness properties in LEAP.

9.3.1 Safety Properties

In order to prove that a program executed by an unbounded number of threads satisfies a temporal safety specification, LEAP requires the following elements:

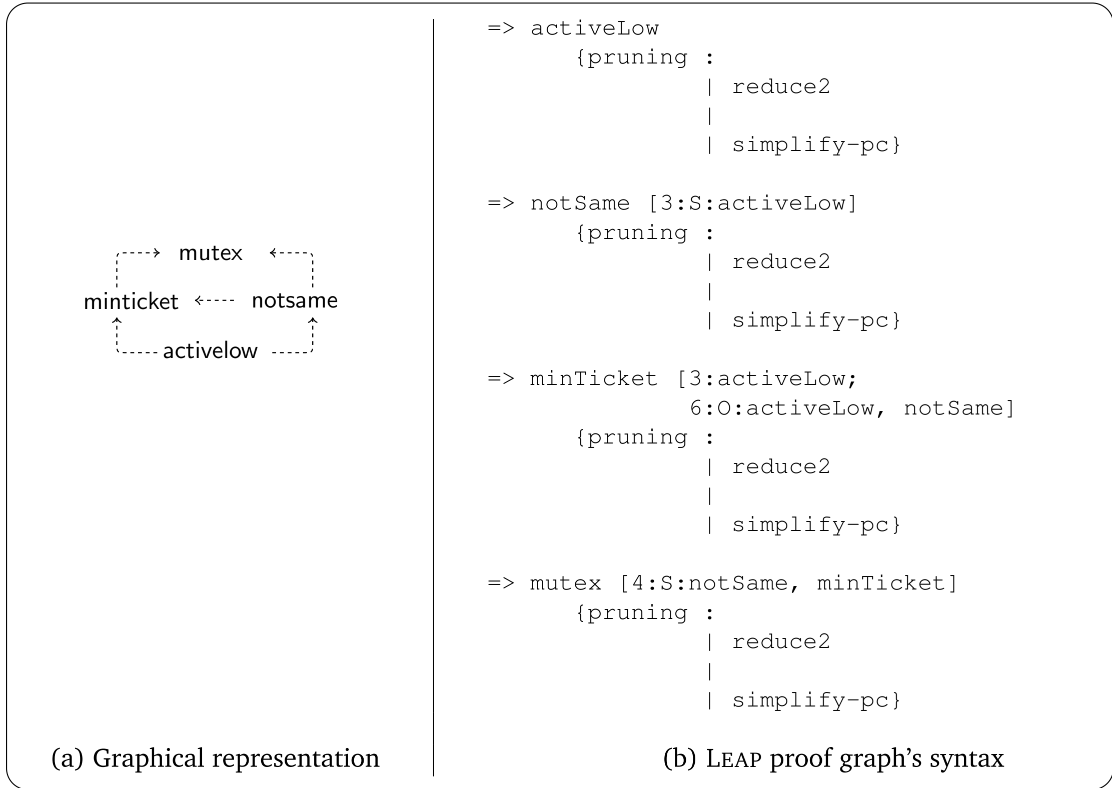
1. the **program**;
2. a **collection of invariant candidates**, which can be used as goals or as support; and
3. a **proof graph**, which declares how invariant candidates relate with each other and which tactics are required to be used in each case.

Currently, the program needs to be written in the internal format of LEAP. See, for instance, Example 9.1. The full syntax for LEAP input programs can be found in Appendix B.2.1.

The collection of invariants follow the syntax for LEAP specifications. We have already seen some examples in Example 9.4 and Example 9.5. Again, the full syntax for LEAP specifications can be consulted in Appendix B.2.2.

Finally, the proof graph is used by LEAP to exploit the inter-dependency between invariant candidates. Additionally, the proof graph contains some hints in the form of tactics on how to assist the verification process. Proof graphs are designed to improve the efficiency of proof development and proof checking, by establishing the necessary support for proving consecution and optionally specifying tactics and heuristics. The full syntax for LEAP proof graphs can be seen in Appendix B.2.3.

Example 9.14
Consider specifications `mutex` and `activeLow` presented in Example 9.4 and Example 9.5. Fig. 9.8 shows an example of a proof graph which uses these two specifications in addition to a couple of new invariant candidates named `notSame` and `minTicket`. The proof graph shown in the figure shows that program SETMUTEX satisfies the safety specification `mutex`. In the figure, on the left is the graphic representation of the proof graph, showing the inter-dependency between invariant candidates. On the right, the figure shows a description of the proof graph using LEAP syntax. ⌟



(a) Graphical representation

(b) LEAP proof graph's syntax

Figure 9.8: Example of a LEAP proof graph.

A proof graph is a collection of rules, each of which tells LEAP how to prove a specific invariant candidate. Note that each rule starts with the "=>" operator. Following this operator, it is the name of the invariant candidate to be proven. The double arrow operator "=>" indicates the use of parametrized proof rules. Optionally, we could use the single arrow operator "->" to specify the use of traditional non-parametrized invariance proof rules to verify non-parametrized programs. This feature allows LEAP to analyze sequential closed systems composed by a single thread.

Specifications which are required as support by an invariant candidate can be specified between brackets following the invariant candidate identifier. For example, note how in Fig. 9.8 invariant candidates `notSame`, `minTicket` and `mutex` are followed by specifications between brackets, while specification `activeLow` is not. This is because invariant candidate `activeLow` is inductive, and hence it does not require any support.

The support provided between brackets can be restricted to certain transitions and premises. By doing so, we are reducing the size of the resulting formulas and easing the job of the decision procedures. For a support specification we can indicate:

1. the program line for which such specification needs to be instantiated; and
2. a switch indicating whether the support needs to be instantiated when considering a thread appearing in the formula (`S`) or other fresh thread (`O`).

Alternatively, support specifications can be declared preceding the arrow operator. In this case,

the support is instantiated for all transitions and proof rules.

Example 9.15
Consider the following rule from the proof graph presented in Fig. 9.8(b):

```
=> minTicket [3:activeLow;
              6:O:activeLow, notSame]
```

This part says that, in order to prove `minTicket` invariant, it is useful to instantiate `activeLow` as support for line 3. Additionally, this entry also suggests to instantiate `activeLow` and `notSame` when considering line 6, but only when a fresh thread is executing this transition.

Alternative, we could write the above rule as:

```
activeLow, notSame => minTicket
```

In which case, specifications `activeLow` and `notSame` are instantiated for all transitions and premises (self-consecution and others-consecution). ⌟

The last component of a rule in a proof graph is a collection of cut-off methods and tactics declared between braces. The format is, first the cut-off technique followed by the list of tactics. The division between the cut-off strategy and the tactics is specified by a colon. A list of available cut-off strategies can be found in Section 9.2.3. Tactics are declared in four different sections, each of them limited by the "`|`" operator. Each of the sections correspond to each of the stages at which tactics can be applied, as introduced in Section 9.2.4.

Example 9.16
Consider the following part of a rule. It is a variation of a rule declared in the proof graph introduced in Fig. 9.8(b):

```
{pruning : split-goal
         | reduce2
         | split-consequent
         | simplify-pc propositional-propagate}
```

The rule above says that:

1. strategy pruning is used for computing the domain cut-offs;
2. tactic split-goal is used in order to split the original verification condition into simpler ones;
3. tactic reduce2 is used for instantiating the support;
4. tactic split-consequent is used for splitting the implications resulting from the previous stage into simpler formulas; and finally
5. tactics simplify-pc and propositional-propagate are used for simplifying the formulas before passing them to the decision procedures. ⌟

Once LEAP receives the program description, the collection of invariant candidates and the proof graph, LEAP analyzes them and outputs whether the invariant candidates declared in the proof graph are in fact invariants of the parametrized system.

9.3.2 Liveness Properties

The verification of liveness properties using LEAP is similar to the verification of safety properties. It requires a collection of input files. The data required by LEAP in order to verify a liveness property is:

1. the **program**;
2. a **collection of specifications**, which is used as support during the verification process;
3. a **parametrized verification diagram**, which represents the diagram that acts as witness of the proof; and
4. a **tactics file**, which indicates the tactics and support to be used.

The program and each of the specifications needs to be written in the LEAP format, as specified in Appendix B.2.1 and Appendix B.2.2.

The parametrized verification diagram, on the other hand, follows the syntax described in Appendix B.2.4. A PVD is a witness that the parametrized system composed by an unbounded number of threads executing the concurrent program provided as input does satisfy a temporal property. The file describing the PVD is divided into sections, each describing a particular element of the diagram. Each section is identified with a special keyword and they must be declared following a strict order:

Diagram: this keyword must appear at the beginning of every PVD to indicate the starting point of a new PVD declaration and it is followed by a string which identifies the PVD.

Nodes: to indicate the declaration of the nodes of the PVD. Each node consists of a identifier and a formula labeling the node. The formula labeling a node must be written between braces and follows the same syntax as for formulas declared in LEAP invariant specifications. Finally, each node declaration must be separated by a comma.

Boxes: this keyword is used to declare boxes in the PVD. Boxes must be declared inside braces and consist of an identifier, a thread identifier which parametrizes the box and the list of nodes included in the box.

Initial: this keyword is used to indicate the initial node of the PVD.

Edges: this keyword precedes the declaration of the edges of the PVD. Each edge consists of two node identifiers connected by an arrow "-->". The arrow can optionally be labeled with a program transition and each edge needs to be terminated with a semicolon. Finally, an edge can be placed inside brackets, meaning that the edge connects two nodes within a box preserving the box variable.

Acceptance: This keyword indicates that the acceptance conditions of the PVD are going to be declared. An acceptance condition is declared between "<<>>" and consists of three sections. The first section starts with the `Bad` keyword and indicates the edges which should strictly decrease the ranking function. The second section starts with the `Good` keyword and declares the edges for which the ranking function has no restriction. The third section is the ranking function itself expressed as a state expression and a binary predicate. An important feature is that LEAP supports ranking functions including lexicographic orders.

Example 9.17
Consider the PVD desceribed in Fig. 4.6 from Chapter 4. We can declare nodes n_1, n_2, n_3 and n_4 of the diagram as:

```
Nodes : n1 { @3(k).},
        n2 { @4(t). /\ k!=t /\ @4(k). /\
             int_of(spmin(bag)) = main::ticket(t)},
        n3 { @5(t). /\ k!=t /\ @4(k). /\
             int_of(spmin(bag)) = main::ticket(t)},
        n4 { @5(t). /\ k!=t /\ @4(k). /\
             int_of(spmin(bag)) = main::ticket(t)}
```

where `int_of(spmin(bag))` refers to the integer component of the minimum pair stored in variable `bag`. For the same diagram, box `b1` is parametrized by thread identifier `t` and contains nodes n_2, n_3 and n_4:

```
Boxes: {b1[t]:n2,n3,n4}
```

Furthermore, we can declare the edges connecting nodes n_0 with n_1, n_1 with n_2, n_2 with n_3, n_3 with n_2 and n_4 with n_2 using the following syntax:

```
Edges:  n0 --> n1;
        n1 -{3(k)}-> n2;
       [n2 -{4(t)}-> n3];
        n3 -{6(t)}-> n2;
        n4 -{6(t)}-> n2;
```

The declaration above establishes that node `n0` is connected to node `n1` by an edge with no labels. Node `n1` is connected to node `n2` through an edge labeled by the transition that corresponds to program line 3 when executed by thread `k`. Node `n2` is connected to node `n3` by an edge labeled with the transition that corresponds to program line 4 when executed by thread `t`. The brackets indicate that this edge is contained within a box, and thus it must preserve the parameter of the box. Finally, note that the edges labeled with transition `6(t)` are not surrounded by brackets even though they connect two nodes within a box. This means that the edge in fact leaves the box before entering it again, which means that the box parameter (thread identifier `t` in this case) does not need to be preserved when this edge is taken.

Now, consider the following acceptance condition:

```
Acceptance:
  <<Bad :{(n0,n1,any),(n1,n2,any)};
    Good:{(n2,n3,any)};
    [(main::ticket(k), less)] >>
```

The acceptance condition above states that when an edge connecting node `n0` with node `n1` or connecting node `n1` with node `n2` is taken, then the ranking function should strictly decrease. On the other hand, when taking the edge that connects node `n2` with node `n3`, the value of the ranking function is not taken into consideration. The ranking function for this acceptance condition indicates that the value of *ticket* for thread k should decrease.

Note that the acceptance condition provided above does not represent the acceptance condition for the PVD presented in Fig. 4.6 and it is just illustrative. ⌟

Example 9.18
Consider once again the PVD shown in Fig. 4.6 in Chapter 4. Fig. 9.9 presents the declaration of a slightly modified version of such PVD following the syntax accepted by LEAP and containing all the elements that define a PVD. ⌟

```
Diagram[ticketset]

Nodes: n0 {@1(k). \/ @2(k). \/ @7(k). \/ @8(k).},
       n1 {@3(k).},
       n2 {@4(t). /\ k!=t /\ @4(k). /\ int_of(spmin(bag))=main::ticket(t)},
       n3 {@5(t). /\ k!=t /\ @4(k). /\ int_of(spmin(bag))=main::ticket(t)},
       n4 {@6(t). /\ k!=t /\ @4(k). /\ int_of(spmin(bag))=main::ticket(t)},
       n5 {@4(k). /\ int_of(spmin(bag)) = main::ticket(k)},
       n6 {(@5(k). \/ @6(k).) /\ int_of(spmin(bag)) = main::ticket(k)}

Boxes: {b1[t]:n2,n3,n4}

Initial: n0

Edges:  n0 --> n1;
        n1 -{3(k)}-> n2;
        n1 -{3(k)}-> n3;
        n1 -{3(k)}-> n4;
        n1 -{3(k)}-> n5;
       [n2 -{4(t)}-> n3];
       [n3 -{5(t)}-> n4];
        n4 -{6(t)}-> n2;
        n4 -{6(t)}-> n3;
        n4 -{6(t)}-> n4;
        n4 -{6(t)}-> n5;
        n5 -{4(k)}-> n6;
        n6 --> n0;

Acceptance:
  << Bad : { (n4,n2,any),(n4,n3,any),(n4,n4,any),(n4,n5,any) };
     Good: { (n6,n0,any) };
     [(splower(bag,main::ticket(k)), pairsubset_op)] >>
```

Figure 9.9: Full example of PVD in LEAP format.

In general, in order to prove the conditions related to a PVD we assume some conditions for the system. This assumptions, called facts, and some extra information such as cut-off strategies and tactics are specified on an auxiliary support file passed to LEAP. This kind of support files contain two main sections. The first section is labeled with the `Tactics` keyword and declares the cut-off strategies and tactics to be used. The second section, labeled with the `Facts` keyword, indicates the specifications that are required to be used as assumptions for specific transitions. In general, these formulas are invariants that have already been proven to be system invariants using the parametrized invariance rules presented in Chapter 3.

There are two kind of tactics which can be declared in the tactics section of a support file. The most important are general tactics, which specifies the tactics that should be used in general for the whole diagram. Additionally, it is possible to declare specific tactics to be used only on particular transitions, nodes or PVD conditions. In the case of facts, they are related to specific transitions.

Example 9.19
We present an example of a support file which declares some tactics and facts to be used in the verification of a PVD:

```
Tactics :
  {pruning : split-goal
           | reduce2
           |
           | simplify-pc } ;
  3 : [n1,n2|C] : {union :
                         | reduce2
                         |
                         | simplify-pc filter-strict } ;

Facts :
   2:activeLow;
   3:notSame,minTicket;
```

The supporting file declared above states that strategy pruning and tactics split-goal, reduce2 and simplify-pc are used in general for all cases. However, there is one exception. When we consider the transition that corresponds to line 3 of the program and we are analyzing nodes `n1` and `n2` under conditions (SelfConsec) and (OtherConsec), the declaration above says that we need to use cut-off strategy union and tactics reduce2, simplify-pc and filter-strict.

Finally, the declarations states that specification `activeLow` needs to be used as an assumption when considering the transition associated to program line 2. Similarly, specifications `notSame` and `minTicket` are required as assumptions for verifying the transition associated to program line 3. ⌟

There is one last activity that needs to be performed, and it is that the PVD is in fact captured by the liveness property we are trying to verify. This is done by checking condition (ModelCheck). This check is currently not implemented as part of LEAP, but it can done using the diagram translation described in Appendix A and an external off-the-shelf model checker.

9.3.3 Decision Procedures

In Section 9.3.1 and Section 9.3.2 we briefly described how to use LEAP in order to check safety and liveness properties of parametrized systems. In both cases, the intermediate object is a finite collection of verification conditions. The validity of these verification conditions is automatically checked using specialized decision procedures implemented as part of LEAP.

The validity of these verification conditions is automatically checked using specialized decision procedures implemented as part of LEAP.

LEAP currently implements all the decision procedures presented in Part II of this work. LEAP automatically transforms each verification condition into queries to the corresponding decision procedures. Additionally, LEAP implements some simple decision procedures based on program location reasoning, which help in checking the validity of many verification conditions before a specialized decision procedure is required.

Finally, for each generated verification condition, LEAP indicates whether the verification condition is valid or not. If a verification condition is not valid, then the decision procedure generates a counter-model illustrating an offending small step of the system that leads a violation of the property. This is typically a very small heap snippet that the programmer can use to either identify a bug or instrument the program with intermediate invariants, or additionally assumptions and facts.

9.4 Summary

In this chapter we have presented LEAP, a prototype theorem prover for the verification of parametrized concurrent systems that manipulate complex data types. LEAP implements the parametrized proof rules and the parametrized verification diagrams presented in Chapter 3 and Chapter 4 respectively. Additionally, LEAP implements the decision procedures described in Part II of this work

We have presented the main ideas behind LEAP and we have briefly described how LEAP works and which are some of the features it implements. The modular design of LEAP makes it straightforward to implement extensions for new program statements, theories and decision procedures.

Later, in Chapter 10 we report the empirical results we have obtained so far using LEAP for the verification of safety and liveness properties on some mutual exclusion protocols and programs that manipulates concurrent data types such as stacks, queues, lists and skiplists.

10

Experimental Results

> *The difference between screwing around and science, is writing it down.*
>
> Adam Savage (The MythBusters)

In this chapter we report some experimental results we have obtained using LEAP (presented in Chapter 9) on a set of programs that manipulate complex data types.

In Chapter 3 we introduce the technique of *parametrized invariance* for the verification of safety properties of parametrized systems. Here we use parametrized invariance in combination with the decision procedures presented in Part II in order to verify some implementations of concurrent data structures.

LEAP is described in more detail in Chapter 9. Before using a full-fledged decision procedure, LEAP first attempts to verify a verification condition using a simpler decision procedure that is only capable to reason about program locations. This decision procedure allows to prove simple formulas before performing complex model searches. In the case of safety, LEAP can prove invariant candidates separately, relying on other invariant candidates, using a proof graph as described in Section 3.2.4. If all verification conditions are verified, then we can conclude that all formulas are indeed invariants. In the case of liveness properties, a parametrized verification diagram is used, as described in Section 4.2.

All the experiments presented here were carried out using a computer with a 2.8 GHz processor and 8GB of memory running a version of LEAP compiled for Linux. Through this chapter we briefly describe the specifications and programs that were used to test our framework. In some cases, we provide the full specification here. Properties and specifications not fully described in this chapter can be consulted in LEAP's website [1].

The rest of this chapter is structured as follows. Section 10.1 briefly shows the results we have obtained verifying safety properties for the mutual exclusion protocol presented in Example 2.2.

[1] `http://software.imdea.org/leap`

Section 10.2 presents the verification of some safety properties using the TL3 decision procedure presented in Chapter 6 for concurrent lists, stacks and queues. Section 10.3 shows the verification of some safety properties of bounded skiplists using an implementation of the TSL_K decision procedure introduced in Chapter 7. Section 10.4 describes the verification of safety properties using the TSL decision procedure described in Chapter 8 over non-concurrent unbounded implementations of skiplists, including an implementation which is part of the KDE library. Section 10.5 presents some results we have obtained using parametrized verification diagrams in order to verify liveness properties of a mutual exclusion protocol and an implementation of concurrent lock-coupling lists. Finally, Section 10.6 presents the empirical evaluation of using the parametrized invariant generation technique presented in Chapter 5 on a collection of parametrized examples.

10.1 Verification of Safety Properties in Numeric Programs

In this section we describe our experience using LEAP for the verification of a mutual exclusion protocol based on tickets. The protocol was presented in Chapter 2 as a motivating example. In Example 2.2 we presented two versions of such protocol. INTMUTEX, which is based solely on integers, and SETMUTEX, which uses a set for keeping the active tickets.

For program SETMUTEX we verify the following safety specifications:

setMutex: which specifies mutual exclusion:

$$\mathsf{setMutex}(i,j) \quad \stackrel{\text{def}}{=} \quad \Box\ (i \neq j \rightarrow \neg(pc(i) = 5,6 \wedge pc(j) = 5,6))$$

setMinTicket: which states that a thread in the critical section must own the minimum ticket in the set of tickets:

$$\mathsf{setMinTicket}(i) \quad \stackrel{\text{def}}{=} \quad \Box\ (pc(i) = 5,6 \rightarrow \mathit{min}(bag) = main{::}ticket(i))$$

setNotSame: which establishes that if two different threads own a ticket, then these tickets should be different:

$$\mathsf{setNotSame}(i,j) \quad \stackrel{\text{def}}{=} \quad \Box \begin{pmatrix} i \neq j & \wedge \\ pc(i) = 4..6 & \wedge \\ pc(j) = 4..6 & \end{pmatrix} \rightarrow main{::}ticket(i) \neq main{::}ticket(j)$$

setActiveLow: which states that if a thread has a ticket, then this ticket is strictly lower than the global ticket to be assigned to the next thread:

$$\mathsf{setActiveLow}(i) \quad \stackrel{\text{def}}{=} \quad \Box\ (pc(i) = 4..6 \rightarrow main{::}ticket(i) < avail)$$

For program INTMUTEX we require similar safety specifications:

intMutex: identical to **setMutex**.

	form. info		#solved VC		single VC time(s.)		num DP	LEAP
	id	#VC	pos	num	slowest	average	time(s)	time(s)
setMutex	2	28	26	2	0.01	0.01	0.01	0.01
setMinTicket	1	19	17	2	0.01	0.01	0.01	0.01
setNotSame	2	28	26	2	0.01	0.01	0.02	0.01
setActiveLow	1	19	17	2	0.01	0.01	0.01	0.01
intMutex	2	28	26	2	0.01	0.01	0.02	0.01
intMinTicket	1	19	18	1	0.01	0.01	0.01	0.01
intNotSame	2	28	26	2	0.01	0.01	0.02	0.01
intActiveLow	1	19	17	2	0.01	0.01	0.01	0.01

Table 10.1: Verification conditions (VCs) proved and executing time using a numeric decision procedure for verifying safety properties of a mutual exclusion protocol.

intMinTicket: which states that a thread in the critical section must own the minimum ticket:

$$\text{intMinTicket}(i) \quad \stackrel{\text{def}}{=} \quad \square \ (pc(i) = 5, 6 \rightarrow min = main{::}ticket(i))$$

intNotSame: which states that different threads own different tickets. The specification of this property is identical to **setNotSame**.

intActiveLow: which states that if a thread has a ticket then the ticket must be lower than $avail$. The specification of this property is the same as **setActiveLow**.

Table 10.1 presents the results obtained using LEAP to verify the aforementioned properties. Each row in the table reports the empirical results obtained when proving a single candidate invariant. The first column shows the index of the formula, i.e. the number of threads parametrizing the safety property. The second column contains the total number of generated verification conditions. The third column shows the number of verification conditions successfully proved by a program location decision procedure. The fourth column reports the number of verification conditions proved valid using the specialized decision procedure for numeric programs. For every candidate invariant, all verification conditions are proved valid. The next three columns report the fastest, slowest and average time for verifying the validity of a single verification condition using the specialized decision procedure. Finally, the last columns report the total running time taken by the decision procedure to check the validity of all verification conditions, and the total running time taken by LEAP for the generation and discharge of all verification conditions (excluding the running time of the decision procedures).

10.2 Verification of Safety Properties using TL3

In this section we report the results of the empirical evaluation using LEAP to verify the lock-coupling list implementation presented in Section 6.1, the unbounded queue of Section 6.1.2, the lock-free implementation of a stack shown in Section 6.1.3 and the lock-free queue presented in Section 6.1.4.

10.2.1 Lock-Coupling Lists

For lock-coupling concurrent lists we prove that the most general client of the data type shown in Fig. 6.4 satisfies:

1. the layout in the heap is always that of a single-linked list;
2. the data type implements a set, whose elements correspond to those elements stored in the ghost variable $elems$.

We define the following specifications and prove them to be invariants:

list: this specification models list shape preservation. The formula list is 0-index because it only refers to global program variables. Unfortunately, list is not an inductive invariant, so support invariants are needed to prove that list is an invariant. The required support invariants are listed below.

region: is a 1-index formula that describes that when local variables $prev$, $curr$ and aux point to cells contained or not in region reg. This formula also captures how global program variable reg is modified through the execution of the procedures which insert and remove elements from the list.

next: specifies the relative position in the list of the cells pointed by $head$ and $tail$ and local variables $prev$, $curr$ and aux.

order: captures the order between the data stored at cells pointed by $curr$, $prev$ and aux. Additionally, it tracks the order relation between elements stored in the list and the element e used as an argument of procedures SEARCH, INSERT and REMOVE.

lock: identifies those program locations at which a thread owns a cell in the heap by a previous acquisition of its lock.

disj: encodes the fact that invocations to *malloc* by different threads return non-aliasing (separated) cells. The formula disj is a 2-index formula, because it needs to refer to local variables of two different threads.

The formal specification of all these invariants is quite large, and hence we do not show them here. However, the full formal definition of each of these invariants can be found in the examples section at LEAP's website.

We also verify some functional properties of the concurrent lock-coupling list implementation. For example, invariant **funSchLin** establishes that procedure SEARCH returns $true$ if and only if the searched element e is present at SEARCH's linearization point. We can check that this specification holds by verifying that at SEARCH's linearization point (line 12):

$$\mathsf{funSchLin}(i) \quad \stackrel{\text{def}}{=} \quad \square\ (pc_{\text{SEARCH}}(i) = 12 \rightarrow (heap[curr(i)].data = e(i) \leftrightarrow e(i) \in elems))$$

For verifying other functional properties, we just need to slightly modify the lock-coupling implementation presented in Section 6.1 adding more ghost code. For example, consider the program depicted in Fig. 10.1. This code presents the modifications to the original lock-coupling single-linked list implementation required to verify some functional properties. We introduce

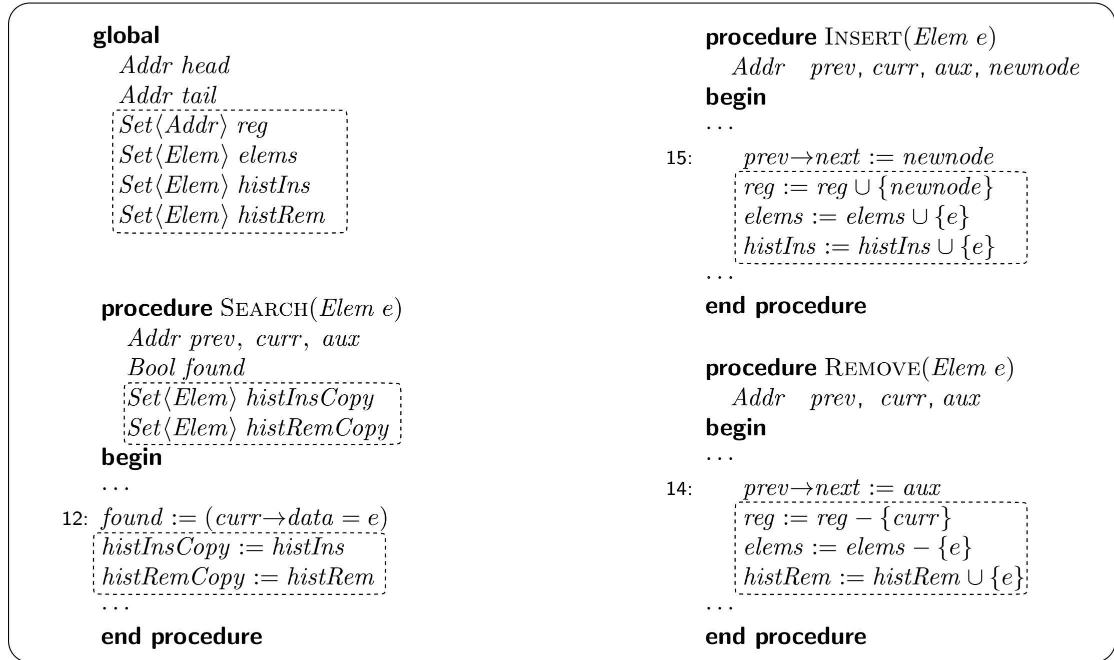


Figure 10.1: Modifications needed to apply over the lock-coupling implementation of Section 6.1 in order to verify some functional properties.

two new global ghost program variables named *histIns* and *histRem*. These two program variables are updated at INSERT and REMOVE procedure with the elements that are inserted and removed respectively. Procedure SEARCH also declares two local ghost program variables named *histInsCopy* and *histRemCopy*. These program variables keep a copy of *histIns* and *histRem* when procedure SEARCH goes through its linearization point, at line 12.

We can now use the procedures SEARCH, INSERT and REMOVE extended with the new ghost notation to prove the following functional properties:

funSchIns: this specification states that, if a call to SEARCH with argument e returns *true*, then e has been previously inserted into the list. As the history of elements inserted into the list is kept by the global program variable *histIns*, then we can prove this specification by verifying that after SEARCH's linearization point the following formula holds:

$$\mathsf{funSchIns}(i) \stackrel{\text{def}}{=} \square\ (pc_{\text{SEARCH}}(i) = 12 \rightarrow (\mathit{found}(i) \rightarrow e \in \mathit{histIns}))$$

funSchRem: this specification captures the fact that if a search is unsuccessful then either e was never inserted or it was removed. In any case, element e was not present at SEARCH's linearization point. That is:

$$\mathsf{funSchRem}(i) \stackrel{\text{def}}{=} \square \left(pc_{\text{SEARCH}}(i) = 12 \rightarrow \left(\begin{array}{l} \mathit{histIns} \subseteq (\mathit{elems} \cup \mathit{histRem}) \wedge \\ \neg\, \mathit{found}(i) \rightarrow \left(\begin{array}{l} e(i) \notin \mathit{histInsCopy} \vee \\ e(i) \in \mathit{histRemCopy} \end{array} \right) \end{array} \right) \right)$$

This formula states that:

(a) If an element was at some point inserted into the list, then it is still in the list or it has been removed by a call to REMOVE.

(b) If element e was not found by SEARCH, then either element e has never been inserted or element e has been removed by a previous call to REMOVE.

The local copy of $histIns$ and $histRem$ at SEARCH serve as a snapshot of the composition of the list at the linearization point of SEARCH.

We can verify other functional properties just by making a slight change to other parts of the program. For example, we can modify the most general client of the lock-coupling list implementation so that only a specific thread uses a specific argument and call. Following this idea, we can declare some specifications such as **funRemove**, **funInsert** and **funSearch**, which are restricted to the case in which one thread handles different elements than all other threads. In this case, the specification is similar to a sequential functional specification: an element is found if and only if it is in the list, an element is not present after removal, and an element is present after insertion.

Fig. 10.2 shows the modifications that are required in order to verify **funSearch**. We pick a chosen thread named $chosen$ and a chosen element named $chosen_e$. The most general client for this example consists of an infinite loop which checks whether the executing thread corresponds to $chosen$. If so, the thread performs a call to SEARCH looking for $chosen_e$ element. In the case the executing thread is not $chosen$, then it gets an arbitrary element and, if this element is not equal to $chosen_e$, non-deterministically performs a call to one of the operations implemented by the list.

For verifying properties **funInsert** and **funRemove** we follow a similar approach. In the case of **funSearch**, the property states that the result of a call to SEARCH over the chosen element

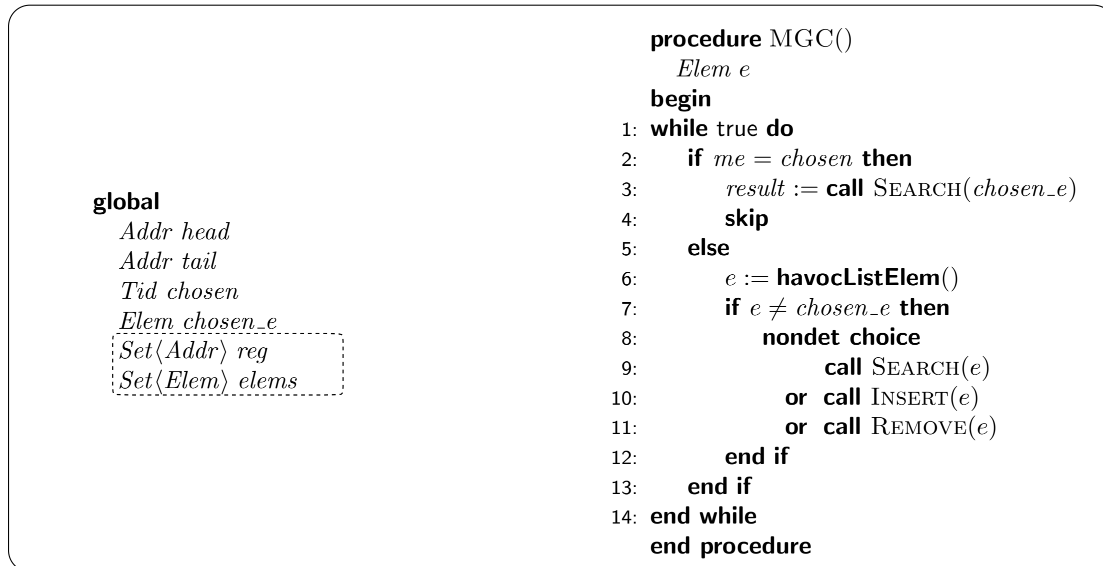


Figure 10.2: Modifications needed to apply over the lock-coupling implementation of Section 6.1 in order to verify funSearch properties.

	form. info		#solved VC		single VC time(s.)		TL3 DP	LEAP
	id	#VC	pos	TL3	slowest	average	time(s)	time(s)
list	0	61	38	23	10.16	0.27	16.57	0.19
order	1	121	62	59	0.10	0.01	0.70	0.38
lock	1	121	76	45	0.72	0.01	1.38	0.15
next	1	121	60	61	0.66	0.01	1.56	0.50
region	1	121	102	19	8.88	0.09	10.62	0.19
disj	2	181	177	4	0.28	0.00	0.64	0.19
funSchLin	1	121	97	24	0.92	0.02	1.95	0.05
funSchIns	1	208	199	9	0.15	0.00	0.62	0.86
funSchRem	1	208	199	9	0.36	0.00	0.93	1.04
funSearch	1	208	197	11	0.40	0.01	1.21	0.78
funInsert	1	208	197	11	0.16	0.01	0.72	0.62
funRemove	1	208	197	11	0.35	0.02	1.07	0.82

Table 10.2: Verification conditions (VCs) proved and executing time using TL3 decision procedure for verifying safety properties of a concurrent lock-coupling list.

should match the presence of the element in the list:

$$\mathsf{funSearch}(i) \stackrel{\text{def}}{=} \Box \; (pc_{\mathsf{MGC}}(i) = 4 \rightarrow (chosen_e \in elems \leftrightarrow result(chosen)))$$

Table 10.2 presents the results we have obtained in the verification of the properties stated above.

10.2.2 Unbounded Lock-based Queue

For the coarse-grain lock-based concurrent queue presented in Section 6.1.2 we prove queue shape preservation, expressed as follows:

$$\mathsf{unbQueuePres} \stackrel{\text{def}}{=} \Box \begin{pmatrix} null \in reg & \wedge \\ tail \in reg & \wedge \\ tail \neq null & \wedge \\ reg = addr2set(heap, head) & \wedge \\ head \neq null & \end{pmatrix}$$

In this case, similar to concurrent lock-coupling lists, we require auxiliary invariants such as **unbQueueNext**, which describes the relation between pointers, and **unbQueueLock**, which describes when locks are owned by a thread. The full specifications for these properties can be found in LEAP's website.

For this data type we also verify that **unbQueueInc** is an invariant. Formula **unbQueueInc** specifies that all elements introduced in the queue are either still in the queue or they have been extracted through a call to the dequeue function. In order to check this property, we need to add auxiliary ghost variables $enqueueSet$ and $dequeueSet$ to keep track of the elements inserted and removed form the queue through calls to ENQUEUE or DEQUEUE.

```
global
   Addr head, tail
   Lock queueLock
   Set⟨Elem⟩ enqueueSet
   Set⟨Elem⟩ dequeueSet

procedure MGCUnboundedQueue()
   Elem e
begin
1: while true do
2:    e := havocQueueElem()
3:    nondet choice
4:          call Enqueue(e)
5:       or call Dequeue(e)
6:    end choice
7: end while
8: return()
end procedure

procedure Enqueue(Elem e)
   Addr n
begin
9:  lock(queueLock)
10: newnode := malloc(e)
11: newnode→next := null
12: tail→next := newnode
    enqueueSet := enqueueSet ∪ {e}
13: tail := newnode
14: unlock(queueLock)
15: return()
end procedure

procedure Dequeue()
   Elem result
begin
16: lock(queueLock)
17: if head→next = null then
18:    unlock(queueLock)
19:    raise(EmptyException)
20: end if
21: result := head→next→data
22: head := head→next
    dequeueSet := dequeueSet∪{result}
23: unlock(queueLock)
24: return(result)
end procedure
```

Figure 10.3: Modifications needed to apply over the lock-based unbounded queue implementation in order to verify unbQueueInc property.

As can be seen in Fig. 10.3, Enqueue and Dequeue are modified so that ghost variables $enqueueSet$ and $dequeueSet$ are updated at their linearization points. In particular, as elements stored in $head$ and $null$ are sentinel, property unbQueueInc can be expressed as:

$$\text{unbQueueInc} \stackrel{\text{def}}{=} \square\ (enqueueSet = set2elemset(heap, reg - \{head, null\}) \cup dequeueSet)$$

Table 10.3 presents the results we have obtained using Leap for verifying the properties stated above. As can be seen, all properties were verified within less than a second.

	form. info		#solved VC		single VC time(s.)		TL3 DP	Leap
	id	#VC	pos	TL3	slowest	average	time(s)	time(s)
unbQueuePres	0	23	18	5	0.08	0.01	0.18	0.00
unbQueueNext	1	45	35	10	0.01	0.00	0.05	0.04
unbQueueLock	1	45	33	12	0.01	0.00	0.05	0.02
unbQueueInc	0	23	19	4	0.09	0.01	0.22	0.01

Table 10.3: Verification conditions (VCs) proved and executing time using TL3 decision procedure for verifying safety properties of an unbounded lock-based queue.

10.2.3 Lock-free Stacks

We now prove some parametrized invariants for an implementation of a lock-free stack. The data type was already presented in Section 6.1.3. A basic property we would like to verify is that all operations performed by the lock-free stack preserve the shape of the data structure. This property is expressed as **lfStackPres**, which simply states that $null$ and the node pointed by top are reachable within the region of the stack:

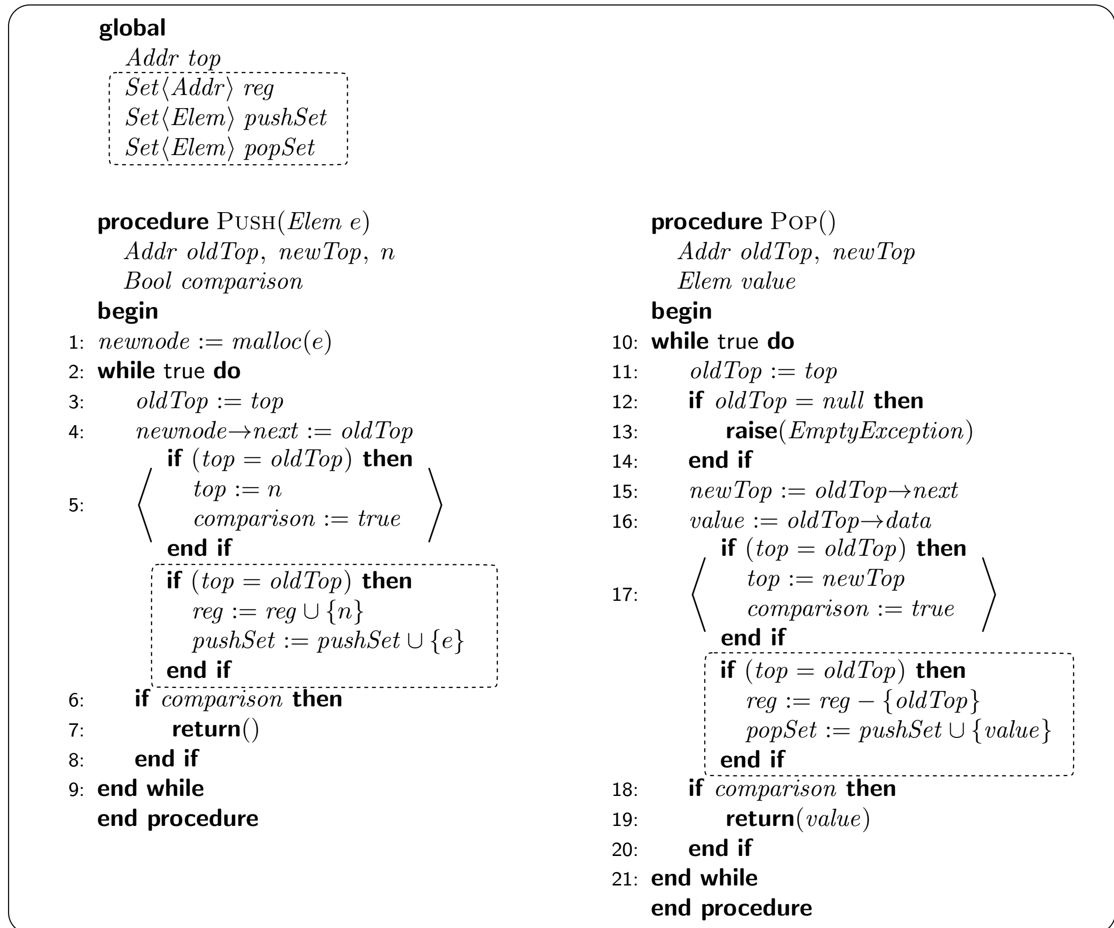
$$\text{lfStackPres} \stackrel{\text{def}}{=} \square\ (null \in reg\ \wedge top \in reg\ \wedge reg = addr2set(heap, top))$$

As in previous examples, in this case we also require some extra auxiliary invariants in order to prove **lfStackPres**. In particular:

lfStackNext: which specifies the relation between the pointers as the stack is traversed.

lfStackRegion: which declares how the region of the heap that corresponds to the stack is modified.

lfStackVals: which states some relation between the values stored in the stack.



```
global
  Addr top
  Set⟨Addr⟩ reg
  Set⟨Elem⟩ pushSet
  Set⟨Elem⟩ popSet

procedure PUSH(Elem e)
  Addr oldTop, newTop, n
  Bool comparison
begin
1:  newnode := malloc(e)
2:  while true do
3:    oldTop := top
4:    newnode→next := oldTop
      ⟨ if (top = oldTop) then
5:        top := n
          comparison := true
        end if ⟩
      if (top = oldTop) then
        reg := reg ∪ {n}
        pushSet := pushSet ∪ {e}
      end if
6:    if comparison then
7:      return()
8:    end if
9:  end while
end procedure

procedure POP()
  Addr oldTop, newTop
  Elem value
begin
10: while true do
11:   oldTop := top
12:   if oldTop = null then
13:     raise(EmptyException)
14:   end if
15:   newTop := oldTop→next
16:   value := oldTop→data
      ⟨ if (top = oldTop) then
17:       top := newTop
          comparison := true
        end if ⟩
      if (top = oldTop) then
        reg := reg − {oldTop}
        popSet := pushSet ∪ {value}
      end if
18:   if comparison then
19:     return(value)
20:   end if
21: end while
end procedure
```

Figure 10.4: Modifications needed to apply over the lock-free stack implementation in order to verify lfStackInc.

	form. info		#solved VC		single VC time(s.)		TL3 DP	LEAP
	id	#VC	pos	TL3	slowest	average	time(s)	time(s)
IfStackPres	0	37	30	7	0.02	0.00	0.06	0.00
IfStackRegion	1	73	69	4	0.01	0.00	0.02	0.02
IfStackReach	1	109	90	19	1.44	0.04	3.99	0.42
IfStackNext	1	73	63	10	0.14	0.00	0.19	0.03
IfStackInc	0	37	30	7	0.01	0.00	0.04	0.01
IfStackDisj	2	109	105	4	0.01	0.00	0.08	0.11
IfStackVals	1	73	62	11	0.01	0.00	0.05	0.01

Table 10.4: Verification conditions (VCs) proved and executing time using TL3 decision procedure for verifying safety properties of a lock-free stack.

IfStackReach: which deals with the relation between reachable nodes within the stack.

IfStackDisj: which states that nodes created by different threads must not intersect in the heap.

In addition to the invariants stated above, we can prove properties such as **IfQueueInc** which states that all elements that has been inserted into the stack using a call to PUSH are still in the stack unless a POP operation has removed them from the data structure:

$$\text{IfStackInc} \stackrel{\text{def}}{=} \square\ (pushSet = (set2elemset(heap, reg - \{null\})) \cup popSet)$$

In order to verify this property, we add some ghost annotations to the original lock-free stack implementation. Fig. 10.4 shows the lock-free stack implementation presented in Section 6.1.3 including the new required modifications that enables the verification of property **IfQueueInc**.

In Fig. 10.4 we omit the specification of the most general client MGC, but its construction is very similar to the MGC for unbounded queues presented in Fig. 10.3. Additionally, line 5 and line 17 define the expanded semantics for the CAS operations.

As with previous examples, Table 10.4 presents the results from using LEAP for the verification of the safety properties stated above.

10.2.4 Lock-free Queues

We now check some invariants of lock-free non-blocking queues, presented in Section 6.1.4, also known as Michael-Scott queue.

Once again, one of the main properties we would like to verify is queue shape preservation, which is expressed by **IfQueuePres**:

$$\text{IfQueuePres} \stackrel{\text{def}}{=} \square \begin{pmatrix} null \in reg & \wedge \\ tail \in reg & \wedge \\ tail \neq null & \wedge \\ head \neq null & \wedge \\ reg = addr2set(heap, head) & \end{pmatrix}$$

```
global
  Addr head, tail
  Set⟨Addr⟩ reg
  Set⟨Elem⟩ enqueueSet
  Set⟨Elem⟩ dequeueSet

procedure ENQUEUE(Elem e)
  Addr last, nextptr, n
  Bool comparison = false
begin
1:  newnode := malloc(e)
2:  newnode→next := null
3:  while true do
4:    last := tail
5:    nextptr := last→next
6:    if last = tail then
7:        ⟨ if (last→next = nextptr) then
              last→next := n
              comparison := true
            end if ⟩
          if (last→next = nextptr) then
              reg := reg ∪ {n}
              enqueueSet := enqueueSet ∪ {e}
          end if
8:      if comparison then
9:          ⟨ if tail = last then
                tail := n
              end if ⟩
10:         return()
11:     else
12:         ⟨ if tail = last then
                tail := nextptr
              end if ⟩
13:     end if
14:   end if
15: end while
16: return()
end procedure

procedure DEQUEUE()
  Addr first, last, nextptr
  Elem value
  Bool comparison = false
begin
17: while true do
18:   first := head
19:   last := tail
20:   nextptr := first→next
21:   if first = head then
22:     if first = last then
23:       if nextptr = null then
24:         raise(EmptyException)
25:       end if
26:       ⟨ if (tail = last) then
              tail := nextptr
            end if ⟩
27:     else
28:       value := nextptr→data
29:       ⟨ if (head = first) then
              head := nextptr
              comparison := true
            end if ⟩
          if (head = first) then
            reg := reg − {first}
            dequeueSet := dequeueSet − {value}
          end if
30:       if comparison then
31:         return(value)
32:       end if
33:     end if
34:   end if
35: end while
end procedure
```

Figure 10.5: Modifications needed to apply over the lock-free queue implementation in order to verify **lfQueueInc**.

In this case, as with other examples, we also require some auxiliary invariants:

lfQueueNext: which keeps track of the pointer assignments as a thread traverses the queue.

lfQueueRegion: which describes the structure of the region of the heap used by the queue.

lfQueueDisj: which states that new nodes created by different threads must not be allocated at equal addresses within the heap.

lfQueueComp: which deals with the values taken by ghost program variable $comparison$.

Contrary to the lock-based queue example analyzed before, now we do not require an invariant describing the state of the locks because the algorithm is lock-free. Instead, we define an invariant **lfQueueReach** to describe the reachability between different nodes within the lock-free queue.

	form. info		#solved VC		single VC time(s.)		TL3 DP	LEAP
	id	#VC	pos	TL3	slowest	average	time(s)	time(s)
lfQueuePres	1	52	40	12	0.12	0.01	0.40	0.02
lfQueueRegion	1	103	99	4	0.01	0.00	0.04	0.03
lfQueueReach	1	103	81	22	11.19	0.23	23.78	0.15
lfQueueNext	1	103	76	27	5.20	0.07	7.76	0.44
lfQueueInc	0	52	40	12	0.10	0.00	0.19	0.01
lfQueueComp	1	103	102	1	0.00	0.00	0.01	0.02
lfQueueDisj	2	154	150	4	0.03	0.00	0.13	0.17

Table 10.5: Verification conditions (VCs) proved and executing time using TL3 decision procedure for verifying safety properties of a lock-free queue.

Once again, all complete specifications for each of the properties described above can be seen in LEAP's website. As for the lock-based implementation, **lfQueueInc** is a safety specification that states that all elements inserted into the queue are still in the queue unless they have been removed through a call to DEQUEUE:

$$\text{lfQueueInc} \stackrel{\text{def}}{=} \square\ (enqueueSet = (set2elemset(heap, reg - \{head, null\}) \cup dequeueSet))$$

As before, in order to check property **lfQueueInc** we annotate the original lock-free stack program with ghost variables. Fig. 10.5 shows the implementation of the lock-free stack introduced in Section 6.1.4 with the additional ghost notation necessary to prove **lfQueueInc** property. Once again, we present the results we have obtained using LEAP in the verification of the safety specifications described above. The results are depicted in Table 10.5.

10.3 Verification of Safety Properties using $\mathsf{TSL_K}$

We now present the results of our experiments using LEAP for verifying skiplists with a bounded number of levels. We consider programs SEARCH presented in Fig. 7.3, INSERT shown in Fig. 7.5 and REMOVE introduced in Fig. 7.7 from Chapter 7. In our experiments, we use a slightly modified version of these procedures, so that they do not use locks.

We study bounded skiplist implementations of 1, 2, 3, 4 ans 5 levels. Each implementation has K instantiated to the appropriated number of levels. For each implementation, the main property we would like to verify is skiplist shape preservation. In the case of skiplists with 2 levels, this property is described through the temporal specification **skiplist$_2$** which states:

$$\text{skiplist}_2 \stackrel{\text{def}}{=} \square \left(\begin{array}{ll} tail \neq null\ \wedge head \neq null\ \wedge head \neq tail & \wedge \\ ordList(heap, head, tail)\ \wedge reg = addr2set_{\mathsf{K}}(heap, head, 0) & \wedge \\ heap[tail].next[0] = null\ \wedge heap[tail].next[1] = null & \wedge \\ addr2set_{\mathsf{K}}(heap, head, 1) \subseteq addr2set_{\mathsf{K}}(heap, head, 0) & \wedge \\ heap[head].data = -\infty\ \wedge heap[tail].data = +\infty & \wedge \\ tail \in addr2set_{\mathsf{K}}(heap, head, 0)\ \wedge tailinaddr2set_{\mathsf{K}}(heap, head, 1) & \end{array} \right)$$

	form. info		#solved VC		single VC time(s.)		$\mathsf{TSL_K}$ DP	LEAP
	id	#VC	pos	$\mathsf{TSL_K}$	slowest	average	time(s)	time(s)
$\mathsf{skiplist_1}$	0	77	70	7	0.10	0.01	0.20	0.32
$\mathsf{region_1}$	1	77	50	27	0.14	0.01	0.37	0.28
$\mathsf{next_1}$	1	77	58	19	0.02	0.01	0.15	0.14
$\mathsf{order_1}$	1	77	52	25	0.02	0.01	0.58	0.11
$\mathsf{value_1}$	1	77	70	7	0.01	0.01	0.03	0.07
$\mathsf{skiplist_2}$	0	79	72	7	1.51	0.04	3.33	0.31
$\mathsf{region_2}$	1	79	52	27	0.33	0.01	1.12	0.49
$\mathsf{next_2}$	1	79	60	19	0.04	0.01	0.24	0.14
$\mathsf{order_2}$	1	79	54	25	0.56	0.01	0.86	1.01
$\mathsf{value_2}$	1	79	72	7	0.01	0.01	0.03	0.11
$\mathsf{skiplist_3}$	0	81	74	7	776.45	15.27	1,221.52	0.45
$\mathsf{region_3}$	1	81	54	27	17.36	0.34	26.92	0.58
$\mathsf{next_3}$	1	81	63	19	0.09	0.01	0.47	0.20
$\mathsf{order_3}$	1	81	56	25	7.80	0.10	8.35	1.31
$\mathsf{value_3}$	1	81	74	7	0.01	0.01	0.03	0.10
$\mathsf{skiplist_4}$	0	83	76	7	**T.O.**	**T.O.**	**T.O.**	0.80
$\mathsf{region_4}$	1	83	56	27	226.08	4.30	348.44	0.79
$\mathsf{next_4}$	1	83	65	19	0.22	0.01	0.83	0.25
$\mathsf{order_4}$	1	83	58	25	43.97	0.56	45.28	1.83
$\mathsf{value_4}$	1	83	76	7	0.01	0.01	0.03	0.12
$\mathsf{skiplist_5}$	0	85	78	7	**T.O.**	**T.O.**	**T.O.**	0.89
$\mathsf{region_5}$	1	85	58	27	385.16	5.64	462.45	0.98
$\mathsf{next_5}$	1	85	67	19	0.24	0.01	1.08	0.31
$\mathsf{order_5}$	1	85	60	25	188.09	2.32	190.03	3.01
$\mathsf{value_5}$	1	85	78	7	0.01	0.01	0.03	0.11

Table 10.6: Verification conditions (VCs) proved and executing time using $\mathsf{TSL_K}$ decision procedure for verifying safety properties for bounded skiplists of 1, 2, 3, 4 and 5 levels.

The formula $\mathsf{skiplist_2}$ describes the heap shape of a skiplist with 2 levels. In order to verify $\mathsf{skiplist_2}$ as invariant, we use auxiliary invariants. These auxiliary properties, in the case of a skiplist with 2 levels, are:

$\mathsf{next_2}$: which states the relation between pointers $head$, $tail$, $prev$, $curr$ and aux as a thread traverses the skiplist.

$\mathsf{region_2}$: which keeps track of the composition of the region in the heap where the skiplist is stored.

$\mathsf{order_2}$: which describes the order relation between the skiplist nodes.

$\mathsf{value_2}$: which controls the relation between the skiplist levels.

As before, the detailed specifications for each of these properties in the case of a skiplist of 2 levels can be found in LEAP's website.

Similarly, we have also considered skiplist implementations of 1, 3, 4 and 5 levels. For each of these implementations, we have analyzed a set of invariant candidates similar to the ones described above. Table 10.6 shows the results we have obtained.

As shown in Table 10.6, the decision procedure for $\mathsf{TSL_K}$ performs reasonably well. However, as the number of levels of a skiplist grows, the decision procedure starts to degrade its performance.

Our empirical evaluation suggests that $\mathsf{TSL_K}$ would not scale to any number of levels. In the next section we present the results we have obtained using TSL, which shows that TSL is a natural candidate for the verification of arbitrary high skiplists.

10.4 Verification of Safety Properties using TSL

We now present experimental results using the decision procedure for skiplists of arbitrary height described in Chapter 8. As TSL currently does not support concurrency, we verify these two sequential implementation of a skiplist:

1. The skiplist implementation presented in Chapter 8; and
2. A skiplist implementation which is part of the KDE library [197].

In the case of the skiplist presented in Chapter 8, the operations we analyze are SEARCH, INSERT and REMOVE, described in Fig. 8.3, Fig. 8.4 and Fig. 8.5 respectively. The first property analyzed is skiplist shape preservation, which is described as follows:

$$\mathsf{skiplist} \stackrel{\text{def}}{=} \Box \begin{pmatrix} tail \neq null \ \wedge head \neq null \ \wedge head \neq tail & \wedge \\ skiplist(heap, reg, max, head, tail, elems) \ \wedge max \geq 0 & \wedge \\ heap[head].data = -\infty \ \wedge heap[tail].data = +\infty & \end{pmatrix}$$

We use the following auxiliary invariants:

next: which keeps track of the state of the pointers used to traverse the skiplist.

region: which specifies the structure of the heap region where the skiplist is allocated.

order: which keeps the order relation between the nodes that are part of the skiplist.

value: which declares the special conditions that need to be considered when the inserted or removed element is present or not in the skiplist.

levels: which states the relation between every skiplist level and the integer variables that are used to index them.

We also verify some functional properties. For functional verification we use the simple specifications shown in Fig. 10.6. Fig. 10.6 presents three programs named FSEARCH, FINSERT and FREMOVE. These programs require the declaration of a new ghost global variable, named $elemsBefore$. The idea is that $elemsBefore$ keeps a copy of the elements stored in the skiplist just before a call to an especial element is performed. After a call to SEARCH, INSERT or REMOVE, it is possible to check whether the set of elements that are part of the skiplist has been correctly updated.

Using the programs described in Fig. 10.6 it is possible to verify the following properties:

funSearch: which states that a call to SEARCH does not modify the set of elements stored in the skiplist and that the value returned by the call corresponds to the presence of the element in the skiplist. This is specified by the following temporal formula:

$$\mathsf{funSearch}(i) \stackrel{\text{def}}{=} \Box \left(pc_{\text{FSEARCH}}(i) = 4 \rightarrow \begin{pmatrix} \text{FSEARCH}::result \leftrightarrow \text{FSEARCH}::e \in elems \ \wedge \\ elemsBefore = elems \end{pmatrix} \right)$$

```
global
  Addr head
  Addr tail
  Int maxLevel
  Set⟨Addr⟩ reg
  Set⟨Elem⟩ elems
  Set⟨Elem⟩ elemsBefore

procedure FSEARCH()
begin
1: Set⟨Elem⟩ elemsBefore := elems
2: v := havocSkiplistElem()
3: result := call SEARCH(v)
4: return()
end procedure

procedure FINSERT()
begin
1: Set⟨Elem⟩ elemsBefore := elems
2: v := havocSkiplistElem()
3: result := call INSERT(v)
4: return()
end procedure

procedure FREMOVE()
begin
1: Set⟨Elem⟩ elemsBefore := elems
2: v := havocSkiplistElem()
3: result := call REMOVE(v)
4: return()
end procedure
```

Figure 10.6: Functional specifications for SEARCH, INSERT and REMOVE

funInsert: which describes the fact that after a call to INSERT, the new element should be in the skiplist. This can be expressed by the following temporal specification:

$$\mathsf{funInsert}(i) \overset{\text{def}}{=} \square \left(pc_{\text{FINSERT}}(i) = 4 \rightarrow \mathit{elems} = \mathit{elemsBefore} \cup \{v\} \right)$$

funRemove: which specifies that after a call to REMOVE with a certain element, such element should not be part of the skiplist anymore. This property is described using the following temporal formula:

$$\mathsf{funRemove}(i) \overset{\text{def}}{=} \square \left(pc_{\text{FREMOVE}}(i) = 4 \rightarrow \mathit{elems} = \mathit{elemsBefore} - \{v\} \right)$$

Additionally, we verify skiplist shape preservation of an unbounded skiplist implementation

	form. info		#Calls to DPs						
	#VC	#PO	pos	TSL	TSL_1	TSL_2	TSL_3	TSL_4	TSL_5
skiplist	80	560	70	10	54	89	44	10	–
region	80	1583	45	35	113	176	76	–	–
next	80	1899	55	25	30	41	22	–	–
order	80	2531	47	33	160	250	116	–	–
$\mathsf{skiplist}_{\mathsf{KDE}}$	54	214	47	7	38	72	39	10	–
$\mathsf{region}_{\mathsf{KDE}}$	54	585	37	17	99	169	76	–	–
$\mathsf{next}_{\mathsf{KDE}}$	54	1115	40	14	42	46	16	–	–
$\mathsf{order}_{\mathsf{KDE}}$	54	797	35	19	107	158	76	–	–
funSearch	76	76	74	2	24	–	–	–	–
funInsert	75	75	68	7	9	2	–	–	–
funRemove	75	75	67	8	15	2	–	–	–

Table 10.7: Number of queries for the verification of unbounded skiplists. "–" means no calls to the decision procedure were required. All TSL queries were ultimately decomposed into TSL_K for $\mathsf{K} \leq 5$.

which is part of the KDE library [197]. This implementation is similar to the one presented in Section 8.1. In order to verify the skiplist shape preservation we need to define slightly modified invariant candidates named **skiplist$_{\mathsf{KDE}}$**, **next$_{\mathsf{KDE}}$**, **region$_{\mathsf{KDE}}$** and **order$_{\mathsf{KDE}}$** which describe properties similar to **skiplist**, **next**, **region** and **order** respectively.

In order to verify these properties we use the decision procedure for TSL introduced in Chapter 8. This decision procedure reduces a TSL satisfiability problem into various queries to a Presburguer arithmetic decision procedure and TSL_K decision procedures. This is reflected in Table 10.7, which shows that a single call to the TSL decision procedure may lead to numerous calls to simpler decision procedures from the TSL_K family. Table 10.7 shows, for each analyzed invariant candidate:

1. #VC: the number of generated verification conditions.

2. #PO: the number of generated proof obligations as a consequence of applying different tactics. We call a proof obligation to each of the implications obtained as a result of applying the tactics presented in Section 9.2.4.

3. pos: the number of verification conditions solved using the simple decision procedure that reasons on program locations.

4. TSL: the number of verification conditions solved using TSL decision procedure.

5. TSL_1, TSL_2, TSL_3, TSL_4 and TSL_5: the number of calls to each decision procedure from the TSL_K family, exercised by the TSL decision procedure.

Finally, Table 10.8 shows the execution time for verifying each of the safety properties specified for unbounded skiplists. The columns of Table 10.8 describe the slowest and average verification time for a single verification condition; the total time required for all decision procedures to check the validity of all generated verification conditions; and the total time required by LEAP to generate all verification conditions. All verification conditions are checked as valid.

	VC time (s.)		Total time (s.)	
	slowest	average	DP	LEAP
skiplist	18.41	0.94	75.58	0.15
region	20.82	0.74	59.92	0.58
next	3.27	0.07	5.95	1.21
order	1.89	0.07	6.28	2.12
skiplist$_{\mathsf{KDE}}$	18.21	0.52	28.16	0.05
region$_{\mathsf{KDE}}$	26.22	0.66	36.11	0.21
next$_{\mathsf{KDE}}$	13.18	0.25	13.68	0.76
order$_{\mathsf{KDE}}$	1.12	0.06	3.32	0.59
funSearch	0.12	0.01	0.15	0.04
funInsert	0.02	0.01	0.05	0.04
funRemove	0.04	0.01	0.10	0.06

Table 10.8: Running times for the verification of safety properties over unbounded skiplists.

10.5 Verification of Liveness Properties using PVD

In this section we describe some results we have obtained using the parametrized verification diagrams introduced in Chapter 4. We use PVD to verify some liveness properties of parametrized systems. In particular, we study 2 concrete examples:

- the mutual exclusion protocol introduced in Example 2.2 from Chapter 2; and
- the concurrent lock-coupling lists described as motivating example in Chapter 6.

10.5.1 Mutual Exclusion Protocol

We consider the mutual exclusion protocol SETMUTEX described in Example 2.2 in page 21. In particular, for the verification of liveness properties, we consider the modified version presented in Fig. 4.5 at Example 4.3 in page 83. For this program, we would like to verify that any thread that wants to access the critical section, it eventually succeeds in accessing. This simple property can be expressed by the following 1-index parametrized temporal formula:

$$\text{eventually_critical}(k) \quad \stackrel{\text{def}}{=} \quad \Box\,(pc(k) = 3 \rightarrow \Diamond pc(k) = 5)$$

In order to check this temporal property we use the parametrized verification diagram described in Example 4.3. For this particular example, we will need to define some auxiliary invariants that will act as support. These invariants are:

minticketActive: this invariant states that a thread with the minimum ticket in the bag is at some point between lines 4 and 6:

$$\text{minticketActive}(i) \quad \stackrel{\text{def}}{=} \quad \Box\ (i = \pi_{tid}(minPair(bag))\ \wedge pc(i) = 4..6)$$

bagIsActive: this invariant is more general, and describes the fact that a pair is in the set bag if and only if the program execution that corresponds to the thread identifier associated to this pair is at line 4, 5 or 6. For specifying this property, we use predicate $inTidPair$. This predicate, which receives a thread identifier i and a set of pairs bag, checks whether there is an integer n such that the pair (i, n) is in the set bag. We can then describe property bagIsActive as:

$$\text{bagIsActive}(i) \quad \stackrel{\text{def}}{=} \quad \Box\,(inTidPair(i, bag) \leftrightarrow pc(i) = 4..6)$$

unique: this invariant specifies that there cannot be two pairs within bag with the same thread identifier or integer component. To do so, we use the predicates $uniqueInt$ and $uniqueTid$ from our theory of pairs which receive a set of pairs as argument and hold whenever there are no repeated integers or repeated thread identifier components in such set. Specification unique is:

$$\text{unique} \quad \stackrel{\text{def}}{=} \quad \Box\,(uniqueTid(bag)\ \wedge uniqueInt(bag))$$

We use LEAP to check that these auxiliary specifications are in fact invariants of program SETMUTEX.

The PVD that represents the proof that program SETMUTEX satisfies the liveness property eventually_critical(k) for an arbitrary thread k, is depicted in Example 4.3. Fig. 10.7 shows the PVD declaration in LEAP input format. Using this PVD, we use LEAP to automatically verify the condition (Init) —for initiation—, conditions (SelfConsec) and (OtherConsec) —for consecution—, conditions (SelfAcc) and (OtherAcc) —for acceptance— and conditions (En) and (Succ) —for

```
Diagram[ticketset]

Nodes: n0 { @1(k). \/ @2(k). },
       n1 { @3(k). },
       n2 { @4(t). /\ k!=t /\ @4(k). /\
            int_of(spmin(bag)) = main::ticket(t) },
       n3 { @5(t). /\ k!=t /\ @4(k). /\
            int_of(spmin(bag)) = main::ticket(t) },
       n4 { @6(t). /\ k!=t /\ @4(k). /\
            int_of(spmin(bag)) = main::ticket(t) },
       n5 { @4(k). /\ int_of(spmin(bag)) = main::ticket(k) },
       n6 { (@5(k). \/ @6(k).) /\ int_of(spmin(bag)) = main::ticket(k) }

Boxes: {b1[t]:n2,n3,n4}

Initial: n0

Edges:  n0 --> n1;
        n1 -{3(k)}-> n2;
        n1 -{3(k)}-> n3;
        n1 -{3(k)}-> n4;
        n1 -{3(k)}-> n5;
       [n2 -{4(t)}-> n3];
       [n3 -{5(t)}-> n4];
        n4 -{6(t)}-> n2;
        n4 -{6(t)}-> n3;
        n4 -{6(t)}-> n4;
        n4 -{6(t)}-> n5;
        n5 -{4(k)}-> n6;
        n6 --> n0;
  // Self-loops
        n0 --> n0;
        n1 --> n1;
       [n2 --> n2];
       [n3 --> n3];
       [n4 --> n4];
        n5 --> n5;
        n6 --> n6;

Acceptance:
  <<Bad :{(n4,n2,any),(n4,n3,any),(n4,n4,any),(n4,n5,any)};
    Good:{(n6,n0,any),
          (n1,n1,any),(n1,n2,any),(n1,n3,any),(n1,n4,any),(n1,n5,any),
          (n0,n0,any)};
                [(splower(bag,main::ticket(k)), pairsubset_op)] >>
```

Figure 10.7: LEAP input representation for the PVD showing that SETMUTEX satisfies property eventually_critical (k).

```
MODULE main
VAR
  state : {q0,q1,q2,q3,q4,q5,q6,g0,g1,g2,g3,g4,g5,g6,
           b0,b1,b2,b3,b4,b5,b6};
  k : 1..6;

INIT state = q0;

TRANS
  case
    state = q6 : next(state) in {g0};
    state = g6 : next(state) in {g0};
    state = q0 : next(state) in {q0,q1};
    state = q1 : next(state) in {q1,q2,q3,q4,q5};
    state = q2 : next(state) in {q2,q3};
    state = q3 : next(state) in {q3,q4};
    state = q4 : next(state) in {q2,q3,q4,q5};
    state = q5 : next(state) in {q5,q6};
    state = q6 : next(state) in {q6};
    state = g0 : next(state) in {q0,q1};
    state = g1 : next(state) in {q1,q2,q3,q4,q5};
    state = g2 : next(state) in {q2,q3};
    state = g3 : next(state) in {q3,q4};
    state = g4 : next(state) in {q2,q3,q4,q5};
    state = g5 : next(state) in {q5,q6};
    state = g6 : next(state) in {q6};
    state = b0 : next(state) in {b0,b1};
    state = b1 : next(state) in {b1,b2,b3,b4,b5};
    state = b2 : next(state) in {b2,b3};
    state = b3 : next(state) in {b3,b4};
    state = b4 : FALSE;
    state = b5 : next(state) in {b5,b6};
    state = b6 : next(state) in {b0,b6};
    state = q0 : next(state) in {b0,b1};
    state = q1 : next(state) in {b1,b2,b3,b4,b5};
    state = q2 : next(state) in {b2,b3};
    state = q3 : next(state) in {b3,b4};
    state = q4 : next(state) in {b2,b3,b4,b5};
    state = q5 : next(state) in {b5,b6};
    state = q6 : next(state) in {b0,b6};
  esac;

TRANS
  case
    state in {q0,g0,b0} : k != 3 & k != 5 & k != 6;
    state in {q1,g1,b1} : k = 3;
    state in {q2,g2,b2} : TRUE;
    state in {q3,g3,b3} : TRUE;
    state in {q4,g4,b4} : TRUE;
    state in {q5,g5,b5} : TRUE;
    state in {q6,g6,b6} : k = 5 | k = 6;
  esac;

  FAIRNESS (state in {g0,g1,g2,g3,g4,g5,g6,b0,b1,b2,b3,b4,b5,b6});
  LTLSPEC G (k = 3 -> F (k = 5 | k = 6));
```

Figure 10.8: NuSMV input representation for checking condition (ModelCheck) for the PVD that shows that program SETMUTEX satisfies property eventually_critical(k).

	#VC	#solved VC		single VC time(s.)		DP	LEAP
		pos	num	slowest	average	time(s)	time(s)
Initiation	1	0	1	0.01	0.01	0.01	0.01
Consecution	153	144	9	2.66	0.03	4.22	0.06
Acceptance	195	132	63	1.46	0.08	15.28	0.05
Fairness	24	20	4	0.03	0.01	0.10	0.02

Table 10.9: Running times for the verification of property eventually_critical(k) in program SETMUTEX.

fairness—. Table 10.9 presents the results we have obtained for the verification of property eventually_critical(k) for the mutual exclusion protocol SETMUTEX. The table shows, for each rule, the total number of generated verification conditions, the number of verification conditions solved using the location based decision procedures and the total number of verification conditions checked using a decision procedure for Presburguer arithmetic. The table also presents the slowest and average time for solving a single verification condition, the total time that takes the decision procedures to check all generated verification conditions and the time it takes LEAP to generate all these verification conditions.

It is easy to see that condition (Prop) holds. For condition (ModelCheck), we check whether the propositional models of the diagram are included in traces of property eventually_critical(k). To do so, we convert the edge-Streett non-deterministic automaton described by the PVD into a non-deterministic Büchi automaton (NBW) using the translation described in Appendix A. Then, we use the NuSMV [43] model-checker to test the resulting NBW against the temporal formula eventually_critical(k). Fig. 10.8 shows the resulting NBW used to check (ModelCheck) using NuSMV. This verification is performed in less than a second. As all verification conditions are checked, we can conclude that program SETMUTEX satisfies property eventually_critical(k), and that our PVD is a witness of such proof.

10.5.2 Lock-coupling Lists

We now use parametrized verification diagrams to verify a liveness property of the concurrent lock-coupling single liked lists presented in Section 6.1.1. Consider a parametrized system composed by an unbounded number of threads, each of which is executing the procedure MGCLIST, which was described in Fig. 6.4 in page 110. A liveness property we would like to verify is that if an arbitrary thread k in this system attempts to insert an element in the concurrent lock-coupling list, then thread k eventually succeeds.

Before we proceed with the verification of this property, we present a variation of the implementation of concurrent lock-coupling single-linked lists first introduced in Section 6.1. Fig. 10.9 present this new implementation, which basically extends the original implementation by adding some extra ghost code. In particular, this new ghost code will help us in computing the ranking functions needed for checking the acceptance conditions of the PVD we will construct later. We add 4 new global ghost variables of type set of thread identifiers:

- $lockedSet$ contains the set of thread identifiers that owns at least a lock in the list. A thread identifier is added to this set when the thread gets its first lock (which can happen at line 10,

```
global
   Addr head
   Addr tail
   Set⟨Addr⟩ reg
   Set⟨Elem⟩ elems
   Set⟨Tid⟩ lockedSet
   Set⟨Tid⟩ lockedInsert
   Set⟨Tid⟩ aheadSet
   Set⟨Tid⟩ aheadInsert

procedure MGCList()
   Elem e
begin
1:  while true do
2:     e := havocListElem()
3:     nondet choice
4:           call Search(e)
5:        or call Insert(e)
6:        or call Remove(e)
7:     end choice
8:  end while
end procedure

procedure Insert(Elem e)
   Addr  prev, curr, aux, newnode
begin
24: prev := head
25: lock(prev→lock)
    lockedSet := lockedSet ∪ {me}
    lockedInsert := lockedInsert ∪ {me}
    if (me = k) then
       aheadSet := lockedSet
       aheadInsert := lockedInsert
    endif
26: curr := prev→next
27: lock(curr→lock)
28: while curr ≠ null ∧ curr→data < e do
29:    aux := prev
30:    prev := curr
31:    unlock(aux→lock)
32:    curr := curr→next
33:    lock(curr→lock)
34: end while
35: if curr ≠ null ∧ curr→data > e then
36:    newnode := malloc(e)
37:    newnode→next := curr
38:    prev→next := newnode
       reg := reg ∪ {newnode}
       elems := elems ∪ {e}
       lockedInsert := lockedInsert − {me}
       aheadInsert := aheadInsert − {me}
39: else
40:    skip
       lockedInsert := lockedInsert − {me}
       aheadInsert := aheadInsert − {me}
41: end if
42: unlock(prev→lock)
43: unlock(curr→lock)
    lockedSet := lockedSet − {me}
    aheadSet := aheadSet − {me}
44: return
end procedure

procedure Search(Elem e)
   Addr prev, curr, aux
   Bool found
begin
9:  prev := head
10: lock(prev→lock)
    lockedSet := lockedSet ∪ {me}
11: curr := prev→next
12: lock(curr→lock)
13: while curr→data < e do
14:    aux := prev
15:    prev := curr
16:    unlock(aux→lock)
17:    curr := curr→next
18:    lock(curr→lock)
19: end while
20: found := (curr→data = e)
21: unlock(prev→lock)
22: unlock(curr→lock)
    lockedSet := lockedSet − {me}
    aheadSet := aheadSet − {me}
23: return found
end procedure

procedure Remove(Elem e)
   Addr  prev, curr, aux
begin
45: prev := head
46: lock(prev→lock)
    lockedSet := lockedSet ∪ {me}
47: curr := prev→next
48: lock(curr→lock)
49: while curr ≠ tail ∧ curr→data < e do
50:    aux := prev
51:    prev := curr
52:    unlock(aux→lock)
53:    curr := curr→next
54:    lock(curr→lock)
55: end while
56: if curr ≠ tail ∧ curr→data = e then
57:    aux := curr→next
58:    prev→next := aux
       reg := reg − {curr}
       elems := elems − {e}
59: end if
60: unlock(prev→lock)
61: unlock(curr→lock)
    lockedSet := lockedSet − {me}
    aheadSet := aheadSet − {me}
62: return
end procedure
```

Figure 10.9: Concurrent lock-coupling single-linked lists implementation with ghost code for the verification of property eventually_insert(k).

25 or 46). Then, the thread identifier is removed from $lockedSet$ when the thread releases its last lock (which can happen at line 22, 43 or 61).

- $lockedInsert$ keeps the set of thread identifiers that own a lock and are executing procedure INSERT but have not reached INSERT's linearization point yet. A thread identifier is added to set $lockedInsert$ when it executes line 25 and is removed from the set when executing line 38 (in case the element e is inserted) or line 40 (in case there is no need to insert element e because the element is already in the list).

- $aheadSet$ contains the set of thread identifiers that own at least one lock in the list and are ahead of thread k. Set $aheadSet$ is set when thread k executes line 25 of procedure INSERT. At this point, it holds that all threads that own a lock in the list are ahead of thread k, and hence we can assign $aheadSet$ to $lockedSet$. It is evident that at all time, $aheadSet$ is a subset of $lockedSet$.

- $aheadInsert$ is similar to $lockedInsert$ and keeps the set of thread identifiers that own a lock in the list, are ahead of thread k and are executing procedure INSERT but have not still reached INSERT's linearization point (what happens at line 38).

On the other hand, ghost variables reg and $elems$ are updated as in the implementation presented in Section 6.1.

As before, in order to check this property valid, we need to define some auxiliary invariants that act as support. These invariants specify properties such as "a thread executing procedure SEARCH, INSERT or REMOVE does not have any lock in the list apart from $prev$, $curr$ and occasionally aux", "if a thread is executing SEARCH, INSERT or REMOVE, then it owns at least a lock in the list" or "any thread executing INSERT that has not executed line 38 or 40 has a lock in $lockedInsert$". We do not give here the formal description of these auxiliary invariants, but they can be consulted at LEAP's website [2]. As usual, we use LEAP to verify that all these auxiliary specifications are system invariants.

The liveness property we verify is that if an arbitrary thread k wants to insert an element in the list using procedure INSERT, then it eventually succeeds. We can express this property through the 1-index temporal formula eventually_insert:

$$\mathsf{eventually_insert}(k) \quad \stackrel{\text{def}}{=} \quad \square\,(pc(k) = 26 \rightarrow \diamond pc(k) = 42..44)$$

That is, whenever thread k gets its first lock in an attempt to insert an element (line 26 of procedure INSERT) it eventually finishes the execution of the procedure (reaches the lines between 42 and 44 of INSERT).

Fig. 10.10 shows the parametrized verification diagram we require for proving that a system composed by an unbounded number of threads executing the program shown in Fig. 10.9 satisfy property eventually_insert(k). In the diagram, we use:

- sch_working(i), for $pc(i) = 11..22$. That is, thread i is executing procedure SEARCH and owns a lock due to the execution of this procedure.

- ins_working(i) for $pc(i) = 26..43$. That is, thread i owns a lock as a consequence of been executing procedure INSERT.

[2] `http://software.imdea.org/leap`

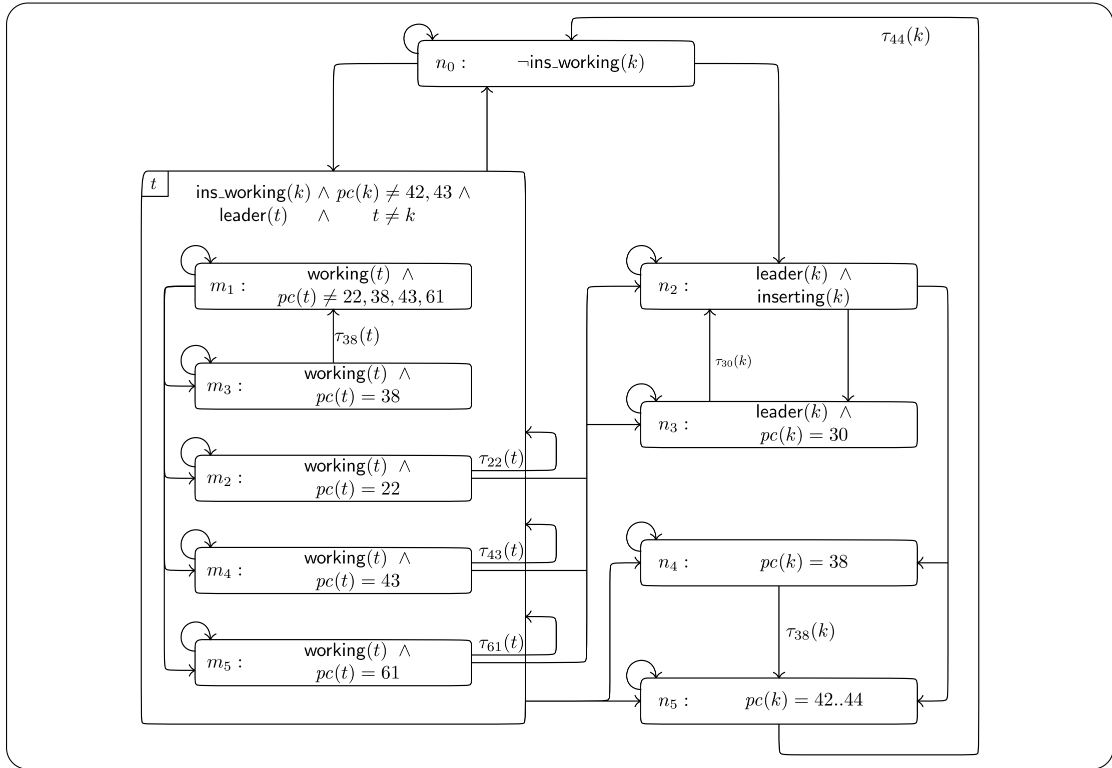


Figure 10.10: PVD for verifying that concurrent lock-coupling lists satisfy eventually_insert(k).

- rem_working(i) for $pc(i) = 47..61$. That is, thread i is executing procedure REMOVE and owns a lock because of the execution of this procedure.

- inserting(i) for $pc(i) = 26..29, 31..37, 39..42$. That is, thread i is executing procedure INSERT but is not at the program line in which pointer INSERT::$prev$ is about to advance or at the program line in which the node with the new element in connected to the rest of the list.

- working(i) for (sch_working(i) ∨ ins_working(i) ∨ rem_working(i)). That is, thread i owns a lock and is executing procedure SEARCH, INSERT or REMOVE.

We also use leader(i) for:

$$i = heap[lstlock(heap, getp(heap, head, null))].lockid$$

That is, for a thread i, leader(i) holds if and only if thread i owns the lock of a node in the list and there is no other locked node between such node and $tail$. We call "*the leader*" to the thread i that satisfies predicate leader(i), as it is the thread owning the lock closest to the tail of the list.

The diagram is formally defined by:

$$N \stackrel{\text{def}}{=} \{n_i \mid i = 0, 2, 3, 4, 5\} \cup \{m_j \mid j = 1, 2, 3, 4, 5\}$$
$$N_0 \stackrel{\text{def}}{=} \{n_0\}$$

$$
\begin{aligned}
E &\stackrel{\text{def}}{=} \{n_0 \to m_j \mid j = 1,2,3,4,5\} \cup \\
&\quad \{n_0 \to n_2, n_2 \to n_3, n_3 \to n_2, n_2 \to n_4, n_2 \to n_5, n_4 \to n_5, n_5 \to n_0\} \cup \\
&\quad \{m_i \to m_j \mid i = 2,4,5 \text{ and } j = 1,2,3,4,5\} \cup \{m_3 \to m_1\} \cup \\
&\quad \{m_j \to n_i \mid j = 2,4,5 \text{ and } i = 2,3\} \cup \{m_j \to n_i \mid j = 1,2,3,4,5 \text{ and } i = 4,5\} \cup \\
&\quad \{m_j \to m_j \mid j = 1,2,3,4,5\} \cup \{n_i \to n_i \mid i = 0,2,3,4,5\} \\
\textit{within} &\stackrel{\text{def}}{=} \{m_1 \to m_j \mid j = 1,2,3,4,5\} \cup \{m_3 \to m_1\} \\
\mathcal{B} &\stackrel{\text{def}}{=} \Big\{ \Big(\{m_1, m_2, m_3, m_4, m_5\}, t \Big) \Big\} \\
\mu(n_0) &\stackrel{\text{def}}{=} \neg\mathsf{ins_working}(k) \\
\mu(m_1) &\stackrel{\text{def}}{=} \mathsf{ins_working}(k) \wedge pc(k) \neq 42,43 \wedge \mathsf{leader}(t) \wedge t \neq k \wedge \mathsf{working}(t) \wedge pc(t) \neq 22,38,43,61 \\
\mu(m_2) &\stackrel{\text{def}}{=} \mathsf{ins_working}(k) \wedge pc(k) \neq 42,43 \wedge \mathsf{leader}(t) \wedge t \neq k \wedge \mathsf{working}(t) \wedge pc(t) = 22 \\
\mu(m_3) &\stackrel{\text{def}}{=} \mathsf{ins_working}(k) \wedge pc(k) \neq 42,43 \wedge \mathsf{leader}(t) \wedge t \neq k \wedge \mathsf{working}(t) \wedge pc(t) = 38 \\
\mu(m_4) &\stackrel{\text{def}}{=} \mathsf{ins_working}(k) \wedge pc(k) \neq 42,43 \wedge \mathsf{leader}(t) \wedge t \neq k \wedge \mathsf{working}(t) \wedge pc(t) = 43 \\
\mu(m_5) &\stackrel{\text{def}}{=} \mathsf{ins_working}(k) \wedge pc(k) \neq 42,43 \wedge \mathsf{leader}(t) \wedge t \neq k \wedge \mathsf{working}(t) \wedge pc(t) = 61 \\
\mu(n_2) &\stackrel{\text{def}}{=} \mathsf{leader}(k) \wedge pc(k) = 26..29, 31..37, 39..41 \\
\mu(n_3) &\stackrel{\text{def}}{=} \mathsf{leader}(k) \wedge pc(k) = 30 \\
\mu(n_4) &\stackrel{\text{def}}{=} \mathsf{leader}(k) \wedge pc(k) = 38 \\
\mu(n_5) &\stackrel{\text{def}}{=} \mathsf{leader}(k) \wedge pc(k) = 42..44 \\
\eta(e) &\stackrel{\text{def}}{=} \begin{cases}
(\tau_{38}, t) & \text{if } e \in \{m_3 \to m_1\} \\
(\tau_{22}, t) & \text{if } e \in \{m_2 \to m_j \mid j = 1,2,3,4,5\} \\
(\tau_{43}, t) & \text{if } e \in \{m_4 \to m_j \mid j = 1,2,3,4,5\} \\
(\tau_{61}, t) & \text{if } e \in \{m_4 \to m_j \mid j = 1,2,3,4,5\} \\
(\tau_{30}, k) & \text{if } e \in \{n_3 \to n_2\} \\
(\tau_{38}, k) & \text{if } e \in \{n_4 \to n_5\}
\end{cases} \\
\mathcal{F} &\stackrel{\text{def}}{=} \Big\langle \Big\langle \{m_i \to m_j \mid i = 2,4,5 \text{ and } j = 1,2,3,4,5\} \cup \\
&\qquad \{n_3 \to n_2, m_3 \to m_1\}\ , \\
&\qquad \{n_0 \to m_j \mid j = 1,2,3,4,5\} \cup \\
&\qquad \{n_0 \to n_0, n_0 \to n_2, n_2 \to n_4, n_2 \to n_5, n_4 \to n_5, n_5 \to n_5, n_5 \to n_0\} \cup \\
&\qquad \{m_j \to n_i \mid j = 1,2,3,4,5 \text{ and } i = 4,5\}\ , \\
&\qquad \lambda n \to \langle aheadSet, aheadInsert, addr2set(heap, \textsc{Insert}::prev(k)) \rangle \Big\rangle \Big\rangle \\
f(n) &\stackrel{\text{def}}{=} \begin{cases}
pc(k) = 26 & \text{if } n = m_1, m_2, m_3, m_4, m_5, n_2 \\
pc(k) = 42..44 & \text{if } n = n_5 \\
true & \text{otherwise}
\end{cases}
\end{aligned}
$$

The parametrized verification diagram described above represents the proof that an arbitrary thread k trying to insert an element eventually succeeds. The diagram consists of 10 nodes which are named n_i (with $i = 0, 2, 3, 4, 5$) and m_j (with $j = 1, 2, 3, 4, 5$). The initial node is n_0. The PVD also contains a box, which is labeled with thread t. This box encloses all nodes m_i.

Node n_0 describes the situation in which thread k has not yet get a lock from procedure INSERT. All nodes n_i with $i = 2, 3, 4, 5$ represent the situation in which thread k is executing procedure INSERT and is the leader. This means that, in this case, thread k has no obstacle for advancing through the list. In particular, node n_2 represents the situation in which thread k is within INSERT procedure and it is the leader. Node n_3 is similar to node n_2, except that it models the case in which pointer INSERT::$prev$ is about to advance in the list. Node n_4 represents the situation in which thread k is about to insert a new element in the list. Finally, node n_5 describes the situation in which the element has already been inserted and thus it just remains to release the locks of the nodes pointed by INSERT::$prev$ and INSERT::$curr$. That means, once at node n_2, n_3, n_4 or n_5, the diagram states that no other thread can prevent thread k from progressing.

The situation in which another thread different from k is the leader is modeled by the box. All nodes within the box represent the situation in which thread k has a lock in the list and is executing procedure INSERT, but it is not k the thread which owns the lock closest to the tail of the list. Instead, we use the box parameter t to denote the leader thread. Note how thread t can be executing any of the procedures SEARCH, INSERT or REMOVE.

The diagram begins at node n_0, until thread k gets its first lock due to the execution of transition $\tau_{25}^{(k)}$ of procedure INSERT. When this transition is executed, if there is no other thread with a lock in the list, then thread k becomes the leader and the diagram moves to node n_2. On the contrary, if some other thread t owns the lock closest to the tail, then the diagram moves to any of the nodes within the box depending on the current program location of thread t. While thread t is the leader, the diagram moves within the box, preserving the value of t. Thread t will stop been the leader whenever it finishes executing procedure SEARCH, INSERT or REMOVE. The situation in which thread t releases its last lock is modeled by transitions $\tau_{22}^{(\tau)}$, $\tau_{43}^{(\tau)}$ and $\tau_{61}^{(\tau)}$. Note how, when these transitions are taken, thread k becomes the new leader or a new t becomes the leader. Once thread k becomes the leader, then it can advances through the list without any interference from other threads.

For proving acceptance we use the ghost variables $aheadSet$ and $aheadInsert$. The ranking function associated to the acceptance condition is a triple that follows in fact a lexicographic order. The first component is $aheadSet$. Remember $aheadSet$ contains the set of thread identifiers owning a lock ahead of thread k. The idea is that every time a thread t stops being the leader (due to the execution of transition $\tau_{22}^{(t)}$, $\tau_{43}^{(t)}$ or $\tau_{61}^{(t)}$), the cardinality of this set decrements. The second component of our ranking function is $aheadInsert$, which keeps the threads that are executing procedure INSERT and own a lock ahead of thread k. To understand the need of this second component we need to first look the third component of the ranking function. The third component is the distance between INSERT::$prev$ and the tail of the list. As thread k progresses through the list, the distance will decrement. However, there is one case in which the distance between INSERT::$prev$ would not decrement. A arbitrary thread j inserting an element between the current location of INSERT::$prev$ and $tail$ will increment the distance between INSERT::$prev$ and $tail$ when executing transition $\tau_{38}^{(j)}$. However, when in this case transition $\tau_{38}^{(j)}$ is taken, the set $aheadInsert$ does decrement, making the ranking function to strictly decrement as desired.

Table 10.10 shows the results we have obtained using LEAP on our verification diagram

```
MODULE main
VAR
  state : {qn0,qn2,qn3,qn4,qn5,qm1,qm2,qm3,qm4,qm5,gn0,gn2,gn3,gn4,gn5,
           gm1,gm2,gm3,gm4,gm5,bn0,bn2,bn3,bn4,bn5,bm1,bm2,bm3,bm4,bm5};
  k_begins_insert : boolean;
        k_finishes_insert : boolean;

INIT state = qn0;
TRANS
  case
    state = qn0 : next(state) in {gm1,gm2,gm3,gm4,gm5,gn0,gn2,
                                  bn0,bm1,bm2,bm3,bm4,bm5,bn2};
    state = qm1 : next(state) in {gn4,gn5,qm1,qm2,qm3,qm4,qm5,
                                  bm1,bm2,bm3,bm4,bm5,bn4,bn5};
    state = qm2 : next(state) in {gn4,gn5,qm1,qm2,qm3,qm4,qm5,qn2,qn3,
                                  bm1,bm2,bm3,bm4,bm5,bn2,bn3,bn4,bn5};
    state = qm3 : next(state) in {gn4,gn5,qm1,qm3,bm1,bm3,bn4,bn5};
    state = qm4 : next(state) in {gn4,gn5,qm1,qm2,qm3,qm4,qm5,qn2,qn3,
                                  bm1,bm2,bm3,bm4,bm5,bn2,bn3,bn4,bn5};
    state = qm5 : next(state) in {gn4,gn5,qm1,qm2,qm3,qm4,qm5,qn2,qn3,
                                  bm1,bm2,bm3,bm4,bm5,bn2,bn3,bn4,bn5};
    state = qn2 : next(state) in {gn4,gn5,qn2,qn3,bn2,bn3,bn4,bn5};
    state = qn3 : next(state) in {qn2,qn3,bn2,bn3};
    state = qn4 : next(state) in {gn5,qn4,bn4,bn5};
    state = qn5 : next(state) in {gn0,gn5,bn5,bn0};
    state = gn0 : next(state) in {gm1,gm2,gm3,gm4,gm5,gn0,gn2};
    state = gm1 : next(state) in {gn4,gn5,qm1,qm2,qm3,qm4,qm5};
    state = gm2 : next(state) in {gn4,gn5,qm1,qm2,qm3,qm4,qm5,qn2,qn3};
    state = gm3 : next(state) in {gn4,gn5,qm1,qm3};
    state = gm4 : next(state) in {gn4,gn5,qm1,qm2,qm3,qm4,qm5,qn2,qn3};
    state = gm5 : next(state) in {gn4,gn5,qm1,qm2,qm3,qm4,qm5,qn2,qn3};
    state = gn2 : next(state) in {gn4,gn5,qn2,qn3};
    state = gn3 : next(state) in {qn2,qn3};
    state = gn4 : next(state) in {gn5,qn4};
    state = gn5 : next(state) in {gn0,gn5};
    state = bn0 : next(state) in {bn0,bm1,bm2,bm3,bm4,bm5,bn2};
    state = bm1 : next(state) in {bm1,bm2,bm3,bm4,bm5,bn4,bn5};
    state = bm2 : next(state) in {bm2,bn2,bn3,bn4,bn5};
    state = bm3 : next(state) in {bm3,bn4,bn5};
    state = bm4 : next(state) in {bm4,bn2,bn3,bn4,bn5};
    state = bm5 : next(state) in {bm5,bn2,bn3,bn4,bn5};
    state = bn2 : next(state) in {bn2,bn3,bn4,bn5};
    state = bn3 : next(state) in {bn3};
    state = bn4 : next(state) in {bn4,bn5};
    state = bn5 : next(state) in {bn5,bn0};
  esac;
TRANS
  case
    state in {qn0,gn0,bn0} : !k_begins_insert & !k_finishes_insert;
    state in {qm1,gm1,bm1} : k_begins_insert & !k_finishes_insert;
    state in {qm2,gm2,bm2} : k_begins_insert & !k_finishes_insert;
    state in {qm3,gm3,bm3} : k_begins_insert & !k_finishes_insert;
    state in {qm4,gm4,bm4} : k_begins_insert & !k_finishes_insert;
    state in {qm5,gm5,bm5} : k_begins_insert & !k_finishes_insert;
    state in {qn2,gn2,bn2} : k_begins_insert & !k_finishes_insert;
    state in {qn3,gn3,bn3} : !k_begins_insert & !k_finishes_insert;
    state in {qn4,gn4,bn4} : !k_begins_insert & !k_finishes_insert;
    state in {qn5,gn5,bn5} : !k_begins_insert & k_finishes_insert;
  esac;

  FAIRNESS (state in {gn0,gm1,gm2,gm3,gm4,gm5,gn2,gn3,gn4,gn5,
                      bn0,bm1,bm2,bm3,bm4,bm5,bn2,bn3,bn4,bn5});
  LTLSPEC G (k_begins_insert -> F k_finishes_insert);
```

Figure 10.11: NuSMV input representation for checking condition (ModelCheck) for the PVD that shows that concurrent lock-coupling lists satisfy property eventually_insert(k).

	#VC	#solved VC		single VC time(s.)		DP	LEAP
		pos	TLL	slowest	average	time(s)	time(s)
Initiation	1	0	1	0.01	0.01	0.01	0.01
Consecution	1550	1343	207	3.80	0.05	78.12	3.42
Acceptance	5404	4352	1052	191.61	0.12	647.04	1.61
Fairness	48	20	28	0.42	0.16	7.82	0.14

Table 10.10: Running times for the verification of property eventually_insert(k) for a concurrent lock-coupling list.

to verify property eventually_insert(k) for the concurrent lock-coupling lists. The rows in the table correspond to the rules of initiation, consecution acceptance and fairness. For each rule we present the total number of generated verification conditions, the number of verification conditions solved using the location based decision procedure and the number of verification conditions solved using the TL3 decision procedure. In all cases, all verification conditions are verified as valid. Additionally, the last four columns in the table present the slowest and average time for solving a single verification condition, the total time it takes the decision procedures to check all verification conditions and the total time it takes LEAP to these verification conditions from the diagram.

Finally, for checking that the propositional models of the diagram are included in traces of the property eventually_insert(k), we translate the edge-Streett automaton represented by our PVD into a non-deterministic Büchi automaton (NBW) following the rules described in Appendix A. We use NuSMV to model-check whether our property eventually_insert(k) holds considering the NBW automaton we have just constructed. Fig. 10.11 shows the representation of the model-checking problem analyzed in NuSMV.

In all cases, all generated verification conditions are valid, which implies that the implementation of lock-coupling lists satisfies property eventually_insert(k).

10.6 Parametrized Invariant Generation

We now present the empirical evaluation we have done over a collection of examples using our parametrized invariant generation technique presented in Chapter 5. We have implemented the reflective and interference abstraction schemes in a prototype tool. The input language for our prototype rule extends the *simple programming language* (SPL) presented in Section 2.1. After compiling a parametrized program written in an imperative language into a transition system, we generate inductive assertions using the lazy, eager, and eager+ reflective abstraction schemes and the interference abstraction scheme. Our tool directly uses the abstract domains implemented in the Apron library [118]. Narrowing is used for the eager, eager+, and interference schemes but not the lazy scheme. The main questions that we seek to answer are:

1. how effective are each of these schemes at generating invariants of interest; and
2. how do the invariants generated by each scheme compare with each other in terms of precision.

Second, we study the performance of the analysis for each scheme.

```
    global
       Int count = N
    procedure SIMPLEBARRIER()
       Int x = 0
    begin
1:  count := count − 1
2:  await (count = 0)
    end procedure
```

Figure 10.12: SIMPLEBARRIER: a simple synchronization barrier program.

We consider a set of five benchmarks, including:

1. A simple barrier algorithm [150];
2. a centralized barrier [150];
3. the work stealing algorithm presented in Example 5.1;
4. a generalized version of dinning philosophers with a bounded number of resources inspired by [183]; and
5. a parametrized system model of autonomous swarming robots inside a $m \times n$ grid [56].

In order to study program locations, in our examples we use *counting abstraction*. That is, we use integer variables to denote the number of threads located at a specific program location. In general, we use M to denote the total number of threads in the system and M_ℓ to denote the number of threads located at program line ℓ.

We now describe in detail each of these examples before analyzing them. For each example, we specify a set of target invariants, with the intention to check whether the invariants that are generated automatically invariants imply a given program's safety specification.

10.6.1 SIMPLEBARRIER: A Simple Barrier Algorithm

The first example is a simple barrier synchronization algorithm, described by program SIMPLEBARRIER. Fig. 10.12 presents the code for program SIMPLEBARRIER.

The program consists on 2 program locations, 3 transitions and 4 variables: global variable $count$, and counting abstraction variables M, M_1 and M_2. In SIMPLEBARRIER, the global variable $count$ is initialized with the total number of threads. Then, each thread decrements the global variable by one. Finally, no thread is allowed to end the program until all threads have decremented the variable.

The list of the 4 invariant properties that form the specification of the system are:

- **basic:** which indicates that all threads are at some position. That is expressed by $M1 + M2 + M3 = M$.
- **bound:** which states that variable $count$ is equal to the number of threads at position 1. That is, $count = M1$.
- **l_bound:** which establishes that variable $count$ is always positive. That is, $0 \leq count$.

- **u_bound:** which states an upper bound for variable $count$, indicating that $count$ is not greater than the total number of threads. That is, $count \leq M$.

10.6.2 CENTRALBARRIER: A Centralized Barrier Synchronization Algorithm

Program CENTRALBARRIER, depicted in Fig. 10.13 implements a memory barrier, similar to SIMPLEBARRIER. This program consists on 8 program locations, 10 program transitions and 13 program variables. The memory barrier implemented by CENTRALBARRIER can be reused inside a loop thanks to the use of the $sense$ global variable. Initially, $sense$ and $localSense$ are both set to 0. A thread executing program CENTRALBARRIER starts by changing its own $localSense$ variable (line 1). Then, the thread makes a local copy of the global counter $count$ and decrements the global counter at line 2. At line 3, the thread checks whether it has been the last thread decrementing the global counter. If so, then it set $count$ again to M and changes variable $sense$. On the other hand, if the thread is not the last thread decrementing $count$, then it waits in line 4 until some other thread modifies $sense$.

The 9 invariant properties that conform the specification of the system are:

- **basic:** which describes the property that all threads are in some location. It is specified by the condition: $M = \sum_{i=1}^{7} Mi$
- **sense_l_bound:** which states that variable *sense* is positive. That is, $0 \leq sense$.
- **sense_u_bound:** which establishes that variable *sense* is not greater than 1. That is, $sense \leq 1$.
- **localsense_l_bound:** which indicates that variable *localsense* is positive. That is, $0 \leq localSense$.

```
global
   Int sense = 0
   Int count = N
procedure CentralBarrier()
   Int localSense = 0
   Int localCount = N
begin
      ⟨ if localSense ≠ 0 then
           localSense := 0
1:      else
           localSense := 1
        end if ⟩
2:    ⟨ localCount := count
        count := count − 1 ⟩
3: if localCount > 1 then
4:     await (sense = localSense)
5: else
6:     count := N
7:     sense := localSense
8: end if
end procedure
```

Figure 10.13: CENTRALBARRIER: a centralized synchronization barrier program.

- **localsense_u_bound:** which states that variable *localsense* is never greater than 1. That is, $localSense \leq 1$.
- **count_l_bound:** which indicates that variable *count* is always positive. That is, $0 \leq count$.
- **count_u_bound:** which states that *count* is not greater than M. That is, $count \leq M$.
- **localcount_l_bound:** which establishes that variable *localCount* is positive. That is, $0 \leq localCount$.
- **localcount_u_bound:** which says that variable *localCount* is not greater than M. That is, $localCount \leq M$.

10.6.3 WORKSTEAL: An Array Processing Work Stealing Algorithm

The program WORKSTEAL has already been introduced in Example 5.1. The program consists on 5 program locations, 6 transitions and 10 program variables including global, local and counting abstraction variables. In this program, threads access an array concurrently and perform some work on each element of the array independently.

For this program there are 5 invariant properties that we would like to prove. Their specifications are:

- **basic:** which states that all threads are at some position. That is, $M1+M2+M3+M4 = M$.
- **c_l_bound:** which indicates that variable *c* is always positive. That is, $0 \leq c$.
- **c_u_bound:** which establishes that variable *c* does not go beyond the position indicated by *len*. That is, $c \leq len$.
- **nextpos_bound:** which states that variable *nextpos* is positive. That is, $0 \leq nextpos$.
- **last_bound:** which indicates that variable *last* is bounded by *len*. That is, $last \leq len$.

10.6.4 PHILOSOPHERS: Dining Philosophers With Bounded Resources

We now present program PHILOSOPHERS, a variation of dining philosophers inspired by [183] in which an arbitrary number of philosophers can be eating at each table position simultaneously, with a bounded number of resources. Fig. 10.14 presents the algorithm for program PHILOSOPHERS.

Program PHILOSOPHERS consists of 10 program locations, 16 transitions and 13 program variables including global variables and counting abstraction variables. As it can be seen in the algorithm, there are only 2 resources of type *one* while there are as many resources of type *two* as philosophers in the system. As it can be seen in line 2, there is always a resource of type *Two* available in order to prevent deadlock.

For program PHILOSOPHERS, we specify 14 invariants that should hold. These invariants are:

- **basic:** which indicates that all threads are in some location of the system. That is $M = \sum_{i=1}^{10} Mi$.
- **cons1:** which establishes that the quantity of resources of type *one* stays constant. That is, $2 = res1 + (M7 + M8 + M9) + M4$.

- **cons2:** which states that the quantity of resources of type *two* stays constant. That is, $2 = \mathit{res2} + (M3 + M4 + M5) + M8$.
- **res1:** which says that the quantity of resources of type *one* is always positive. That is, $\mathit{res1} \geq 0$.
- **res2:** which indicates that the quantity of resources of type *two* is also always positive. That is, $\mathit{res2} \geq 0$.
- **use1:** which establishes that if no resource of type *one* is available, then some thread is using it. That is, $\mathit{res1} = 0 \rightarrow (M7 + M8 + M9) + M4 \geq 1$.
- **use2:** which states that if no resource of type *two* is available, then some thread is using it. That is, $\mathit{res2} = 0 \rightarrow (M3 + M4 + M5) + M8 \geq 1$.
- **actLim_res1:** which indicates that the number of threads using resources of type *one* is bounded by the number of resources. That is, $(M7 + M8 + M9) + M4 \leq 2$. i
- **actLim_res2:** which states that the number of threads using resources of type *two* is bounded by the number of resources. That is, $(M3 + M4 + M5) + M8 \leq 2$.
- **actLim_res0:** which says that only one thread can own the last resource. That is, $(M3 + M4 + M5) \leq 1$.
- **someProgress2_1:** which establishes that if a resource of type *two* is required but unavailable, then some thread owns one and will release it. That is, $(M2 > 0 \wedge \mathit{res2} <= 1) \rightarrow (M3 + M4 + M5) + M8 \geq 1$.
- **someProgress2_0:** which indicates that if no resource of type *two* is available, then some thread owns one and will release it. That is, $\mathit{res2} = 0 \rightarrow M8 \geq 1$.

```
    global
      Int res1 = 2
      Int res2 = N
    procedure PHILOSOPHERS()
    begin
 1: nondet choice
 2:         ⟨ await(res2 > 1)
              res2 := res2 − 1 ⟩
 3:         ⟨ await(res1 > 0)
              res1 := res1 − 1 ⟩
 4:         res1 := res1 + 1
 5:         res2 := res2 + 1
 6:   or    ⟨ await(res2 > 0)
              res1 := res1 − 1 ⟩
 7:         ⟨ await(res2 > 0)
              res2 := res2 − 1 ⟩
 8:         res2 := res2 + 1
 9:         res1 := res1 + 1
10: end choice
    end procedure
```

Figure 10.14: PHILOSOPHERS: dining philosophers with a bounded number of resources.

- **someProgress1_1:** which says that if a resource of type *one* is required but available, then some thread owns one and will release it. That is, $(M3 > 0 \wedge \mathit{res1} = 0) \rightarrow (M7 + M8 + M9) + M4 \geq 1$.

- **someProgress1_0:** which states that if no resource of type *tone* is available, then some thread owns one and will release it. That is, $(M6 > 0 \ \wedge \ \mathit{res1} = 0) \rightarrow (M7 + M8 + M9) + M4 \geq 1$.

10.6.5 ROBOTS: Robot Swarm

The last example consists of a swarm of robots moving across an $m \times n$ grid. The example is inspired in part by [56] which models a swarm of robots that perform inspections of tight spaces such as the wing of an airplane.

Fig. 10.15 presents ROBOTS_2_2, an instance of this program in 2×2 board. Program ROBOTS_2_2 consist of 25 program locations, 51 program transitions and 35 variables, including program variables and counting abstractions. In the example, the robots communicate with a central process that maps the number of robots in any grid cell. Each robot updates this central process with its position. Likewise, the central process updates each robot with a global map consisting of the number of robots in each cell. The details of this process are captured in our model by a shared memory view of the number of robots in a given cell.

If a given cell has more than a minimal number of robots (line 1), some of the robots in the cell may optionally migrate to a neighbouring cell. This cell is chosen non-deterministically and the robot's velocity vector is set to point towards the cell. The robot then subtracts itself from the number of robots in the cell and moves for 10 time units in the specified direction (lines 3 to 12). After the time units elapse, the robot re-evaluates its position and adds itself to the count of the number of robots in whatever cell it finds itself in (lines 19 to 24).

The invariant properties that were analyzed for the example of ROBOTS_2_2 are:

- **basic(i,j):** we verify this property for all i and j, such that $1 \leq i \leq 2$ and $1 \leq j \leq 2$. The property states that the number of robots in position i, j should never be negative. That is, $x_i y_j \geq 0$.

- **velocity-bounds:** which states that the velocity of the robots must stay within bounds:

$$-1 \leq vx \leq 1 \quad \wedge \quad -1 \leq vy \leq 1 \quad \wedge \quad -1 \leq vx + vy \leq 1$$

- **bounds:** which indicates that (x, y) coordinates of robots must be within the grids.

$$0 \leq x \leq 20 \ \wedge \ 0 \leq y \leq 20$$

10.6.6 Results and Observations

We now compare the results we have obtained using the different schemes applied to each of the examples described from Section 10.6.1 to Section 10.6.5. Table 10.11 presents a comparison of timings and precision across the lazy, eager, eager+, and interference schemes. We study 8 algorithms, including:

```
global
   Int x1y1 = N
   Int x1y2 = 0
   Int x2y1 = 0
   Int x2y2 = 0
procedure ROBOTS()
   Int x
   Int y
   Int vx = 0
   Int vy = 0
   Int j = 0
begin
1:  await (x ≥ 0 ∧ x ≤ 10 ∧ y ≥ 0 ∧ y ≤ 10 ∧ N ≥ 6)
2:  while 1 = 1 do
3:     nondet choice
4:            ⟨ await(x ≥ 0 ∧ x ≤ 10 ∧ y ≥ 0 ∧ y ≤ 10 ∧ x1y1 ≥ 2)
                vx := 1
                vy := 0
                x1y1 := x1y1 − 1 ⟩
5:        or  ⟨ await(x ≥ 0 ∧ x ≤ 10 ∧ y ≥ 0 ∧ y ≤ 10 ∧ x1y1 ≥ 2)
                vx := 0
                vy := 1
                x1y1 := x1y1 − 1 ⟩
6:        or  ⟨ await(x ≥ 0 ∧ x ≤ 10 ∧ y ≥ 10 ∧ y ≤ 20 ∧ x1y2 ≥ 2)
                vx := 1
                vy := 0
                x1y2 := x1y2 − 1 ⟩
7:        or  ⟨ await(x ≥ 0 ∧ x ≤ 10 ∧ y ≥ 10 ∧ y ≤ 20 ∧ x1y2 ≥ 2)
                vx := 0
                vy := −1
                x1y2 := x1y2 − 1 ⟩
8:        or  ⟨ await(x ≥ 10 ∧ x ≤ 20 ∧ y ≥ 0 ∧ y ≤ 10 ∧ x2y1 ≥ 2)
                vx := 0
                vy := 1
                x2y1 := x2y1 − 1 ⟩
9:        or  ⟨ await(x ≥ 10 ∧ x ≤ 20 ∧ y ≥ 0 ∧ y ≤ 10 ∧ x2y1 ≥ 2)
                vx := −1
                vy := 0
                x2y1 := x2y1 − 1 ⟩
10:       or  ⟨ await(x ≥ 10 ∧ x ≤ 20 ∧ y ≥ 10 ∧ y ≤ 20 ∧ x2y2 ≥ 2)
                vx := −1
                vy := 0
                x2y2 := x2y2 − 1 ⟩
11:       or  ⟨ await(x ≥ 10 ∧ x ≤ 20 ∧ y ≥ 10 ∧ y ≤ 20 ∧ x2y2 ≥ 2)
                vx := 0
                vy := −1
                x2y2 := x2y2 − 1 ⟩
12:    end choice
13:    j := 0
14:    while j ≤ 9 do
15:       x := x + vx
16:       y := y + vy
17:       j := j + 1
18:    end while
19:    nondet choice
20:           ⟨ await(x ≥ 0 ∧ x ≤ 10 ∧ y ≥ 0 ∧ y ≤ 10)
                x1y1 := x1y1 + 1 ⟩
21:       or  ⟨ await(x ≥ 0 ∧ x ≤ 10 ∧ y ≥ 10 ∧ y ≤ 20)
                x1y2 := x1y2 + 1 ⟩
22:       or  ⟨ await(x ≥ 10 ∧ x ≤ 20 ∧ y ≥ 0 ∧ y ≤ 10)
                x2y1 := x2y1 + 1 ⟩
23:       or  ⟨ await(x ≥ 10 ∧ x ≤ 20 ∧ y ≥ 10 ∧ y ≤ 20)
                x2y2 := x2y2 + 1 ⟩
24:    end choice
25: end while
end procedure
```

Figure 10.15: ROBOTS_2_2: a set of robots moving around a 2×2 grid.

- Program SIMPLEBARRIER, presented in Section 10.6.1.
- Program CENTRALBARRIER, presented in Section 10.6.2.
- Program WORKSTEAL, introduced in Example 5.1 and Section 10.6.3.
- Program PHILOSOPHERS, described in Section 10.6.4.
- Four instances of the program ROBOTS, introduced in Section 10.6.5. The instances include ROBOTS_2_2, described in Fig. 10.15 and also instances of 2×3, 3×3 and 4×4 boards named ROBOTS_2_3, ROBOTS_3_3 and ROBOTS_4_4 respectively.

For each problem we study 3 different domains:

- Integers, labeled I in Table 10.11.
- Polyhedra, labeled P in Table 10.11.
- Octogons, labeled O in Table 10.11.

For each scheme we compute the total time (in seconds) that takes our tool to finish with the generation of invariant candidates, the number of widening iterations (Wid) and the number of properties that could be automatically generated (Prp). We use ∞ to represent timeout of 5400

ID	Dom	Prps	Lazy			Eager			Eager+			Interf.		
			Time	Wid*	Prp	Time	Wid	Prp	Time	Wid	Prp	Time	Wid	Prp
SIMPLEBARRIER	I	4	0.1	2	0	0.1	5	0	0.1	5	0	0.1	4	0
	P		0.2	4	4	0.1	5	4	0.1	5	4	0.1	4	4
	O		0.8	3	3	0.1	5	3	0.1	5	3	0.1	4	3
CENTRALBARRIER	I	9	0.9	3	4	0.1	7	0	0.1	8	0	0.1	7	0
	P		∞		0	1.7	11	4	2.7	12	5	1.1	10	6
	O		∞		0	7.5	9	6	11.3	9	6	6.2	8	4
WORKSTEAL	I	5	0.3	6	2	0.1	5	1	0.1	5	1	0.1	4	0
	P		2.4	6	1	0.1	7	1	0.2	7	3	0.1	7	5
	O		8.2	6	4	7.5	6	4	0.2	6	4	6.2	5	4
PHILOSOPHERS	I	14	1.9	4	2	0.1	8	2	0.1	8	2	0.1	7	0
	P		11.8	6	14	1.1	11	8	1.8	11	8	6.3	13	14
	O		∞		0	25	12	4	40	12	4	20	12	4
ROBOTS _2_2	I	16	31.3	8	4	0.4	10	4	0.4	11	4	0.2	10	0
	P		∞		0	9.3	22	3	15	23	3	5.8	15	4
	O		∞		0	142	25	3	225	26	3	105	18	3
ROBOTS _2_3	I	18	133	8	6	0.7	10	6	0.9	11	6	0.5	10	0
	P		∞		0	23	22	5	36.8	23	5	16	15	5
	O		∞		0	404	25	5	629	26	5	320	18	5
ROBOTS _3_3	I	23	1141	8	9	1.6	10	9	2.1	11	9	0.9	10	0
	P		∞		0	68.2	22	8	111.5	23	8	52	15	8
	O		∞		0	1414	25	8	2139	26	8	1168	18	8
ROBOTS _4_4	I	29	∞		0	6.7	11	16	9.4	11	16	3.2	11	0
	P		∞		0	49	23	15	396	23	15	303	15	15
	O		∞		0	∞		0	∞		0	∞		0

Table 10.11: Timing and precision results for Lazy, Eager, Eager+ and Interference abstract interpretations. Legend: **ID:** benchmark identifier, **Dom:** abstract domains, **I:** intervals, **O:** octagons, **P:** polyhedra, **Prps**: total number of properties to be proven, **Time**: seconds, **Prp**: number of properties proved, **Wid**: number of widening iterations.

ID	Dom	L:E	L:E+	L:I	E:I	E+:I	E:E+
SIMPLEBARRIER	I	$-^{3}$	$-^{3}$	$+^{3}$	$+^{3}$	$+^{3}$	$=^{3}$
	P	$=^{3}$	$=^{3}$	$+^{1}\ =^{2}$	$+^{1}\ =^{2}$	$+^{1}\ =^{2}$	$=^{3}$
	O	$=^{3}$	$=^{3}$	$+^{1}\ =^{2}$	$+^{1}\ =^{2}$	$+^{1}\ =^{2}$	$=^{3}$
WORKSTEAL	I	$+^{4}$	$+^{4}$	$+^{4}$	$+^{4}$	$+^{4}$	$=^{4}$
	P	$+^{4}$	$+^{1}\ \neq^{3}$	$=^{1}\neq^{3}$	$-^{2}\ \neq^{2}$	$+^{2}-^{1}\ =^{1}$	$-^{4}$
	O	$=^{4}$	$=^{4}$	$+^{2}\ =^{2}$	$+^{2}\ =^{2}$	$+^{2}\ =^{2}$	$=^{4}$
CENTRALBARRIER	I	$+^{6}\ \neq^{1}$	$+^{6}\ \neq^{1}$	$+^{7}$	$+^{7}$	$+^{7}$	$=^{7}$
	P	n.a.	n.a.	n.a.	$\neq^{7}$	$+^{1}\ \neq^{6}$	$-^{7}$
	O	n.a.	n.a.	n.a.	$+^{7}$	$+^{7}$	$=^{7}$
PHILOSOPHERS	I	$+^{1}\ =^{9}$	$+^{1}\ =^{9}$	$+^{10}$	$+^{10}$	$+^{10}$	$=^{10}$
	P	$+^{10}$	$+^{10}$	$+^{1}\ =^{9}$	$-^{10}$	$-^{10}$	$=^{10}$
	O	n.a.	n.a.	n.a.	$+^{1}\ =^{9}$	$+^{1}\ =^{9}$	$=^{10}$
ROBOTS (2,2)	I	$+^{22}$	$+^{22}$	$+^{22}$	$+^{22}$	$+^{22}$	$=^{22}$
	P	n.a.	n.a.	n.a.	$+^{2}-^{3}\ =^{17}$	$+^{2}-^{1}\ =^{19}$	$-^{2}\ =^{20}$
	O	n.a.	n.a.	n.a.	$+^{1}-^{4}\ =^{16}\neq^{1}$	$+^{2}\ =^{20}$	$-^{5}\ =^{17}$
ROBOTS (2,3)	I	$+^{30}$	$+^{30}$	$+^{30}$	$+^{30}$	$+^{30}$	$=^{30}$
	P	n.a.	n.a.	n.a.	$+^{2}-^{3}\ =^{25}$	$+^{2}-^{1}\ =^{27}$	$-^{2}\ =^{28}$
	O	n.a.	n.a.	n.a.	$+^{1}-^{7}\ =^{21}\neq^{1}$	$+^{2}\ =^{28}$	$-^{8}\ =^{22}$
ROBOTS (3,3)	I	$+^{43}$	$+^{43}$	$+^{43}$	$+^{43}$	$+^{43}$	$=^{43}$
	P	n.a.	n.a.	n.a.	$+^{2}-^{3}\ =^{38}$	$+^{2}-^{1}\ =^{40}$	$-^{2}\ =^{41}$
	O	n.a.	n.a.	n.a.	$+^{1}-^{10}\ =^{31}\neq^{1}$	$+^{2}\ =^{41}$	$-^{11}\ =^{32}$
ROBOTS (4,4)	I	n.a.	n.a.	n.a.	$+^{74}$	$+^{74}$	$=^{74}$
	P	n.a.	n.a.	n.a.	$+^{3}-^{3}\ =^{68}$	$+^{3}-^{1}\ =^{70}$	$-^{2}\ =^{72}$
	O	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.

Table 10.12: Comparison of invariants of various schemes.

seconds. In the case of the lazy scheme, the number of widening iterations corresponds to the number of external widening applications.

From our analysis, we can conclude that while interference abstractions are the fastest, as expected, it is perhaps surprising to note that the lazy scheme was markedly slower than the remaining techniques considered. In fact, the lazy scheme times out on many instances. Likewise, we note that eager and eager+ were only a little slower on most of the benchmarks when compared to interference abstraction. Also, the time for using polyhedra is generally faster than octagons. According to the Apron authors, the execution time of polyhedra can vary widely between good and bad cases, while the worst case and best case execution time of octagons is the same, which may explain this observation. Other result that catches our attention is that the interference semantics fares noticeably better than the other schemes for the polyhedral domain but noticeably worse on the interval domain. Also, the interval domain itself seems to fare surprisingly better than the polyhedral domain in terms of properties proved. In many cases, however, the properties proved by these domains were non-overlapping. Perhaps the simplest explanation for this result is that the properties themselves mostly concern proving bounds on variables. It is not surprising then that the interval domain can establish this. Yet another factor is the use of polyhedral widening. Since a widening needs to be carried out at every location in the program, the loss of precision in the polyhedral domain can be considerable.

Table 10.12 compares each pair of methods in terms of the relative strengths of the invariants inferred. In the table, we use “L” to denote the lazy scheme, “I” for the interference scheme, “E” for the eager approach and “E+” for the eager+ scheme. In a comparison $A : B$, $+$ indicates

that A obtained a stronger invariant at a location, $-$ means A obtained strictly weaker invariant, $=$ means A and B obtained equivalent location and $\neq$ means A and B obtained incomparable invariant. We use "n.a." (not analyzed) when one of the analyzed schemes timed out and hence it was not possible to make a comparison. The comparison shows outcomes and number of locations as superscript.

From our experiments, some surprising patterns are revealed. For example, the lazy, eager and eager+ schemes prove stronger invariants for the interval domain when compared to the interference scheme. On the other hand, the trend is reversed for the polyhedral domain. In many cases, the invariants are either incomparable or invariants of one technique are stronger at some location and weaker at others. Conjoining the invariants in these cases can produce stronger invariants overall.

In theory, all the methods presented can be viewed as post-fixed point computations in the product domain representing sets of states and reflective abstractions. Our intuition with abstract interpretation suggests that the interference scheme, which applies a single iteration on the sequential system generated from the $\top$ reflection, should fare worse than the eager scheme which computes a least fixed point using Kleene iteration. However, the experiments let us conclude that the widening and the associated non-monotonicity play a significant role for parametrized systems. This effect is much more evident than for sequential systems, where our experience suggests that non-monotonicity of widening plays a more limited role.

11 Conclusions

> *Exactly! So once you do know what the question actually is, you'll know what the answer means.*
>
> Deep Thought
> (The Hitchhiker's Guide to the Galaxy)

11.1 Summary

We have presented a novel framework based on deductive techniques for the verification of temporal properties of concurrent parametrized systems. The current target is the verification of concurrent data structures that manipulate the heap, but the framework can be extended to support other parametrized systems.

A key novelty of the approach we have presented is that it cleanly differentiates the control flow of the program from the data type being manipulated:

- The program control flow is analyzed using specialized parametrized proof rules and verification techniques, named *parametrized invariance* and *parametrized verification diagrams*, which were presented in Part I.
- The data types are analyzed using specialized decision procedures, as the ones presented in Part II.

An advantage of this scheme is that the methods presented in Part I can be applied independently of the data types the program manipulates, as far as a decision procedure for such data type exists.

The *parametrized invariance* rules presented in Chapter 3 extend the classical deductive proof rules for closed systems adapting them for parametrized systems executed by an unbounded

number of threads. These proof rules are specifically designed for tackling the uniform verification problem of safety properties of parametrized infinite state processes.

On the other hand, the *parametrized verification diagrams* introduced in Chapter 4 extend *generalized verification diagrams*, enabling the verification of temporal properties, including liveness properties, in concurrent systems executed by an unbounded number of processes. In the case of PVDs, a diagram is designed so that it encodes in a single proof an evidence that all instances of the parametrized system satisfy a given temporal specification.

In both cases, *parametrized invariance* rules and PVDs ultimately generate a finite collection of verification conditions whose validity ensures that the property is satisfied when the system is executed by an unbounded number of threads. A key characteristic of our approach is that the size of this collection of verification conditions depends only on the program description and the index of the formula to prove, but not on the number of threads in a particular instance. Additionally, the verification conditions are quantifier-free as long as the initial specifications are quantifier free.

So far, the proof rules for safety and liveness are amenable for fully symmetric systems in which thread identifiers are only compared with equality, which encompasses many real systems. However, other topologies like rings of processes or totally ordered collections of processes can be handled with variations of our proof rules. In fact, it is straightforward to extend our framework to a finite family of process classes. For example, in the current model, we consider all threads explicitly appearing in the specification plus a single extra fresh thread which abstracts the effect of the remaining threads. If we would like to model client-server systems, then we would just need to consider a fresh *client* thread and a fresh *server* thread in order to abstract the remaining threads of the system.

In Part II we presented a set of theories for some pointer based data structures such as concurrent lists and skiplists. These theories were:

- Chapter 6 presented TL3, a *Theory of Linked Lists with Locks* capable of reasoning about addresses, pointer manipulation, elements, cells, memories, paths, explicit heap regions and lock ownership. The theory is specifically designed for describing rich properties, including structural and functional properties, of concurrent data structures that preserve linked-lists. This makes the theory also a candidate for the analysis of not only lists but also other data structures alike such as stacks and queues.

- Chapter 7 introduced $\mathsf{TSL_K}$, a *Family of Theories of Concurrent Skiplists with at Most K Levels*. Each member of the $\mathsf{TSL_K}$ family is a theory capable of reasoning about concurrent skiplists of unbounded length but with a bounded number of levels (at most K). The $\mathsf{TSL_K}$ family extends from TL3 by extending the functions and predicates of TL3 to the K levels of a skiplist.

- Chapter 8 presented TSL, a *Theory of Skiplists with Unbounded Levels* which is suitable for analyzing skiplist of both unbounded length and unbounded number of levels.

For each of these theories, we showed that the quantifier-free fragment is decidable and we presented a decision procedure for each theory.

In the case of TL3 and $\mathsf{TSL_K}$ we presented a bounded model theorem, which ensures that given a quantifier-free TL3 or $\mathsf{TSL_K}$ formula, it is possible to compute the bounds of the domains for a model of the formula (if such model exists). These bounds depend only on the literals

occurring in the formula. Hence, a decision procedure for TL3 and TSL_K can determine the satisfiability of a formula just by exploring the models up to the computed bounds.

In the case of TSL, we showed that the quantifier-free TSL fragment is decidable by reducing its satisfiability problem to TSL_K, which keeps the complexity of the satisfiability problem NP-complete. When reducing the TSL satisfiability problem to TSL_K, the proposed decision procedure only needs to reason about those levels explicitly mentioned in the TSL formula.

In order to show the feasibility of the proposed verification techniques and decision procedures, in Chapter 9 we presented LEAP, a prototype theorem prover which implements the ideas presented in this work. We used LEAP to verify some implementations of realistic data structures including concurrent lock-couping single-linked lists, lock-free stacks and queues, concurrent bounded and unbounded skiplists, with very promising results, as shown in Chapter 10.

11.1.1 Answered Questions

At the beginning of this work, in Chapter 1, we presented the target problems we intend to tackle. We can conclude that:

Parametrized Verification: One of the main objectives of this work was to study the verification of concurrent parametrized systems. That is, given a program and a parametrized temporal property, we try to answer the question of whether such temporal property is satisfied when the program is executed by any number of threads.

Our novel deductive techniques solve this problem and is specifically designed for parametrized systems. Our technique is based on well known and used deductive methods for closed systems. A key aspect of the methods we have presented is that they generate a finite collection of verification conditions whose validity entails the satisfiability of the parametrized temporal property. Moreover, the proposed methods do not rely on the data type manipulated by the program, which make them suitable for the analysis of a wide range of programs.

Safety and Livenes Verification: When we started this work, we wanted to find methods for the verification of general temporal parametrized properties including safety and liveness properties.

We have developed methods suitable for both kind of properties. The *parametrized invariance* rules are specifically designed for the uniform verification problem of safety properties of parametrized infinite state processes. On the other hand, the *parametrized verification diagrams* allow to prove temporal properties, specially of liveness, of parametrized concurrent systems.

Analysis of Wide Range of Data Structures: A third objective we had was to make our techniques capable of dealing with a wide range of different data structures, not limiting ourselves to programs that manipulate only simple types like Boolean or integers.

We have accomplished this by developing a framework which separates the analysis of the control flow (through the techniques described above) from the analysis of the data types manipulated by the program. To show the feasibility of our framework, we have studied decidable theories for complex pointer based concurrent data structures such as lists and skiplists. We proved each theory to be decidable (for the quantifier-free fragment) and we

have proposed a decision procedure for each theory. Moreover, we have used the proposed decision procedures in combination with the parametrized verification techniques in order to successfully verify structural and functional properties of pointer-based data structures.

Parametrized Invariant Generation: Related to the verification of parametrized systems, we explored the possibility of automatically generate parametrized invariants. The result was a technique capable of inferring multi-indexed invariants for parametrized systems, but what is even more important, is that our method relies on off-the-shelf invariant generators for sequential systems. Despite out research was limited to programs that manipulate only integer values, the empirical evaluation we carried out have demonstrated the applicability of this novel technique over a wide range of examples.

An advantage of the approach we have followed is that the verification techniques for parametrized systems and the decision procedures can be used in combination (as in our framework) or independently, which enables the future analysis of more data structures as far a decision procedure exists for such data structure.

11.2 Open Questions and Future Work

There are still some open questions which may lead to future work extending our framework.

11.2.1 Parametrized Invariance

Regarding the parametrized invariance rules, an interesting research direction is to relax the requirement of full symmetry to cover other process topologies, like for example process rings or totally order processes. Handling these topologies requires to specialize the proof rules presented in Chapter 3 by adapting the premises that refer to threads not in the formula (premises P3, S3, R3 and G3) to consider all cases according to the topology. For example, totally order processes would require to split the case $i \neq j$ into $i < j$ and $i > j$.

Other interesting direction for future work includes invariant generation to simplify or even automate proofs. A preliminary work involving the generation of invariants for parametrized systems that manipulate integers was presented in Chapter 5. The idea we have presented involves the use of off-the-shelf existing invariant generators for sequential systems, adapting them for the use on parametrized systems. However we have not studied the applicability of a invariant generator for more complex data types using the same technique. This is still an open question.

Another interesting direction for future work involves the study of how to apply the decision procedures for the calculation of precondition formulas (like [132]), extended to parametrized systems, to effectively infer candidate invariants from the target specification. Similarly, it would be interesting to study how to extend the “invisible invariant” approach [10,169,214] to processes that manipulate infinite state, not only by instantiating small systems with a few threads (like in invisible invariants) but also by limiting the counter-model exploration to a bounded size, heuristically determined. The candidate invariants produced this way should then be verified with the proof rules presented in this work for the unrestricted system. We envision this method to be a smart exploration of the space of candidate invariants, but its fully feasibility is still an open question.

11.2.2 Parametrized Verification Diagrams

We have shown that PVDs are sound but we do not know whether they are a complete formalism. The study of completeness for PVDs is still part of the future work.

As for parametrized invariance, PVDs currently are suitable for symmetric systems. Again, an interesting area for future exploration involves the relaxations of the symmetry requirement.

Another interesting method for the verification of liveness properties is *transition invariants* [170]. It would be interested to study how to adapt such method for the verification of parametrized systems.

Finally, as PVDs highly rely on invariant candidates, it would be interesting to study invariant generation methods (as the ones discussed previously in Section 11.2.1) for generating support for PVDs.

11.2.3 Parametrized Invariant Generation

The invariant generation technique presented in Chapter 5 is an abstract interpretation-based method which leverages existing off-the-shelf invariant generators for sequential systems, making them suitable for the generation of parametrized invariants. However, all the invariant generators that we have explored are limited to integers. An interesting direction for future work would be the adaption of our framework for the use of invariant generators capable of handling more complex data types and not only numerical domains. Related to this, it would also be interested to study the development of novel invariant generators based on abstract-interpretation techniques, even for sequential systems, which will allow us to explore further in the invariant generation for complex systems.

Related to widening, a future direction of research is to focus on minimizing the use of widening or avoiding widening altogether using constraint-based techniques [52], or recent advances based on policy and strategy iterations [85, 86].

Finally, another interesting approach is to explore the possibility of using the parametrized invariance proof rules presented to enable a Horn-clause verification engine [95] to automatically generate parametrized invariants guided by the invariant candidate goal.

11.2.4 Decision Procedures

In this work we have introduced some theories for concurrent data structures such as single-linked lists and skiplists. We have also shown that the quantifier-free fragments of these theories are decidable and we have proposed a decision procedure for each of them.

For future development, regarding $\mathsf{TL3}$ and TSL_K it would be interesting to study improvements regarding the bounds of the finite model theorem which will lead to a more efficient decision procedure. In order to improve these decision procedures, one possibility would be to implement first-order resolution steps in combination with decision procedures. This way, we could syntactically simplify and reduce the formula.

A possibility for improving the efficiency of $\mathsf{TL3}$ and TSL_K decision procedures would be to implement a Nelson-Oppen version of them. To do so, we would need to implement decision procedures for each independent subtheory and build a framework to propagate equalities and inequalities and decide on splits. We would start consulting between the decision procedures of the different theories. If new equalities/inequalities arises, then we need to propagate them

between the different theories. When no more propagations are available, we need to start splitting, guessing equalities (for instance, $a = b$ and $a \neq b$). At most, when the formula is saturated, we can get a conclusion whether the formula is satisfiable or not.

In the case of TSL, it would be interesting to extend this theory so that it can deal with concurrent skiplists implementations. This extension may involve a considerable amount of theoretical work, but it should be feasible, as TSL naturally extends from $\mathsf{TSL_K}$, which already support concurrency. The concurrent extension of TSL would allow to deal with other industrial skiplist implementations like the implementation in the `java.concurrent` standard library.

A different approach would be to experiment with local theory extensions [192] and its combination methods [193], which can be used to reduce the satisfiability problem of a theory into a satisfiability problem of the base theory from which the main theory extends.

A more general direction for future work is the study of more decidable theories for describing complex data types. As interesting approach is to study how to extend the theories presented in this work in the construction of richer theories. For instance, a theory of hash maps may use TL3 as a subtheory for describing the structure that stores elements within the same bucket.

11.2.5 LEAP

LEAP is a prototype tool under development, so there are numerous future extensions.

Currently, LEAP parses programs written in its own input language. It would be possible to modify LEAP to parse programs written in real languages such as Java or C. For example, a possibility could be using CIL/Frama-C [60, 125, 160] as a front-end for C.

Following some of the ideas for parametrized invariance and verification diagrams, a possibility is to adapt LEAP to cover non-fully symmetric concurrent systems.

As the empirical evaluation suggests, the instantiation of support for parametrized invariance and verification diagrams is critical to the efficiency of our decision procedures and hence to the effectiveness of our verification method. This is because the size of the formula passed to the decision procedure depends heavily on the instantiation of the support. The tactics currently supported by LEAP for instantiating support are rather heuristic. A long term research is to consider the study of more rigorous and sophisticated methods for instantiation, or even to develop decision procedures that include instantiation. Promising directions for this study are local theory extensions [192] and the search for natural proofs [172]. This line can also potentially lead to complete methods for some class of programs and theories of data [201]. Automation can also be improved by the generation and propagation of invariants. Regarding tactics, we believe that adding tactics based on first-order and theory related axioms could also contribute in easing the proofs using LEAP.

Our experience with the use of LEAP suggests that general invariants for data structures are widely used later for the verification of more specific properties. Hence, the creation of libraries with collections of already verified properties and invariants associated to specific data structures would ease the verification of further future specifications.

Bibliography

[1] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. "Handling Global Conditions in Parametrized System Verification". In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 134–145. Springer-Verlag, 1999.

[2] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. "General Decidability Theorems for Infinite-State Systems". In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 313–321. IEEE Computer Society Press, 1996.

[3] Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. "Parameterized Verification of Infinite-State Processes with Global Conditions". In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 145–157. Springer-Verlag, 2007.

[4] Parosh Aziz Abdulla, Lukás Holík, Bengt Jonsson, Ondrej Lengál, Cong Quy Trinh, and Tomás Vojnar. "Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata". In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13)*, volume 8172 of *LNCS*, pages 224–239. Springer-Verlag, 2013.

[5] Anant Agarwal and Mathews Cherian. "Adaptive Backoff Synchronization Techniques". In *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA'89)*, pages 396–406. ACM, 1989.

[6] Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. "Comparison Under Abstraction for Verifying Linearizability". In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 477–490. Springer-Verlag, 2007.

[7] Thomas E. Anderson. "The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors". In *Proceedings of the 18th International Conference on Parallel Processing (ICPP'89)*, pages 170–174. Pennsylvania State University Press, 1989.

[8] Krzysztof R. Apt, Jan A. Bergstra, and Lambert G. L. T. Meertens. “Recursive Assertions Are Not Enough - Or Are They?”. *Theoretical Computer Science*, 8:73–87, 1979.

[9] Krzysztof R. Apt and Dexter C. Kozen. “Limits for Automatic Verification of Finite-State Concurrent Systems”. *Information Processing Letters*, 22(6):307–309, 1986.

[10] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore D. Zuck. “Parameterized Verification with Automatically Computed Inductive Assertions”. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 221–234. Springer-Verlag, 2001.

[11] James Aspnes, Maurice Herlihy, and Nir Shavit. “Counting Networks”. *Journal of the ACM*, 41(5):1020–1048, 1994.

[12] Franz Baader and Cesare Tinelli. “A New Approach for Combining Decision Procedure for the Word Problem, and Its Connection to the Nelson-Oppen Combination Method”. In *Proceedings of the 14th International Conference on Automated Deduction (CADE'97)*, volume 1249 of *LNCS*, pages 19–33. Springer-Verlag, 1997.

[13] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. “Shape Analysis by Predicate Abstraction”. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, pages 164–180. Springer-Verlag, 2005.

[14] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. “Regional Logic for Local Reasoning about Global Invariants”. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, pages 387–411. Springer-Verlag, 2008.

[15] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. “CVC4”. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 171–177. Springer-Verlag, 2011.

[16] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesar Tinelli. *Handbook of Satisfiability*, chapter Satisfiability Modulo Theories. IOS Press, 2008.

[17] Clark Barrett and Cesare Tinelli. “CVC3”. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 298–302. Springer-Verlag, 2007.

[18] Clark W. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories*. Ph.D., Stanford University, January 2003.

[19] Peter Baumgartner, Alexander Fuchs, Hans de Nivelle, and Cesare Tinelli. “Computing Finite Models by Reduction to Function-free Clause Logic”. *Journal of Applied Logic*, 7(1):58–74, 2009.

[20] Michael Benedikt, Thomas W. Reps, and Shmuel Sagiv. “A Decidable Logic for Describing Linked Data Structures”. In *Proceedings of the 8th European Symposium on Programming (ESOP'99)*, pages 2–19, 1999.

[21] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. "Shape Analysis for Composite Data Structures". In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590, pages 178–192. Springer-Verlag, 2007.

[22] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. "A Decidable Fragment of Separation Logic". In *Proceedings of the 24th Conference Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*, volume 3328, pages 97–109. Springer-Verlag, 2004.

[23] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. "Smallfoot: Modular Automatic Assertion Checking with Separation Logic". In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *LNCS*, pages 115–137. Springer-Verlag, 2005.

[24] Josh Berdine, Tal Lev-Ami, Roman Manevich, Ganesan Ramalingam, and Shmuel Sagiv. "Thread Quantification for Concurrent Shape Analysis". In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV'08)*, volume 5123 of *LNCS*, pages 399–413. Springer-Verlag, 2008.

[25] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[26] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. "Leveraging Existing Instrumentation to Automatically Infer Invariant-constrained Models". In *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13)*, pages 267–277. ACM, 2011.

[27] Jesse D. Bingham and Zvonimir Rakamaric. "A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs". In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, pages 207–221. Springer-Verlag, 2006.

[28] Nikølaj Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner, Zohar Manna, Henny Sipma, and Tomás E. Uribe. "Verifying Temporal Properties of Reactive Systems: A STeP Tutorial". *Formal Methods in System Design*, 16(3):227–270, 2000.

[29] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software (invited chapter).". In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, pages 85–108. Springer-Verlag, 2005.

[30] Guy E. Blelloch, Perry Cheng, and Phillip B. Gibbons. "Room Synchronizations". In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'01)*, pages 122–133, 2001.

[31] Stefan Blom and Marieke Huisman. "The VerCors Tool for Verification of Concurrent Programs". In *Proceedings of the 19th International Symposium on Formal Methods (FM'14)*, volume 8442 of *LNCS*, pages 127–131. Springer-Verlag, 2014.

[32] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. "Permission Accounting in Separation Logic". In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 259–270. ACM, 2005.

[33] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. "Variables as Resource in Separation Logic". *Electr. Notes Theor. Comput. Sci.*, 155:247–276, 2006.

[34] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. "A Logic-Based Framework for Reasoning about Composite Data Structures". In *Proceedings of the 20th International Conference on Concurrency Theory (CONCUR'09)*, pages 178–195. Springer-Verlag, 2009.

[35] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. "Accurate Invariant Checking for Programs Manipulating Lists and Arrays with Infinite Data". In *Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis (ATVA'12)*, volume 7561 of *LNCS*, pages 167–182. Springer-Verlag, 2012.

[36] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, volume 6538 of *LNCS*, pages 70–87. Springer-Verlag, 2011.

[37] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. "What's Decidable About Arrays?". In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *LNCS*, pages 427–442. Springer-Verlag, 2006.

[38] Stephen D. Brookes. "A Semantics for Concurrent Separation Logic". In *Proceedings of the 15th International Conference in Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 16–34. Springer-Verlag, 2004.

[39] Anca Browne, Zohar Manna, and Henny B. Sipma. "Generalized Verification Diagrams". In *Proceedings of the 15th Conference Foundations of Software Technology and Theoretical Computer Science (FSTTCS'15)*, volume 1206 of *LNCS*, pages 484–498. Springer-Verlag, 1995.

[40] R. M. Burstall. "Some techniques for Proving Correctness of Programs Which Alter Data Structures". *Machine Intelligence*, 7:23–50, 1972.

[41] Cristiano Calcagno and Dino Distefano. "Infer: An Automatic Program Verifier for Memory Safety of C Programs". In *Proceedings of the 3rd International Symposium on NASA Formal Methods (NFM'11)*, volume 6617 of *LNCS*, pages 343–358. Springer-Verlag, 2011.

[42] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. "The TLA$^+$ Proof System: Building a Heterogeneous Verification Platform". In *Proceedings of the 7th International Colloquium on Theoretical Aspects of Computing (ICTAC'10)*, volume 6255 of *LNCS*. Springer-Verlag, 2010.

[43] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. "NUSMV: A New Symbolic Model Verifier". In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 495–499. Springer-Verlag, 1999.

[44] Edmund M. Clarke and Orna Grumberg. "Avoiding The State Explosion Problem in Temporal Logic Model Checking". In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 294–303. ACM, 1987.

[45] Edmund M. Clarke, Orna Grumberg, and Michael C. Browne. "Reasoning About Networks With Many Identical Finite-State Processes". In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing (PODC'86)*, pages 240–248. ACM, 1986.

[46] Edmund M. Clarke, Orna Grumberg, and Somesh Jha. "Verifying Parameterized Networks Using Abstraction and Regular Languages". In *Proceedings of the 6th International Conference in Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 395–407. Springer-Verlag, 1995.

[47] Edmund M. Clarke, Orna Grumberg, and Somesh Jha. "Verifying Parameterized Networks". *ACM Transactions on Programming Languages and Systems*, 19(5):726–750, 1997.

[48] Edmund M. Clarke, Somesh Jha, Reinhard Enders, and Thomas Filkorn. "Exploiting Symmetry in Temporal Logic Model Checking". *Formal Methods in System Design*, 9(1/2):77–104, 1996.

[49] Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. "Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems". In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 33–47. Springer-Verlag, 2008.

[50] Ariel Cohen and Kedar S. Namjoshi. "Local Proofs for Global Safety Properties". *Formal Methods in System Design*, 34(2):104–125, 2009.

[51] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. "VCC: A Practical System for Verifying Concurrent C". In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLS'09)*, volume 5674 of *LNCS*, pages 23–42, 2009.

[52] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. "Linear Invariant Generation Using Non-linear Constraint Solving". In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 420–433. Springer-Verlag, 2003.

[53] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. "Unifying Type Checking and Property Checking for Low-level Code". In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, pages 302–314. ACM Press, 2009.

[54] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. "Proving That Programs Eventually Do Something Dood". In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 265–276. ACM, 2007.

[55] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. "Terminator: Beyond Safety". In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 415–418. Springer-Verlag, 2006.

[56] Nikolaus Correll and Alcherio Martinoli. "Collective Inspection of Regular Structures Using a Swarm of Miniature Robots". In *Proceedings of the 9th International Symposium on Experimental Robotics (ISER'04)*, volume 21 of *Springer Tracts in Advanced Robotics*, pages 375–386. Springer-Verlag, 2004.

[57] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.". In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977.

[58] Patrick Cousot and Radhia Cousot. "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation, invited paper". In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming (PLILP'92)*, volume 631 of *LNCS*, pages 269–295. Springer-Verlag, 1992.

[59] Travis Craig. "Building FIFO and Priority-Queuing Spin Locks from Atomic Swap". Technical report, University of Washington, Department of Computer Science, 1993.

[60] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C – A Software Analysis Perspective". In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods (SEFM'12)*, volume 7504 of *LNCS*, pages 233–247. Springer-Verlag, 2012.

[61] Dennis Dams and Kedar S. Namjoshi. "Shape Analysis through Predicate Abstraction and Model Checking". In *Proceedings of the 4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'03)*, pages 310–324, 2003.

[62] Frank S. de Boer, Marcello M. Bonsangue, and Jurriaan Rot. "Automated Verification of Recursive Programs with Pointers". In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR'06)*, volume 7364 of *LNCS*, pages 149–163. Springer-Verlag, 2012.

[63] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.

[64] Nachum Dershowitz, Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. "A General Framework for Automatic Termination Analysis of Logic Programs". *Applicable Algebra in Engineering, Communication and Computing*, 12(1/2):117–156, 2001.

[65] David Detlefs, Greg Nelson, and James B. Saxe. "Simplify: A Theorem Prover for Program Checking". *Journal of the ACM*, 52(3):365–473, 2005.

[66] Edsger W. Dijkstra and Carel S. Scholten. "Termination Detection for Diffusing Computations". *Information Processing Letters*, 11(1):1–4, 1980.

[67] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. "Concurrent Abstract Predicates". In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*, volume 6183 of *LNCS*, pages 504–528. Springer-Verlag, 2010.

[68] Kamil Dudka, Petr Peringer, and Tomás Vojnar. "Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic". In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806, pages 372–378. Springer-Verlag, 2011.

[69] Bruno Dutertre. "Yices 2.2". In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV'14)*, volume 8559 of *LNCS*, pages 737–744. Springer-Verlag, 2014.

[70] E. Allen Emerson and Vineet Kahlon. "Reducing Model Checking of the Many to the Few". In *Proceedings of the 17th International Conference on Automated Deduction (CADE'00)*, volume 1831 of *LNAI*, pages 236–254. Springer-Verlag, 2000.

[71] E. Allen Emerson and Vineet Kahlon. "Model Checking Large-Scale and Parameterized Resource Allocation Systems". In *TACAS*, volume 2280 of *LNCS*, pages 251–265. Springer-Verlag, 2002.

[72] E. Allen Emerson and Kedar S. Namjoshi. "Reasoning about Rings". In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 85–94. ACM, 1995.

[73] E. Allen Emerson and Kedar S. Namjoshi. "Automatic Verification of Parameterized Synchronous Systems". In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 87–98. Springer-Verlag, 1996.

[74] E. Allen Emerson and A. Prasad Sistla. "Symmetry and Model Checking". *Formal Methods in System Design*, 9(1/2):105–131, 1996.

[75] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. "Quickly Detecting Relevant Program Invariants". In *Proceedings of the 22nd ACM/IEEE International Conference on Software Engineering (ICSE'00)*, pages 449–458. ACM, 2000.

[76] Azadeh Farzan and Zachary Kincaid. "Verification of Parameterized Concurrent Programs by Modular Reasoning about Data and Control". In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 297–308. ACM, 2012.

[77] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and Natarajan Shankar. "ICS: Integrated Canonizer and Solver". In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 246–249. Springer-Verlag, 2001.

[78] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 - Where Programs Meet Provers". In *Proceedings of the 22nd European Symposium on Programming (ESOP'13)*, volume 7792 of *LNCS*, pages 125–128. Springer-Verlag, 2013.

[79] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. "Impossibility of Distributed Consensus with One Faulty Process". *Journal of the ACM*, 32(2):374–382, 1985.

[80] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. "Theorem Proving Using Lazy Proof Explication". In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 355–367. Springer-Verlag, 2003.

[81] Cormac Flanagan and Shaz Qadeer. "Thread-Modular Model Checking". In *Proceedings of the 10th International SPIN Workshop on Model Checking Software (SPIN'03)*, pages 213–224, 2003.

[82] Juan Pablo Galeotti, Carlo A. Furia, Eva May, Gordon Fraser, and Andreas Zeller. "DynaMate: Dynamically Inferring Loop Invariants for Automatic Full Functional Verification". In *Proceedings of the 10th International Haifa Verification Conference (HVC'14)*, volume 8855 of *LNCS*, pages 48–53. Springer-Verlag, 2014.

[83] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row, 1986.

[84] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Alberto Oliveras, and Cesare Tinelli. "DPLL(T): Fast Decision Procedures". In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 175–188. Springer-Verlag, 2004.

[85] Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou. "Static Analysis by Policy Iteration on Relational Domains". In *Proceedings of the 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 237–252. Springer-Verlag, 2007.

[86] Thomas Gawlitza and Helmut Seidl. "Precise Fixpoint Computation Through Strategy Iteration". In *Proceedings of the 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 300–315. Springer-Verlag, 2007.

[87] Steven M. German and A. Prasad Sistla. "Reasoning About Systems with Many Processes". *Journal of the ACM*, 39(3):675–735, 1992.

[88] Silvio Ghilardi. "Reasoners' Cooperation and Quantifier Elimination". In *Technical report*. Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, 2003.

[89] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. "Proving Termination of Programs Automatically with AProVE". In *Proceedings of the 7th International Joint Conference on Automated Reasoning (IJCAR'08)*, volume 8562 of *LNCS*, pages 184–191. Springer-Verlag, 2014.

[90] Amit Goel, Sava Krstic, Rebekah Leslie, and Mark R. Tuttle. "SMT-Based System Verification with DVF". In *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories (SMT'12)*, volume 20 of *EPiC Series*, pages 32–43. EasyChair, 2012.

[91] James R. Goodman, Mary K. Vernon, and Philip J. Woest. "Efficent Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors". In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'89)*, pages 64–75, 1989.

[92] Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. "Proving That Non-blocking Algorithms Don't Block". In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, pages 16–28. ACM, 2009.

[93] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer". *IEEE Transactions on Computers*, 32(2):175–189, 1983.

[94] Gary Graunke and Shreekant S. Thakkar. "Synchronization Algorithms for Shared-Memory Multiprocessors". *IEEE Computer*, 23(6):60–69, 1990.

[95] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. "Synthesizing Software Verifiers from Proof Rules". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, 2012.

[96] Lindsay Groves. "Verifying Michael and Scott's Lock-Free Queue Algorithm Using Trace Reduction". In *CATS*, volume 77 of *CRPIT*, pages 133–142. Australian Computer Society, 2008.

[97] Christian Haack and Clément Hurlin. "Separation Logic Contracts for a Java-like Language with Fork/Join". In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology (AMAST'08)*, volume 5140 of *LNCS*, pages 199–215. Springer-Verlag, 2008.

[98] Timothy L. Harris and Keir Fraser. "Language Support for Lightweight Transactions". In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems (OOPSLA'03)*, pages 388–402, 2003.

[99] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. "A Practical Multi-word Compare-and-Swap Operation". In *Proceedings of the 16th International Conference on Distributed Computing (DISC'02)*, pages 265–279, 2002.

[100] Debra Hensgen, Raphael Finkel, and Udi Manber. "Two Algorithms for Barrier Synchronization". *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[101] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. "Thread-Modular Abstraction Refinement". In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 262–274. Springer-Verlag, 2003.

[102] Maurice Herlihy. "Wait-Free Synchronization". *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[103] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. "Scalable Concurrent Counting". *ACM Transactions on Computer Systems*, 13(4):343–364, 1995.

[104] Maurice Herlihy, Victor Luchangco, and Mark Moir. "Obstruction-Free Synchronization: Double-Ended Queues as an Example". In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, pages 522–529. IEEE Computer Society Press, 2003.

[105] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. "Software Transactional Memory for Dynamic-sized Data Structures". In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC'03)*, pages 92–101, 2003.

[106] Maurice Herlihy and J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures". In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, pages 289–300, 1993.

[107] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgran-Kaufmann, 2008.

[108] Maurice Herlihy and Jeannette M. Wing. "Axioms for Concurrent Objects". In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL'87)*, pages 13–26. ACM, 1987.

[109] Maurice Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[110] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. "Oracle Semantics for Concurrent Separation Logic". In *Proceedings of the 17th European Symposium on Programming (ESOP'08)*, volume 4960 of *LNCS*, pages 353–367. Springer-Verlag, 2008.

[111] Krystof Hoder and Nikolaj Bjørner. "Generalized Property Directed Reachability". In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, volume 7317 of *LNCS*, pages 157–171. Springer-Verlag, 2012.

[112] Lukás Holík, Ondrej Lengál, Adam Rogalewicz, Jirí Simácek, and Tomás Vojnar. "Fully Automated Shape Analysis Based on Forest Automata". In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*, volume 8044, pages 740–755. Springer-Verlag, 2013.

[113] IBM. "System/370 Principles of Operation. Order Number GA22-7000".

[114] IBM. "PowerPC Microprocessor Family: Programming Environments Manual for 64 and 32-bit Microprocessors, version 2.0", 2003.

[115] Eugene D. Brooks III. "The Butterfly Barrier". *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[116] Intel. "Pentium Processor Family User's Manual: Vol 3, Architecture and Programming Manual", 1994.

[117] Radu Iosif, Adam Rogalewicz, and Jirí Simácek. "The Tree Width of Separation Logic with Recursive Definitions". In *Proceedings of the 24th International Conference on Automated Deduction (CADE'13)*, volume 7898 of *LNCS*, pages 21–38. Springer-Verlag, 2013.

[118] Bertrand Jeannet and Antoine Miné. “Apron: A Library of Numerical Abstract Domains for Static Analysis”. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV’09)*, volume 5643 of *LNCS*, pages 661–667. Springer-Verlag, 2009.

[119] Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, and Michael I. Schwartzbach. “Automatic Verification of Pointer Programs Using Monadic Second-Order Logic”. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’97)*, pages 226–236, 1997.

[120] Yuh-Jzer Joung. “Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors”. *Distributed Computing*, 15(3):155–175, 2002.

[121] Vineet Kahlon, Sriram Sankaranarayanan, and Aarti Gupta. “Semantic Reduction of Thread Interleavings in Concurrent Programs”. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’09)*, volume 5505 of *LNCS*, pages 124–138. Springer-Verlag, 2009.

[122] G. Kane. *MIPS RISC Architecture*. Prentice-Hall, Inc., New York, USA, 1989.

[123] `http://api.kde.org/4.1-api/kdeedu-apidocs/kstars/html/SkipList_8cpp_source.html`.

[124] Patrick Keane and Mark Moir. “A Simple Local-Spin Group Mutual Exclusion Algorithm”. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC’99)*, pages 23–32, 1999.

[125] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. “Frama-C: A software analysis perspective”. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[126] Orran Krieger, Michael Stumm, Ronald C. Unrau, and Jonathan Hanna. “A Fair Fast Scalable Reader-Writer Lock”. In *Proceedings of the 22nd International Conference on Parallel Processing (ICPP’93)*, pages 201–204, 1993.

[127] Saul Kripke. “Semantical Considerations on Modal Logic”. *Journal of Symbolic Logic*, 34, 1969.

[128] Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. “An Algorithm for Deciding BAPA: Boolean Algebra with Presburger Arithmetic”. In *Proceedings of the 20th International Conference on Automated Deduction (CADE’05)*, pages 260–277. Springer-Verlag, 2005.

[129] Viktor Kuncak and Martin C. Rinard. “An Overview of the Jahob Analysis System: Project Goals and Current Status”. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS’06)*, IEEE Computer Society Press, 2006.

[130] H. T. Kung and John T. Robinson. “On Optimistic Methods for Concurrency Control”. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[131] Shuvendu K. Lahiri and Shaz Qadeer. “Verifying Properties of Well-founded Linked Lists”. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’06)*, pages 115–126. ACM, 2006.

[132] Shuvendu K. Lahiri and Shaz Qadeer. "Back to the Future: Revisiting Precise Program Verification Using SMT Solvers". In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 171–182. ACM, 2008.

[133] Leslie Lamport. "A New Solution of Dijkstra's Concurrent Programming Problem". *Commun. ACM*, 17(8):453–455, 1974.

[134] Leslie Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[135] K. Rustan M. Leino. "Verifying Concurrent Programs with Chalice". In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI'10)*, volume 5944 of *LNCS*. Springer-Verlag, 2010.

[136] K. Rustan M. Leino and Peter Müller. "A Basis for Verifying Multi-threaded Programs". In *Proceedings of the 18th European Symposium on Programming (ESOP'09)*, volume 5502 of *LNCS*, pages 378–393. Springer-Verlag, 2009.

[137] David Lesens, Nicolas Halbwachs, and Pascal Raymond. "Automatic Verification of Parameterized Linear Networks of Processes". In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 346–357. ACM, 1997.

[138] M. Loui and H. Abu-Amara. "Memory Requirements for Agreement Among Unreliable Asynchronous Processes". *Advances in Computing Research*, 4:163–183, 1987.

[139] P. Lucas. "Two Constructive Realizations of the Block Concept and Their Equivalence". In *Technical report*. IBM Laboratory, Vienna, 1968.

[140] P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. "Decidable Logics Combining Heap Structures and Data". In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*, pages 611–622. ACM, 2011.

[141] Peter S. Magnusson, Anders Landin, and Erik Hagersten. "Queue Locks on Cache Coherent Multiprocessors". In *Proceedings of the 8th International Symposium on Parallel Processing (IPPS'94)*, pages 165–171. IEEE Computer Society Press, 1994.

[142] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. "Precise Thread-Modular Verification". In *Proceedings of the 14th International Static Analysis Symposium (SAS'07)*, volume 4634 of *LNCS*, pages 218–232. Springer-Verlag, 2007.

[143] Zohar Manna, Anuchit Anuchitanukul, Nikølaj Bjørner, Anca Browne, Edward Chang, Michael Colón, Luca de Alfaro, Harish Devarajan, Henny Sipma, and Tomás Uribe. "STeP: The Stanford Temporal Prover". Technical Report STAN-CS-TR-94-1518, Computer Science Department, Stanford University, July 1994.

[144] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems*. Springer-Verlag, 1995.

[145] Zohar Manna and Henny Sipma. "Verification of Parameterized Systems by Dynamic Induction on Diagrams". In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*. Springer-Verlag, 1999.

[146] Maria Manzano. *Extensions of First Order Logic*. Cambridge University Press, 1996.

[147] Giorgio Delzanno Marco Bozzano. "Beyond Parameterized Verification". In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 221–235. Springer-Verlag, 2002.

[148] Kenneth L. McMillan and Lenore D. Zuck. "Invisible Invariants and Abstract Interpretation". In *Proceedings of the 18th International Static Analysis Symposium (SAS'11)*, volume 6887 of *LNCS*, pages 249–262. Springer-Verlag, 2011.

[149] Scott McPeak and George C. Necula. "Data Structure Specifications via Local Equality Axioms". In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, pages 476–490. Springer-Verlag, 2005.

[150] John M. Mellor-Crummey and Michael L. Scott. "Barriers for the Sequent Symmetry". `ftp://ftp.cs.rochester.edu/pub/packages/scalable_synch/locks_and_barriers/Symmetry.tar.Z`.

[151] John M. Mellor-Crummey and Michael L. Scott. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors". *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[152] John M. Mellor-Crummey and Michael L. Scott. "Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors". In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'91)*, pages 106–113, 1991.

[153] Maged M. Michael and Michael L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms". In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 267–275, 1996.

[154] Maged M. Michael and Michael L. Scott. "Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors". *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[155] Antoine Miné. "Static Analysis of Run-Time Errors in Embedded Critical Parallel C Programs". In *Proceedings of the 20th European Symposium on Programming (ESOP'11)*, volume 6602 of *LNCS*, pages 398–418, 2011.

[156] Mark Moir and Nir Shavit. "Concurrent Data Structures". In *Handbook of Data Structures and Applications*, pages 47–14 — 47–30, 2007. Chapman and Hall/CRC Press.

[157] Anders Møller and Michael I. Schwartzbach. "The Pointer Assertion Logic Engine". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 221–231. ACM, 2001.

[158] Kedar S. Namjoshi. "Symmetry and Completeness in the Analysis of Parameterized Systems". In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'07)*, volume 4349 of *LNCS*, pages 299–313. Springer-Verlag, 2007.

[159] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. "Communicating State Transition Systems for Fine-Grained Concurrent Resources". In *Proceedings of the 23rd European Symposium on Programming (ESOP'14)*, volume 8410 of *LNCS*, pages 290–310. Springer-Verlag, 2014.

[160] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs". In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 213–228. Springer-Verlag, 2002.

[161] Greg Nelson. "Verifying Reachability Invariants of Linked Structures". In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages (POPL'83)*, pages 38–47. ACM, 1983.

[162] Greg Nelson and Derek C. Oppen. "Simplification by Cooperating Decision Procedures". *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[163] Huu Hai Nguyen and Wei-Ngan Chin. "Enhancing Program Verification with Lemmas". In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV'08)*, volume 5123, pages 355–369. Springer-Verlag, 2008.

[164] Peter W. O'Hearn. "Resources, Concurrency and Local Reasoning". In *Proceedings of the 15th International Conference in Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 49–67. Springer-Verlag, 2004.

[165] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL'01)*, volume 2142 of *LNCS*, pages 1–19. Springer-Verlag, 2001.

[166] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. "Variables as Resource in Hoare Logics". In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 137–146. IEEE Computer Society Press, 2006.

[167] Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. "Modular Verification of a Non-blocking Stack". In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 297–302. ACM, 2007.

[168] Amir Pnueli. "The Temporal Logic of Programs". In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS'77),*, pages 46–57. IEEE Computer Society Press, 1977.

[169] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. "Automatic Deductive Verification with Invisible Invariants". In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 82–97. Springer-Verlag, 2001.

[170] Andreas Podelsky and Andrey Rybalchenko. "Transition Invariants". In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41. IEEE Computer Society Press, 2004.

[171] William Pugh. “Skip Lists: A Probabilistic Alternative to Balanced Trees”. *Communications of the ACM*, 33(6):668–676, 1990.

[172] Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and P. Madhusudan. “Natural Proofs for Structure, Data, and Separation”. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’13)*, pages 231–242. ACM, 2013.

[173] Azalea Raad, Jules Villard, and Philippa Gardner. “CoLoSL: Concurrent Local Subjective Logic”. In *Proceedings of the 24th European Symposium on Programming (ESOP’15)*, volume 9032 of *LNCS*, pages 710–735. Springer-Verlag, 2015.

[174] Ravi Rajwar and James R. Goodman. “Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution”. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO’01)*, pages 294–305. ACM/IEEE Computer Society, 2001.

[175] Ravi Rajwar and James R. Goodman. “Transactional Lock-free Execution of Lock-based Programs”. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’02)*, pages 5–17, 2002.

[176] Zvonimir Rakamaric, Jesse D. Bingham, and Alan J. Hu. “An Inference-Rule-Based Decision Procedure for Verification of Heap-Manipulating Programs with Mutable Data and Cyclic Data Structures”. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’07)*, volume 4349 of *LNCS*, pages 106–121. Springer-Verlag, 2007.

[177] Silvio Ranise, Christophe Ringeissen, and Calogero G. Zarba. “Combining Data Structures with Nonstably Infinite Theories Using Many-Sorted Logic”. In *Proceedings of the 5th International Workshop on Frontiers of Combining Systems (FroCoS’05)*, pages 48–64. Springer-Verlag, 2005.

[178] Silvio Ranise and Calogero G. Zarba. “A Theory of Singly-Linked Lists and its Extensible Decision Procedure”. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006)*, pages 206–215. IEEE Computer Society Press, 2006.

[179] Silvio Ranise and Calogero G. Zarba. “A Theory of Singly-Linked Lists and its Extensible Decision Procedure”. In *Technical report*, 2006.

[180] John C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS’02)*, pages 55–74. IEEE Computer Society Press, 2002.

[181] Harald Rueß and Natarajan Shankar. “Deconstructing Shostak”. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS’01)*, pages 19–28. IEEE Computer Society Press, 2001.

[182] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. “Solving Shape-Analysis Problems in Languages with Destructive Updating”. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.

[183] César Sánchez. *Deadlock Avoidance for Distributed Real-time and Embedded Systems*. PhD thesis, Computer Science Department, Stanford University, May 2007.

[184] Michael L. Scott. "Non-blocking Timeout in Scalable Queue-based Spin Locks". In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC'02)*, pages 31–40, 2002.

[185] Michael L. Scott and William N. Scherer III. "Scalable Queue-based Spin Locks with Timeout". In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 44–52, 2001.

[186] Michael L. Scott and John M. Mellor-Crummey. "Fast, Contention-Free Combining Tree Barriers". Technical report, University of Rochester, Rochester, NY, USA, 1992.

[187] Nir Shavit and Dan Touitou. "Software Transactional Memory". *Distributed Computing*, 10(2):99–116, 1997.

[188] Robert E. Shostak. "Deciding Combinations of Theories". *Journal of the ACM*, 31(1):1–12, 1984.

[189] Henny B. Sipma. *Diagram-Based Verification of Discrete, Real-Time and Hybrid Systems*. PhD thesis, Stanford University, 1999.

[190] R. Sites. "Alpha Architecture Reference Manual", 1992.

[191] Jan Smans, Bart Jacobs, and Frank Piessens. "VeriCool: An Automatic Verifier for a Concurrent Object-Oriented Language". In *Proceedings of the 10th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'08)*, volume 5051 of *LNCS*, pages 220–239, 2008.

[192] Viorica Sofronie-Stokkermans. "Hierarchic Reasoning in Local Theory Extensions". In *Proceedings of the 20th International Conference on Automated Deduction (CADE'05)*, volume 3632 of *LNCS*, pages 219–234. Springer-Verlag, 2005.

[193] Viorica Sofronie-Stokkermans. "On Combinations of Local Theory Extensions". In *Programming Logics - Essays in Memory of Harald Ganzinger*, volume 7797 of *LNCS*, pages 392–413. Springer-Verlag, 2013.

[194] CORPORATE SPARC International, Inc. *The SPARC Architecture Manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[195] Ichiro Suzuki. "Proving Properties of a Ring of Finite-state Machines". *Information Processing Letters*, 28:213–214, 1988.

[196] Kasper Svendsen and Lars Birkedal. "Impredicative Concurrent Abstract Predicates". In *Proceedings of the 23rd European Symposium on Programming (ESOP'14)*, volume 8410 of *LNCS*, pages 149–168. Springer-Verlag, 2014.

[197] `http://kde.org/`.

[198] `https://ocaml.org/`.

[199] Cesare Tinelli and Calogero G. Zarba. "Combining Decision Procedures for Sorted Theories". In *JELIA'04*, volume 3229 of *LNCS*, pages 641–653. Springer-Verlag, 2004.

[200] Cesare Tinelli and Calogero G. Zarba. "Combining Nonstably Infinite Theories". *Journal of Automated Reasoning*, 34:209–238, 2005.

[201] Nishant Totla and Thomas Wies. "Complete Instantiation-Based Interpolation". In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*, pages 537–548. ACM, 2013.

[202] Aaron Turon, Derek Dreyer, and Lars Birkedal. "Unifying Refinement and Hoare-style Reasoning in a Logic for Higher-order Concurrency". In *Proceedings of the International Conference on Functional Programming (ICFP'13)*, pages 377–390. ACM, 2013.

[203] Viktor Vafeiadis. *Modular Fine-grained Concurrency Verification*. PhD thesis, University of Cambridge, 2007.

[204] Viktor Vafeiadis. "Shape-Value Abstraction for Verifying Linearizability". In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'09)*, volume 5403 of *LNCS*, pages 335–348. Springer-Verlag, 2009.

[205] Viktor Vafeiadis. "Automatically Proving Linearizability". In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*, volume 6174 of *LNCS*, pages 450–464. Springer-Verlag, 2010.

[206] Viktor Vafeiadis. "RGSep Action Inference". In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI'10)*, volume 5944 of *LNCS*, pages 345–361. Springer-Verlag, 2010.

[207] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. "Proving Correctness of Highly-Concurrent Linearisable Objects". In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, pages 129–136. ACM, 2006.

[208] Viktor Vafeiadis and Matthew J. Parkinson. "A Marriage of Rely/Guarantee and Separation Logic". In *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR'2007)*, volume 4703 of *LNCS*, pages 256–271. Springer-Verlag, 2007.

[209] Thomas Wies, Viktor Kuncak, Karen Zee, Andreas Podelski, and Martin Rinard. "Verifying Complex Properties Using Symbolic Shape Analysis". In *Workshop on heap abstraction and verification (collocated with ETAPS)*, 2007.

[210] Thomas Wies, Ruzica Piskac, and Viktor Kuncak. "Combining Theories with Shared Set Operations". In *Proceedings of the 7th International Symposium on the Frontiers of Combining Systems (FroCoS'09)*, volume 5749 of *LNCS*, pages 366–382. Springer-Verlag, 2009.

[211] Eran Yahav. "Verifying Safety Properties of Concurrent Java Programs Using 3-valued Logic". In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 27–40. ACM, 2001.

[212] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors". *IEEE Trans. Computers*, 36(4):388–395, 1987.

[213] Greta Yorsh, Alexander Moshe Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. "A Logic of Reachable Patterns in Linked Data-Structures". In *Proceedings of the 9th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'06)*, pages 94–110, 2006.

[214] Lenore D. Zuck and Amir Pnueli. "Model Checking and Abstraction to the Aid of Parameterized Systems (a survey)". *Computer Languages, Systems & Structures*, 30:139–169, 2004.

Index

$:=$ (SPL assignment), 19
α (concretization), 47
$\circ$ (path concatenation), 131
$\rightarrow$ (SPL pointer access), 19
$\varphi(\{\cdots\})$ (parametrized formula), 45
$\triangleright$ (support operator), 59
$\oslash$ (no thread), 18
--> (LEAP PVD edges), 241
-> (LEAP sequential operator), 239
<<>> (LEAP ranking function), 241
=> (LEAP concurrent operator), 239
@ (ghost field), 26
$[M]$, 41
$m_{a \mapsto c}$ (memory map), 159
$Mod^{\Sigma}(\Phi)$ (class of many-sorted structures), 36
:: (local variables), 18
$ (LEAP ghost code), 228
{ } (LEAP atomic section), 228
(LEAP formula labeling), 230

acceptance, 320
accepting path (PVD), 78
$Addr$, 18
always ($\square$), 38
$Array\text{h}AB$, 18
assertion map, 92
atomic section ($\text{h}A \cdots B$), 20
await , 20

backoff, 30
Bad (LEAP ranking function), 241
barriers, 34
B-INV, 55

blocking algorithms, 29
blocking synchronization methods, 29
$Bool$, 18
bounded model property, 37
$BoundedSkiplistNode$, 146
boxes, 76
BP-INV, 57

cache-coherent, 29
call, 20
CAS, *see* Compare-and-swap
classes, 26
clause, 46
coarse-grained locking, 29
theory combination ($\oplus$), 37
combining trees, 30
$common$, 132
compare-and-swap, 31
$compress$, 131
computation, 40
computation (PVD), 78
conjunctive normal form, 46
consecution, 320
counting abstraction, 274
critical, 20
critical section, 28
set of constants C_{σ}, 36
cut-off, 231
  DNF, 231
  Pruning, 232
  Union, 233

deadlock, 30

diseq, 131
disjunctive normal form, 46
DNF, 231

eager
  see reflective abstraction, 100
eager+
  see reflective abstraction, 102
Elem, 18
En(), 40
En, 81
enabling condition, 40
η^* (concrete collecting semantics), 98
eventually ($\diamond$), 38
exponential backoff, 34

Facts, 244
fair path (PVD), 78
fairness, 320
filter-strict, 237
finite model property, 37
first, 130
$first_{\mathsf{K}}$, 170
flat literal, 36
for, 20
F_{param} (parametrized formulas), 44
$F_{\Sigma}^{()}$ (set of $\Sigma - formula$), 36
$F_{\Sigma}^{(V)}$ (set of Σ-formula over V), 37
full, 235

gap (TSL), 203
gap-less model (TSL), 203
generalized verification diagram, 71
ghost, 225
ghost (LEAP ghost code), 228
ghost code, 26
ghost field, 26
ghost variables, 26
G-INV, 65
Good (LEAP ranking function), 241
granularity, 29
group mutual exclusion, 34
GVD, 71

hand-in-hand, 148
heap, 22

i-index formula, 44
identity, 235
if, 19
inductive invariant, 55
Init, 79
initiation, 320
Int, 18
interference semantics, 100
INV, 55

lazy
  see reflective abstraction, 99
LEAP, 223
  cut-off, 231
  tactics, 234
linear temporal logic, 38
linearizability, 32
linearization point, 33
ListNode, 106
LL/SC, *see* Load-linked/store-conditional
$\mathcal{L}^{[M]}(\mathcal{D})$ (set of PVD computations), 78
load-linked/store-conditional, 31
Loc, 41
local spinning, 30
Lock, 18
lock, 34
lock elision, 35
lock-coupling lists, 106
lock-freedom, 31
Locs, 41
LTL, *see* Lnear temporal logic38

main (LEAP main procedure), 225
materialized threads (invariant generation), 89
me, 18
memory contention, 29
MIRROR, 94
model-checking, 33
ModelCheck, 81

next ($\bigcirc$), 38
non-blocking algorithms, 29
non-blocking synchronization methods, 31
noncritical, 20
nondet choice, 20

obstruction-freedom, 31
offending transition, 58
optimistic concurrency control, 35
OtherAcc, 80
OtherConsec, 80

parametrized invariance, 53
parametrized System, 41
parametrized verification diagram, 76
path (PVD), 78
pc (program counter), 19
pc_P (procedure-based program counter), 19
permitted edges (PVD), 80
P-INV, 60
$\mathcal{P}_P$ (parametrized fair transition system), 44
$pres$, 40
program counter, *see* 19
program labeling, 228
proof graph, 65
Prop, 81
propagate-disj-conseq-fst, 237
propagate-disj-conseq-snd, 237
propositional-propagate, 237
Pruning, 232
PVD (parametrized verification diagram), 76

queuelocks, 34
quiescent consistency, 33

reader-writer locks, 34
reduce, 235
reduce2, 235
REFLECT$_{\mathcal{P}}(\eta)$ (reflective abstraction), 94
reflective abstraction, 91, 94
    eager, 100
    eager+, 102
    lazy, 99
release ($\mathcal{R}$), 38
return, 20
room synchronization, 34
run, 40

$\mathcal{S}$ (non-parametrized fair transition system, 39
scalability, 29
SelfAcc, 80
SelfConsec, 80
sequential consistency, 32
serializability, 32
$Set\mathrm{h}AB$, 18
Σ-formula, 36
Σ-interpretation, 36
Σ-literal, 36
Σ-structure, 36
Σ-term, 36
Σ-theory, 36
Σ (signature), 35
Simplified Programming Language (SPL), 17
simplify-pc, 236
simplify-pc-plus, 237
skip, 19
skiplist, 144
skiplist (unbounded), 182
speedup, 29
spinlocks, 34
spinning, 30
SP-INV, 64
SPL
    assignment, 19
    atomic sections, 20
    await, 20
    begin, 18
    conditionals, 19
    critical section, 20
    for, 20
    global, 17
    loops, 19
    no operation, 19
    non critical section, 20
    non deterministic choice, 20
    pointer access, 19
    procedure, 17
    procedure call, 20
    procedure return, 20
split-antecedent-pc, 236
split-consequent, 236
split-goal, 234
$\{\mathcal{S}_P[M]\}$ (instance family), 41
$state$, 40
state, 40
$\mathbb{S}$, 40
Succ, 81

support, 59

Tactics
    propagate-disj-conseq-fst, 237
tactics, 234
    filter-strict, 237
    full, 235
    identity, 235
    propagate-disj-conseq-snd, 237
    propositional-propagate, 237
    reduce, 235
    reduce2, 235
    simplify-pc, 236
    simplify-pc-plus, 237
    split-antecedent-pc, 236
    split-consequent, 236
    split-goal, 234
`Tactics`, 244
$\mathcal{T}_{\mathsf{param}}$ (parametrized transitions), 45
temporal parametrized formulas, 45
theory combination, 37
Θ_{param} (parametrized initial condition), 45
Tid, 18
TL3, 119
trail (PVD), 78
TSL, 189
$\mathsf{TSL_K}$, 154

unbounded lock-free queue, 115
unbounded lock-free stack, 113
unbounded queue, 111
UnboundedQueueNode, 111
UnboundedSkiplistNode, 182
uniform verification problem, 54
Union, 233
unordered, 171
until ($\mathcal{U}$), 38

$\mathbb{V}$ (invariant generation valuation map), 93
variable symmetry (LEAP), 231
$V_\sigma^{\mathcal{A}}$ (set of variable interpretations), 36
V_{box} (set of box parameters), 76
V_{global} (set of global variables), 41
V_{local} (set of local variables), 41
Voc (formula vocabulary), 46
V_{param} (parametrized variables), 44
set of variables (V_σ), 36
$V_{tidParams}$, 76

wait for ($\mathcal{W}$), 38
wait-freedom, 31
while, 19
widening, 99

ξ (invariant generation state), 93

Part IV

Appendices

A

Checking that a PVD *Satisfies a Temporal Property*

We show here how to check that $\mathcal{L}_p^{[M]}(\mathcal{D}) \subseteq \mathcal{L}^{[M]}(\varphi)$, for a temporal property φ. The first step is extracting the propositional alphabet from φ. Then, the main idea is to construct two non-deterministic Büchi automata NBW: $\mathcal{A}_\mathcal{D}$ that captures the propositional computations of the diagram, and $\mathcal{A}_{\neg\varphi}$ for the negation of the property (obtained by classical constructions). Both automata use the propositional alphabet of the property. Then, an algorithmic check for emptiness: $\mathcal{A}_\mathcal{D} \times \mathcal{A}_{\neg\varphi} = \emptyset$ allows to decide whether $\mathcal{L}_p(\mathcal{D}) \subseteq \mathcal{L}_p(\varphi)$.

We now show how to build an NBW for the propositional traces of the diagram.

A.1 The Intended Meaning of $\mathcal{F}$

The intended meaning of each edge Streett condition $\langle B_i, G_i, \delta_i \rangle$ is to ensure that in any accepting trail of the diagram either some edge from G_i is visited infinitely often, or all edges from B_i are visited finitely often. The verification conditions (SelfAcc) and (OtherAcc) show that the ranking function δ_i is (strictly) decreasing in B_i edges, and non-increasing in P_i edges (that is, $E - (G_i \cup B_i)$ edges).

Without loss of generality we assume that $G_i \cap B_i = \emptyset$. The reason is that, otherwise the pair $\langle G_i, B_i' \rangle$—where $B_i' = B_i \setminus G_i$—satisfies that

- $G_i \cap B_i' = \emptyset$, and
- every trail π is accepting for (G_i, B_i) precisely when it is accepting for (G_i, B_i'). To see this, observe that a good trail that traverses G_i edges infinitely often is accepted for both. Also, if a trail visits all B_i edges only finitely often,then it visits all B_i' finitely often. For the other direction, consider the cases:

  1. If a trail visits all edges in B_i' only finitely often but some edge in $B_i \setminus B_i'$ infinitely often, then some edge in G_i is seen infinitely (because all edges in $B_i \setminus B_i'$ are G_i edges), and the trail is accepting in both cases again.

2. The last case is that all edges in B'_i are traversed only finitely and alls edge in $B_i \setminus B'_i$ are also traversed only finitely often, which again is accepting for both

A.2 Edge Streett Automaton on Words

We define here an edge-Streett non-deterministic automaton on words (ESNW for short) as a tuple $\langle AP, Q, Q_0, L, T, F\rangle$, where:

- AP is a finite set of propositions.
- Q is a finite set of states.
- $Q_0 \subseteq Q$ is the initial set of states.
- L is a map $L : Q \to 2^{AP}$ assigning predicates from AP to states.
- $T \subseteq Q \times Q$ is a transition function.
- F is an edge-Streett acceptance condition, $F = \langle\langle B_1, G_1\rangle, \ldots, \langle B_k, G_k\rangle\rangle$ described by a finite collection of pairs, where $B_j, G_j \subseteq T$ are sets of edges.

A *trace* of an ESNW is an infinite sequence $s_0 t_0 s_1 t_1 \ldots$ of states and transitions where for each position i, $(s_i, s_{i+1}) \in t_i$. The set of edges from T seen infinitely often in a trace π is denoted by $\mathit{inf}_T(\pi)$.

A trace π of an ESNW is *accepting* whenever for all $1 \leq j \leq k$, either:

- $\mathit{inf}_T(\pi) \cap G_j \neq \emptyset$, or
- $\mathit{inf}_T(\pi) \cap B_j = \emptyset$.

A.3 From ESNW into NBW

We show now how to translate an ESNW into an NBW that accepts the same language. Our definition of NBW differs a bit from the common in the literature, so we introduce it here. An NBW is a tuple $\langle AP, Q, Q_0, L, T, F\rangle$, where AP, Q, Q_0, L and T are like in ESNW. The termination condition F is a subset of Q. A trace is accepting whenever $\mathit{inf}_Q(\pi) \cap F \neq \emptyset$.

The translation works as follow. Given an ESNW $\mathcal{E}$ we first generate an NBW $\mathcal{A}_j$ for each edge Streett pair (B_j, G_j) separately. $\mathcal{A}_j : \langle AP, Q^j, I^j, L^j, T^j, F^j\rangle$ is as follows. $\mathcal{A}_j$ is essentially composed of two sub-automata that we describe next: $Q^j = Q_1 \cup Q_2$ and $T^j = T_1 \cup T_2$ and $F^j = F_1 \cup F_2$, $I^j = I_1 \cup I_2$. We describe the two sub-automata separately:

- The set of states Q_1 contains two copies q_1^G and q_1 for each state q in Q. Essentially, q_1^G encodes that a good edge has just been taken, while q_1 encodes that a good edge was not taken to reach q.
- For edges:
  - For every good edge $p \to q$ in $T \cap G_j$ we add an edge $p_1 \to q_1^G$ and an edge $p_1^G \to q_1^G$ into T_1.

  - For every non-good edge $p \to q$ we add an edge $p_1 \to q_1$ and an edge $p_1^G \to q_1$ into T_1.

- The accepting states are $F_1 = \{q_1^G\}$.

- $I_1 = \{q_1 \mid q \in I\}$.

- $L_1(q_1) = L(q)$, and $L_1(q_1^G) = L(q)$.

Accepting traces in $\mathcal{A}_1$ must visit q_1^G states infinitely often. Since incoming edges to these states are G edges, then good edges must be traversed infinitely often in the corresponding trace in $\mathcal{E}$. The other direction holds similarly.

The second component of $\mathcal{A}^j$ is $\mathcal{A}_2 : \langle Q_2, I_2, L_2, T_2, F_2 \rangle$ described as follows:

- Q_2 contains one state q for each state q in Q.

- $I_2 = \emptyset$.

- $L_2(q_2) = L(q)$.

- $F_2 = Q_2$: all states are accepting.

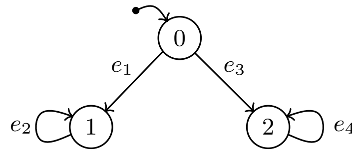
- T_2 contains an edge $p_2 \to q_2$ whenever there is a non-B edge $p \to q$ in T. Additionally, we add one edge $p_1 \to q_2$ for every edge $p \to q$ (good or bad) in T; these additional edges allow to jump from $\mathcal{A}_1$ into $\mathcal{A}_2$.

Note that there are not transitions back into $\mathcal{A}_1$ from $\mathcal{A}_2$, so if one of the jump transitions in a trace is traversed, the trace stays in $\mathcal{A}_2$. Then, since all states in $\mathcal{A}_2$ are Büchi accepting and all B edges are removed in $\mathcal{A}_2$, the trace that gets trapped in $\mathcal{A}_2$ corresponds to a trace in $\mathcal{E}$ that only traverses B edges finitely often. Conversely, a trace in $\mathcal{E}$ that only traverses B edges finitely often, will—at some finite point—not traverse B edges any longer. Then, $\mathcal{E}$ can jump at that point in time to $\mathcal{A}_2$, and will be able to simulate the rest of trace in $\mathcal{A}_2$, which will guarantee the acceptance of the accepting trace.

The construction described so far allows to translate an ESNW with only one pair (B, G) of edge-Streett condition into an NBW. The size of the generating automaton is $(|Q| + |G|) + |Q|$, where $|Q| + |G|$ corresponds to $\mathcal{A}_1$ and the last $|Q|$ term to $\mathcal{A}_2$.

In order to create a single NBW that captures the general case of k edge-Streett conditions, it seems plausible to construct an alternating automaton, by merging all NBW computed for each of the individual accepting conditions by simply letting $I = I_1 \wedge I_2 \wedge \ldots I_k$. The resulting automaton is an alternating Büchi automaton that can be easily converted into an NBW. Unfortunately, this construction is not correct as illustrated with the following example:

Example A.1
$\mathcal{E}$ is as follows.

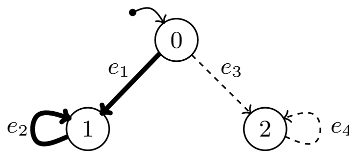


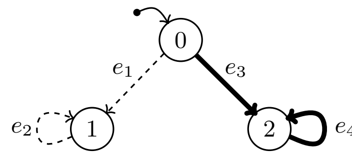
$B_1 : \{e_3, e_4\}$ $B_2 : \{e_1, e_2\}$

$G_1 : \{e_1, e_2\}$ $G_2 : \{e_3, e_4\}$

We set $L(n) =$ true for all nodes n.

For clarity we depict the two accepting edge conditions separately:





$B_1 : \{e_3, e_4\}$

$G_1 : \{e_1, e_2\}$

$B_2 : \{e_1, e_2\}$

$G_2 : \{e_3, e_4\}$

It is easy to see that $\mathcal{E}$ admits no computation. The reason is that there are only two traces:

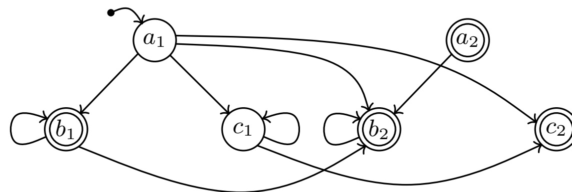
$$0 \to 1 \to 1 \to 1 \to 1 \to 1 \to 1 \ldots$$

and

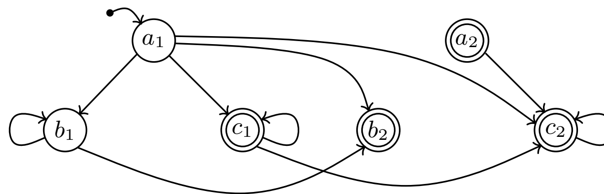
$$0 \to 2 \to 2 \to 2 \to 2 \to 2 \to 2 \to \ldots$$

The first case is rejecting for (B_2, G_2) because no good edge is traversed infinitely often, but a bad edge (namely $e_2 : (1 \to 1)$) is traversed infinitely often. Symmetrically, the second case is rejecting for (B_1, G_1).

The two NBWs obtained by the construction above are:



for pair (B_1, G_1), and the following for (B_2, G_2):



The alternating automaton obtained by conjoining these two NBWs has the following accepting run DAG:

$$(a_1, a_1) \to (b_1, c_1) \to (b_1, c_1) \to (b_1, c_1) \to (b_1, c_1) \to \ldots$$

The problem is that in the alternating automaton we allow each NBW to traverse different edges independently, only checking that the labels of the states are compatible. In our case, all states are compatible because all states are labeled *true*.

We fix this problem by building a unique NBW in which every component is forced to be in the same state. ⌟

The NBW $\mathcal{A}$ resulting from translating the full ESNW is built from the NBWs $\mathcal{A}_i$ obtained by translating each of the edge-Streett conditions. The resulting automaton is $\langle AP, Q, I, L, T, F\rangle$ where:

- Q^f contains $(Q \times 3^k \times 2^k)$. A state (q, v, o) represents that all automata are:
  - in state q,
  - that automaton $\mathcal{A}_i$ is in state q_1 (if $v[i] = 0$), in state q_1^G (if $v[i] = 1$) or instate q_2 (if $v[i] = 2$);
  - that automaton $\mathcal{A}$ owes a visit to a final state (if $o[i] = 1$), or already visited a final state since the last reset (see F below).
- label: $L((q, v, o)) = L(q)$.
- initial states: $(q, v, o)) \in I$ whenever $q \in I$, and $v[i] = 0$ for all i. For o, $o[i] = 0$ if and only if $q_1^i \in F^i$. Otherwise $o[i] = 1$.
- transitions: $(q, v, o) \to (p, v', o') \in T$ whenever, for all i, one of the following hold:
  - $(q_1 \to p_1) \in T^i$, $v[i] = 0$ and $v'[i] = 0$.
  - $(q_1 \to p_1^G) \in T^i$, $v[i] = 0$ and $v'[i] = 1$.
  - $(q_1^G \to p_1) \in T^i$, $v[i] = 0$ and $v'[i] = 1$.
  - $(q_1^G \to p_1^G) \in T^i$, $v[i] = 1$ and $v'[i] = 1$.
  - $(q_1 \to p_2) \in T^i$, $v[i] = 0$ and $v'[i] = 2$.
  - $(q_1^G \to p_2) \in T^i$, $v[i] = 1$ and $v'[i] = 2$.
  - $(q_2 \to p_2) \in T^i$, $v[i] = 2$ and $v'[i] = 2$.

  For the owing set o, if $(q, v, o) \in F$, then, for all i:

  - $o'[i] = 1$ whenever $v'[i] = 1$ or $v'[i] = 2$.
  - $o'[i] = 0$ whenever $v'[i] = 0$.

  Finally, if $(q, v, o) \notin F$, then, for all i:

  - $o'[i] = 1$ whenever $o[i] = 1$ and $v'[i] = 0$.
  - $o'[i] = 0$ whenever either $o[i] = 0$ or $v'[i] = 1$ or $v'[1] = 2$.
- accepting:
$$F = \{(q, v, o) \mid \text{for all } i,\ o[i] = 0\}$$

Informally, v serves to distinguish in which version of q each automaton is. The owing vector o, inspired by the Miyano-Hayashi construction is introduced to remember which sub-automaton has visited a final state since the last visit to a global final state, that only occurs when all sub-automata have visited a final state.

B

LEAP

This appendix provides more details about the syntax for invoking and describing programs and properties in LEAP. The appendix is structured as follows. Section B.1 lists the available command line options for invoking LEAP. Section B.2 presents the grammar which describes the syntax accepted by LEAP. This section is composed by 4 subsections. Subsection B.2.1 describes the syntax accepted by LEAP's programs. Subsection B.2.2 contains the syntax that LEAP's expressions need to follow. Subsection B.2.3 describes the syntax for LEAP's proof graphs. Finally, Subsection B.2.4 provides the description for LEAP's parametrized verification diagrams.

B.1 LEAP Command Line Options

Here we describe how to invoke LEAP and the command line options that it supports. LEAP is invoked through the following command line:

```
leap [options] [prg file]
```

where `[prg file]` is a file describing a program, following the syntax presented in Section B.2.1 and `[options]` are any of the following available command line options:

`-g [file]`: Provides the input file containing the proof graph for a safety verification. The file given as input in `[file]` must follow the syntax described in Section B.2.3. This argument is required when verifying a safety property.

`-d [folder]`: Provides the input folder where specifications and invariant candidates are stored. LEAP will search in this folder for any file with the `.inv` extension and will consider each of these files as a LEAP specification. Specifications need to follow the syntax described in Section B.2.2. This argument is required when verifying a safety property and optional when verifying a liveness property.

`-pvd [file]`: Provides the input file containing a parametrized verification diagram for the verification of a liveness property. The file given as input in `[file]` must follow the syntax described in Section B.2.4. This argument is required when verifying a safety property.

-ps [file]: Describes the input file containing auxiliary support information for a parametrized verification diagram. This support information consists of a collection of tactics and support invariants that are required for specific transitions and proof rules. The file provided as argument must follow the syntax described in Section B.2.4. This argument is optional when verifying a liveness property.

-focus [n1,n2,..]: This option tells LEAP to verify and generate verification conditions solely for the transitions passed as argument. Transitions are described as a list of integers separated by comma and each integer corresponds to the program line associated with the transition relation. Dashes can be used to specify program line ranges. For instance, 1,2,5-7,9 denotes program lines 1, 2, 5, 6, 7 and 9.

-ignore [n1,n2,..]: This option tells LEAP not to verify nor generate verification conditions for the transitions passed as argument. As the -focus options, this option receives a list of transitions represented as a list of comma separated integers or ranges. Option -ignore overrides option -focus. That means that is a transition is passed as argument to -focus and to -ignore, it will finally be ignored by LEAP.

-pvdconds [c1,c2,..]: This option tells LEAP which conditions to bear in mind when analyzing a parametrized verification diagram, in case only some conditions want to be checked. Currently supported options are: initiation, consecution, acceptance and fairness.

-pvdnodes [n1,n2,...]: This option tells LEAP which nodes of a parametrized verification diagram consider when checking a liveness property. The list of nodes must be provided as a list of comma separated node identifiers.

-dp [dp]: This option tells LEAP which specialized decision procedure use when trying to check the validity of a verification condition. Argument [dp] denotes the decision procedure to be used. Currently supported decision procedures are:

- num, a simple decision procedure for Presburger Arithmetic with finite sets.
- loc, a simple decision procedure that reasons on program locations.
- tll, a decision procedure for TL3, the theory presented in Chapter 6.
- tsl, a decision procedure for TSL, the theory of skiplists of unbounded height presented in Chapter 8.
- tslk[_], a decision procedure for TSL_K, the theory of skiplists presented in Chapter 7. Note that in this case, the maximum number of levels on which TSL_K will reason needs to be passed as argument. That is, for using TSL_2, the appropriate option would be: -dp tslk[2].

This option is mandatory for doing any verification.

-co [op]: This option specifies the general cutoff strategy to use in order to compute the domain bounds when using a model based decision procedure. Currently supported options for [op] are: dnf, union and pruning. By deafult, this option is set to dnf.

-z3: This option eanble the use of Z3 as SMT solver.

-yices+z3: This option enbales the use of Yices for the reasoning about program locations and Z3 to for the reasoning about non program locations.

-smt: This option enables the use of standard SMT-LIB translation for the queries passed to the SMT solvers. This option enables the use of virtually any SMT solver that supports SMT-LIB input format. The SMT-LIB translation is experimental and only some decision procedures currently support it.

-q: This option enables the use of quantifiers over finite domains when constructing the queries that are then passed to SMT solvers.

-si: This option tells LEAP to stop execution when an invalid verification condition is found. By default, this option is disabled and LEAP will continue executing even if it finds any invalid verification condition.

-sm: This option tells LEAP to present a counter model of any invalid verification condition. By default, this option is disabled.

-show: This option tell LEAP to print in standard output the representation of the parsed program. It is useful for finding out which transition relations are related to each program location.

-sl: When this option is enabled, LEAP prints through standard output a list of all labels declared in the parsed program in addition the program lines each label is associated with.

-v [n]: This option indicates the verbosity level used by LEAP. Argument [n] corresponds to an integer value and the greater is [n], the more information outputs LEAP.

-sf: If this option is enabled, then LEAP shows the path to the temporary files where the SMT queries are written. By default, this option is disabled.

-ovc+: This option enables the output of the generated verification conditions to a specific folder. The folder is specified using the -o option.

-o [folder]: This option indicates the folder where the generated verification conditions are output when -ovc option is enabled.

-l [file]: This option indicates LEAP to print log information to a specific file, indicated by [file].

-debug: This option enables the output of debug information.

-version: This option prints the current version of LEAP.

-help: Prints information of how to use LEAP and the command line options it supports.

--help: Same as -help option.

B.2 LEAP Syntax

In this section we describe the syntax accepted by LEAP as input for its programs description, expressions, proof graphs and parametrized verification diagrams.

We use standard grammar rules for describing the syntax. We use capital letters to denote non-terminals and bold letters between quotes for terminal symbols. We use a bar | to denote option and ? to represent 0 or 1 occurrence of a given symbol. Similarly, we use + and * to denote 1 or more and 0 or more occurrences respectively.

B.2.1 LEAP Programming Language

In this section we present the syntax accepted by LEAP's programs. Comments in a LEAP program begins with a double slash // and expands until the end of the line. For LEAP's programs, the initial symbol describing the syntax is LEAP_PROG:

```
LEAP_PROG ::=
  'global' (GLOBAL_DECL *) INITIAL_ASSUMPTIONS PROC_LIST

GLOBAL_DECL ::=
    KIND IDENT IDENT
  | KIND IDENT IDENT ':=' TERM
  | KIND IDENT IDENT ':=' FORMULA
  | KIND IDENT IDENT '<' TERM
  | KIND IDENT IDENT '>' TERM
  | KIND IDENT IDENT '<=' TERM
  | KIND IDENT IDENT '>=' TERM
  | KIND IDENT IDENT 'in' TERM
  | KIND IDENT IDENT 'subSETeq' TERM
  | KIND IDENT IDENT 'tin' TERM
  | KIND IDENT IDENT 'tsubSETeq' TERM
  | KIND IDENT IDENT 'iin' TERM
  | KIND IDENT IDENT 'isubSETeq' TERM
  | KIND IDENT IDENT 'ein' TERM
  | KIND IDENT IDENT 'esubSETeq' TERM
  | KIND IDENT 'spin' '(' IDENT ',' TERM ')'
  | KIND IDENT 'spsubSETeq' '(' IDENT ',' TERM ')'

INITIAL_ASSUMPTIONS ::=
  ('assume' FORMULA)?

PROC_LIST ::=
  (PROCEDURE)+

PROCEDURE ::=
```

```
  'procedure' IDENT ARGS PROCEDURE_SORT
    (LOCAL_DECL*)
    'begin'
      PROGRAM
    'end'

PROCEDURE_SORT ::=
  (':' IDENT)?

ARGS ::=
  '(' (ARG_LIST)? ')'

ARG_LIST ::=
    ARG
  | ARG ',' ARG_LIST

ARG ::=
  IDENT ':' IDENT

LOCAL_DECL ::=
    KIND IDENT IDENT
  | KIND IDENT IDENT ':=' TERM
  | KIND IDENT IDENT ':=' FORMULA

KIND ::=
  ('ghost')?

PROGRAM ::=
  (STATEMENT_LIST)?

STATEMENT_LIST ::=
    (LINE_LABEL*) STATEMENT (LINE_LABEL*)
  | (LINE_LABEL*) STATEMENT STATEMENT_LIST

LINE_LABEL ::=
    ':' IDENT
  | ':' IDENT '['
  | ':' IDENT ']'

STATEMENT ::=
    STM_WITH_GHOST_AND_SEMICOLON (GHOST_BLOCK | ';')
  | STM_WITH_GHOST (GHOST_BLOCK?)
  | STM

STM_WITH_GHOST_AND_SEMICOLON ::=
```

```
    'skip'
  | 'assert'
  | 'await'
  | 'noncritical'
  | 'critical'
  | TERM ':=' TERM
  | IDENT ':=' TERM
  | IDENT ':=' FORMULA
  | TERM ':=' 'call' IDENT '(' PARAMS ')'
  | IDENT ':=' 'CALL' IDENT '(' PARAMS ')'
  | TERM '->' 'lock' LOCK_POS
  | TERM '->' 'unlock' LOCK_POS
  | 'call' IDENT '(' PARAMS ')'
  | 'return' '(' (TERM?) ')'

STM_WITH_GHOST ::=
    'if' FORMULA 'then' STATEMENT_LIST
                ('else' STATEMENT_LIST*)?  'endif'
  | '{' (ATOMIC_STM+) '}'

STM ::=
    'while' FORMULA 'do' STATEMENT_LIST 'endwhile'
  | 'choice' STATEMENT_CHOICE 'endchoice'

STATEMENT_CHOICE ::=
  (STATEMENT)+ ('_or_' (STATEMENT+) )?

ATOMIC_STM ::=
    'skip' ';'
  | 'assert' FORMULA ';'
  | 'noncritical' ';'
  | 'critical' ';'
  | TERM ':=' TERM ';'
  | IDENT ':=' TERM ';'
  | IDENT ':=' FORMULA ';'
  | 'if' FORMULA 'then' (ATOMIC_STM)+
                ('else' (ATOMIC_STM)+)?  'endif'
  | 'while' FORMULA 'do' (ATOMIC_STM)+ 'endwhile'
  | 'choice' ATOMIC_CHOICE 'endchoice'

ATOMIC_CHOICE ::=
  (ATOMIC_STM)+ ('_or_' ATOMIC_CHOICE)?

GHOST_BLOCK ::=
    '$' (GHOST_STM*) '$'
```

```
GHOST_STATEMENT ::=
    'skip' ';'
  | 'assert' FORMULA ';'
  | 'await' FORMULA ';'
  | 'noncritical' ';'
  | 'critical' ';'
  | TERM ':=' TERM ';'
  | IDENT ':=' TERM ';'
  | IDENT ':=' FORMULA ';'
  | 'if' FORMULA 'then' (GHOST_STM+)
                  ('else' GHOST_STM+)?  'endif'
  | 'while' FORMULA 'do' (GHOST_STM+) 'endwhile'
  | 'choice' GHOST_CHOICE 'end_choice'

GHOST_CHOICE ::=
  (GHOST_STM)+ ('_or_' GHOST_CHOICE)?

LOCK_POS ::=
  ('[' TERM ']')?

PARAMS ::=
  (PARAM_LIST)?

PARAM_LIST ::=
    TERM
  | TERM ',' PARAM_LIST

FORMULA ::=
    '(' FORMULA ')'
  | ATOM
  | 'true'
  | 'false'
  | '~' FORMULA
  | FORMULA '/\' FORMULA
  | FORMULA '\/' FORMULA
  | FORMULA '->' FORMULA
  | FORMULA '=' FORMULA
  | '(' IDENT ')'

ATOM ::=
    'append' '(' TERM ',' TERM ',' TERM ')'
  | 'reach' '(' TERM ',' TERM ',' TERM ',' TERM ')'
  | 'orderlist' '(' TERM ',' TERM ',' TERM ')'
  | 'skiplist' '(' TERM ',' TERM ',' TERM ','
```

```
                   TERM ',' TERM ',' TERM ')'
  | 'in' '(' TERM ',' TERM ')'
  | 'subSETeq' '(' TERM ',' TERM ')'
  | 'tin' '(' TERM ',' TERM ')'
  | 'tsubSETeq' '(' TERM ',' TERM ')'
  | 'iin' '(' TERM ',' TERM ')'
  | 'isubSETeq' '(' TERM ',' TERM ')'
  | 'ein' '(' TERM ',' TERM ')'
  | 'esubSETeq' '(' TERM ',' TERM ')'
  | 'spin' '(' TERM ',' TERM ')'
  | 'spsubSETeq' '(' TERM ',' TERM ')'
  | TERM '<' TERM
  | TERM '>' TERM
  | TERM '<=' TERM
  | TERM '>=' TERM
  | TERM '=' TERM
  | TERM '!=' TERM

TERM ::=
    IDENT
  | SET
  | ELEM
  | THID
  | ADDR
  | cell
  | SETTH
  | SETINT
  | SETELEM
  | SETPAIR
  | PATH
  | MEM
  | INTEGER
  | PAIR
  | ARRAYLOOKUP
  | '(' TERM ')'

SET ::=
    'empty'
  | '{' TERM '}'
  | 'union' '(' TERM ',' TERM ')'
  | 'intr' '(' TERM ',' TERM ')'
  | 'diff' '(' TERM ',' TERM ')'
  | 'path2set' '(' TERM ')'
  | 'addr2set' '(' TERM ',' TERM ')'
  | 'addr2set' '(' TERM ',' TERM ',' TERM ')'
```

```
ELEM ::=
    TERM '.' 'data'
  | TERM '->' 'data'
  | 'havocListElem' '(' ')'
  | 'havocSLElem' '(' ')'
  | 'lowestElem'
  | 'highestElem'

THID ::=
    TERM '.' 'lockid'
  | '#'
  | TERM '->' 'lockid'
  | 'me'
  | 'tid_of' '(' TERM ')'

ADDR ::=
    'null'
  | TERM '.' 'next'
  | TERM '.' 'nextat' '[' TERM ']'
  | TERM '.' 'arr' '[' TERM ']'
  | 'firstlocked' '(' TERM ',' TERM ')'
  | 'lastlocked' '(' TERM ',' TERM ')'
  | 'malloc' '(' TERM ',' TERM ',' TERM ')'
  | 'mallocSL' '(' TERM ',' TERM ')'
  | 'mallocSLK' '(' TERM ',' TERM ')'
  | TERM '->' 'next'
  | TERM '->' 'nextat' '[' TERM ']'
  | TERM '->' 'arr' '[' TERM ']'

CELL ::=
    'error'
  | 'mkcell' '(' TERM ',' TERM ',' TERM ')'
  | 'mkcell' '(' TERM ',' '[' TERM_list ']' ','
                          '[' TERM_list ']' ')'
  | 'mkcell' '(' TERM ',' TERM ',' TERM ',' TERM ')'
  | TERM '.' 'lock'
  | TERM '.' 'unlock'
  | 'rd' '(' TERM ',' TERM ')'

SETTH ::=
    'tempty'
  | 'tsingle' '(' TERM ')'
  | 'tunion' '(' TERM ',' TERM ')'
  | 'tintr' '(' TERM ',' TERM ')'
```

```
  | 'tdiff' '(' TERM ',' TERM ')'

SETINT ::=
    'iempty'
  | 'isingle' '(' TERM ')'
  | 'iunion' '(' TERM ',' TERM ')'
  | 'iintr' '(' TERM ',' TERM ')'
  | 'idiff' '(' TERM ',' TERM ')'

SETELEM ::=
    'eempty'
  | 'esingle' '(' TERM ')'
  | 'eunion' '(' TERM ',' TERM ')'
  | 'eintr' '(' TERM ',' TERM ')'
  | 'ediff' '(' TERM ',' TERM ')'
  | 'set2elem' '(' TERM ',' TERM ')'

SETPAIR ::=
    'spempty'
  | 'spsingle' '(' TERM ')'
  | 'spunion' '(' TERM ',' TERM ')'
  | 'spintr' '(' TERM ',' TERM ')'
  | 'spdiff' '(' TERM ',' TERM ')'

PATH ::=
    'epsilon'
  | 'singlePath' '(' TERM ')'
  | 'getp' '(' TERM ',' TERM ',' TERM ')'

MEM ::=
    'upd' '(' TERM ',' TERM ',' TERM ')'

INTEGER ::=
    NUMBER
  | '-' TERM
  | TERM '+' TERM
  | TERM '-' TERM
  | TERM '*' TERM
  | TERM '/' TERM
  | 'setIntMin' '(' TERM ')'
  | 'setIntMax' '(' TERM ')'
  | 'havocLevel' '(' ')'
  | 'int_of' '(' TERM ')'

PAIR ::=
```

```
    '(' TERM ',' TERM ')'
  | 'spmin' '(' TERM ')'
  | 'spmax' '(' TERM ')'

ARRAYLOOKUP ::=
    TERM '[' TERM ']'

IDENT ::=
    ['A'-'Z','a'-'z']
    (['A'-'Z','a'-'z','0'-'9','_','/',''','@'])*

NUMBER ::=
  (['0'-'9'])+
```

B.2.2 LEAP Expressions

In this section we describe the LEAP syntax for specification files and LEAP's expression.

For specification files, the initial symbol is INVARIANT:

```
INVARIANT ::=
    'vars' ':' (VAR_DECL*)
    'invariant' ('[' IDENT ']')? ':' FORMULA_DECL+

FORMULA_DECL ::=
    FORMULA
  | '#' IDENT ':' FORMULA

VAR_DECL ::=
  IDENT IDENT
```

And for LEAP's expression, the initial symbol is FORMULA:

```
FORMULA ::=
    '(' formula ')'
  | LITERAL
  | 'true'
  | 'false'
  | '~' FORMULA
  | FORMULA '/\' FORMULA
  | FORMULA '\/' FORMULA
  | FORMULA '->' FORMULA
  | FORMULA '=' FORMULA
```

```
  | '@' NUMBER (TH_PARAM?) '.'
  | '@' IDENT (TH_PARAM?) '.'


TH_PARAM ::=
    '(' IDENT ')'
  | '(' NUMBER ')'


LITERAL ::=
    'append' '(' TERM ',' TERM ',' TERM ')'
  | 'reach' '(' TERM ',' TERM ',' TERM ',' TERM ')'
  | 'reach' '(' TERM ',' TERM ',' TERM ',' TERM ',' TERM ')'
  | 'orderlist' '(' TERM ',' TERM ',' TERM ')'
  | 'skiplist' '(' TERM ',' TERM ',' TERM ','
                   TERM ',' TERM ',' TERM ')'
  | 'in' '(' TERM ',' TERM ')'
  | 'subSETeq' '(' TERM ',' TERM ')'
  | 'tin' '(' TERM ',' TERM ')'
  | 'tsubSETeq' '(' TERM ',' TERM ')'
  | 'iin' '(' TERM ',' TERM ')'
  | 'isubSETeq' '(' TERM ',' TERM ')'
  | 'ein' '(' TERM ',' TERM ')'
  | 'esubSETeq' '(' TERM ',' TERM ')'
  | 'spin' '(' TERM ',' TERM ')'
  | 'spsubSETeq' '(' TERM ',' TERM ')'
  | 'inintPAIR' '(' TERM ',' TERM ')'
  | 'intidPAIR' '(' TERM ',' TERM ')'
  | 'uniqueint' '(' TERM ')'
  | 'uniquetid' '(' TERM ')'
  | TERM '<' TERM
  | TERM '>' TERM
  | TERM '<=' TERM
  | TERM '>=' TERM
  | TERM '=' TERM
  | TERM '!=' TERM
  | '.' ID '.'
  | '.' IDENT '::' IDENT TH_PARAM '.'

TERM ::=
    IDENT
  | SET
  | ELEM
  | THID
  | ADDR
```

```
  | CELL
  | SETTH
  | SETINT
  | SETELEM
  | SETPAIR
  | PATH
  | MEM
  | INTEGER
  | PAIR
  | ARRAYS
  | ADDRARR
  | TIDARR
  | '(' TERM ')'

ID ::=
    IDENT
  | IDENT '::' IDENT
  | IDENT '::' IDENT TH_PARAM

SET ::=
    'empty'
  | '{' TERM '}'
  | 'union' '(' TERM ',' TERM ')'
  | 'intr' '(' TERM ',' TERM ')'
  | 'diff' '(' TERM ',' TERM ')'
  | 'path2set' '(' TERM ')'
  | 'addr2set' '(' TERM ',' TERM ')'
  | 'addr2set' '(' TERM ',' TERM ',' TERM ')'

ELEM ::=
    TERM '.' 'data'
  | 'lowestElem'
  | 'highestElem'

THID ::=
    TERM '.' 'lockid'
  | '#'
  | 'tid_of' '(' TERM ')'

ADDR ::=
    'null'
  | TERM '.' 'next'
  | TERM '.' 'nextat' '[' TERM ']'
  | 'firstlocked' '(' TERM ',' TERM ')'
  | 'firstlocked' '(' TERM ',' TERM ',' TERM ')'
```

```
  | 'lastlocked' '(' TERM ',' TERM ')'

CELL ::=
    'error'
  | 'mkcell' '(' TERM ',' TERM ',' TERM ')'
  | 'mkcell' '(' TERM ',' TERM ',' TERM ',' TERM ')'
  | 'mkcell' '(' TERM ',' '[' TERM_LIST ']' ','
                          '[' TERM_LIST ']' ')'
  | TERM '.' 'lock' '(' TERM ')'
  | TERM '.' 'lockat' '(' TERM ',' TERM ')'
  | TERM '.' 'unlock'
  | TERM '.' 'unlockat' '(' TERM ')'
  | 'rd' '(' TERM ',' TERM ')'

TERM_LIST ::=
    TERM ',' TERM
  | TERM ',' TERM_LIST

SETTH ::=
    'tempty'
  | 'tsingle' '(' TERM ')'
  | 'tunion' '(' TERM ',' TERM ')'
  | 'tintr' '(' TERM ',' TERM ')'
  | 'tdiff' '(' TERM ',' TERM ')'
  | 'lockset' '(' TERM ',' TERM ')'

SETINT ::=
    'iempty'
  | 'isingle' '(' TERM ')'
  | 'iunion' '(' TERM ',' TERM ')'
  | 'iintr' '(' TERM ',' TERM ')'
  | 'idiff' '(' TERM ',' TERM ')'
  | 'setLower' '(' TERM ',' TERM ')'


SETPAIR ::=
    'spempty'
  | 'spsingle' '(' TERM ')'
  | 'spunion' '(' TERM ',' TERM ')'
  | 'spintr' '(' TERM ',' TERM ')'
  | 'spdiff' '(' TERM ',' TERM ')'
  | 'splower' '(' TERM ',' TERM ')'


SETELEM :
```

```
    'eempty'
  | 'esingle' '(' TERM ')'
  | 'eunion' '(' TERM ',' TERM ')'
  | 'eintr' '(' TERM ',' TERM ')'
  | 'ediff' '(' TERM ',' TERM ')'
  | 'set2elem' '(' TERM ',' TERM ')'

PATH ::=
    'epsilon'
  | '[' TERM ']'
  | 'getp' '(' TERM ',' TERM ',' TERM ')'
  | 'getp' '(' TERM ',' TERM ',' TERM ',' TERM ')'

MEM ::=
  | 'upd' '(' TERM ',' TERM ',' TERM ')'

INTEGER ::=
    NUMBER
  | '-' TERM
  | TERM '+' TERM
  | TERM '-' TERM
  | TERM '*' TERM
  | TERM '/' TERM
  | 'setIntMin' '(' TERM ')'
  | 'setIntMax' '(' TERM ')'
  | TERM '.' 'max'
  | 'int_of' '(' TERM ')'

PAIR ::=
    '(' TERM ',' TERM ')'
  | 'spmin' '(' TERM ')'
  | 'spmax' '(' TERM ')'

ARRAYS ::=
    TERM '[' TERM ']'
  | 'arrUpd' '(' TERM ',' TERM ',' TERM ')'

ADDRARR ::=
    TERM '.' 'arr'

TIDARR ::=
    TERM '.' 'tids'

IDENT ::=
  ['A'-'Z','a'-'z','$']
```

```
  (['A'-'Z','a'-'z','0'-'9','_','/',''','@'])*

NUMBER ::=
  (['0'-'9'])+
```

B.2.3 LEAP Proof Graphs and PVD Support

We now present the syntax for proof graph and parametrized verification diagram support files. For proof graphs the initial symbol is GRAPH, while for support files for parametrized verification diagrams the initial symbol is PVD_SUPPORT.

```
PVD_SUPPORT ::=
    'Tactics' ':' (TACTIC_CASE*) 'Facts' ':' (FACT*)

TACTIC_CASE ::=
    (TACTICS?) ';'
  | NUMBER ':' (TACTICS?) ';'
  | NUMBER ':' CONDITION ':' (TACTICS?) ';'
  | NUMBER ':' '[' IDENT_LIST BAR CONDITION_LIST ']'
           ':' (TACTICS?) ';'

FACT ::=
    INV_LIST ';'
  | NUMBER ':' INV_LIST ';'
  | NUMBER ':' CONDITION ':' INV_LIST ';'

GRAPH ::=
  RULE*

RULE ::=
  | (INV_LIST?) '=>' INV CASES (TACTICS?)
  | (INV_LIST?) '->' INV SEQ_CASES (TACTICS?)

INV_LIST ::=
    INV_GROUP
  | INV
  | INV_GROUP ',' INV_LIST
  | INV ',' INV_LIST

INV_GROUP ::=
    IDENT '::' '{' IDENT_LIST '}'
```

```
IDENT_LIST ::=
    IDENT
  | IDENT ',' IDENT_LIST

INV ::=
    IDENT
  | IDENT '::' IDENT

CASES ::=
    ('[' CASE_LIST ']')?

SEQ_CASES ::=
    ('[' SEQ_CASE_LIST ']')?

CASE_LIST ::=
    CASE
  | CASE ';' CASE_LIST

SEQ_CASE_LIST ::=
    SEQ_CASE
  | SEQ_CASE ';' SEQ_CASE_LIST

CASE ::=
    NUMBER ':' (PREMISE?) (INV_LIST?) (TACTICS?)

SEQ_CASE ::=
    NUMBER ':' (INV_LIST?) (TACTICS?)

PREMISE ::=
    'S' ':'
  | 'O' ':'

CONDITION ::=
    'I'
    'C'
    'A'
    'F'

CONDITION_LIST ::=
    CONDITION
  | CONDITION ',' CONDITION_LIST

TACTICS ::=
    '{' (SMP_STRATEGY?)
               SUPP_SPLIT_TACTIC_LIST
```

```
               SUPP_TACTIC_LIST
               FORMULA_SPLIT_TACTIC_LIST
               (FORMULA_TACTIC_LIST?) '}'

SMP_STRATEGY ::=
    'union' ':'
  | 'pruning' ':'
  | 'dnf' ':'

SUPP_SPLIT_TACTIC_LIST ::=
  | '|'
  | IDENT SUPP_SPLIT_TACTIC_LIST

SUPP_TACTIC_LIST ::=
  | '|'
  | IDENT SUPP_TACTIC_LIST

FORMULA_SPLIT_TACTIC_LIST ::=
  | '|'
  | IDENT FORMULA_SPLIT_TACTIC_LIST

FORMULA_TACTIC_LIST ::=
    IDENT FORMULA_TACTIC_LIST


IDENT ::=
  ['A'-'Z','a'-'z']
  (['A'-'Z','a'-'z','0'-,'9','/',''','@','_','-'])*

NUMBER ::=
  (['0'-'9'])+
```

B.2.4 LEAP Parametrized Verification Diagrams

We now present the syntax for parametrized verification diagrams. The FORMULA non-terminal used below corresponds to the one described in Section B.2.2. The initial symbol describing the syntax for LEAP's parametrized verification diagrams is PVD:

```
PVD ::=
    'Diagram' '[' IDENT ']'
    'Nodes' ':' NODE_LIST
    'Boxes' ':' (BOX*)
    'Initial' ':' NODE_ID_LIST
```

```
    'Edges' ':' (EDGE+)
    'Acceptance' ':' (ACCEPTANCE+)

NODE_LIST ::=
    NODE
  | NODE ',' NODE_LIST

NODE ::=
    IDENT
  | IDENT '{' FORMULA '}'

node_id_list ::=
    IDENT
  | IDENT ',' NODE_ID_LIST

BOX ::=
    '{' IDENT '[' IDENT ']' ':' NODE_ID_LIST '}'

TRANS_LIST ::=
    TRANS
  | TRANS ',' TRANS_LIST

TRANS ::=
    NUMBER '(' IDENT ')'

EDGE ::=
    '[' IDENT '-->' ']' ';'
  | IDENT '-->' IDENT ';'
  | '[' IDENT '-{' TRANS_LIST '}->' IDENT ']' ';'
  | IDENT '-{' TRANS_LIST '}->' IDENT ';'

ACCEPTANCE ::=
    '<<' 'Bad' ':' '{' (ACCEPT_EDGE_LIST?) '}' ';'
        'Good' ':' '{' (ACCEPT_EDGE_LIST?) '}' ';'
        '[' DELTA_LIST ']' '>>'

ACCEPT_EDGE_LIST ::=
    ACCEPT_EDGE
  | ACCEPT_EDGE ',' ACCEPT_EDGE_LIST

ACCEPT_EDGE ::=
    '(' IDENT ',' IDENT ',' IDENT ')'

DELTA_LIST ::+
    DELTA
```

```
  | DELTA ';' DELTA_LIST

DELTA ::=
    '(' TERM ',' WF_OP ')'

WF_OP ::=
    'subset_op'
  | 'pairsubset_op'
  | 'addrsubset_op'
  | 'elemsubset_op'
  | 'tidsubset_op'
  | 'less_op'
```