Noname manuscript No.
(will be inserted by the editor)

Parametrized Verification Diagrams:

Temporal Verification of Symmetric Parametrized Concurrent Systems

Alejandro Sánchez · César Sánchez

the date of receipt and acceptance should be inserted later

Abstract This paper studies the problem of verifying temporal properties (including liveness properties) of parametrized concurrent systems executed by an unbounded number of threads. To solve this problem we introduce *parametrized verification diagrams* (PVDs), that extend the so-called generalized verification diagrams (GVDs) adding support for parametrized verification. Even though GVDs are known to be a sound and complete proof system for non-parametrized systems, the application of GVDs to parametrized systems requires using quantification or finding a potentially different diagram for each instantiation of the parameter (number of threads). As a consequence, the use of GVDs in parametrized verification requires discharging and proving either quantified formulas or an unbounded collection of verification conditions.

Parametrized verification diagrams enable the use of a *single* diagram to represent the proof that all possible instances of the parametrized concurrent system satisfy the given temporal specification. Checking the proof represented by a PVD requires proving only a finite collection of quantifier-free verification conditions.

The PVDs we present here assume that the parametrized systems are symmetric, which covers a large class of concurrent and distributed systems, including concurrent data types. Our second contribution is an implementation of PVDs and its integration into Leap, our prototype theorem prover. Finally, we illustrate empirically, using Leap, the practical applicability of PVDs by building and checking proofs of liveness properties of mutual exclusion protocols and concurrent data structures. To the best of our knowledge, these are the first machine-checkable proofs of liveness properties of these concurrent data types.

Keywords temporal logic; formal verification; formal methods; liveness properties; parametrized systems; concurrent data types; deductive method; verification conditions

Alejandro Sánchez
IMDEA Software Institute, Madrid, Spain
E-mail: alejandro.sanchez@imdea.org

César Sánchez
IMDEA Software Institute, Madrid, Spain
E-mail: cesar.sanchez@imdea.org

1 Introduction

We are interested in the verification of parametrized temporal properties of systems executed by an arbitrary number of threads, with a special emphasis on programs that modify the heap by concurrently manipulating complex pointer-based data structures. Examples of these programs include concurrent implementations of conventional software data types.

The problem of verifying parametrized systems has received a lot of attention in recent years, particularly the verification of finite-state parametrized systems. However, in general, the problem is undecidable [1], even for finite-state components [51]. There are two ways to overcome this limitation: (a) algorithmic approaches, which are necessarily incomplete; and (b) deductive proof methods, that require manual intervention. Most of the research performed in recent years focuses on algorithmic methods trying to improve their applicability.

Typically, algorithmic methods—in order to regain decidability—are restricted to finite-state processes [15, 16, 20] and finite-state shared data. In the work presented in this paper we propose a complementary approach, seeking a *general* method to prove arbitrary temporal properties of a large class of infinite-state parametrized systems. We extend temporal deductive methods such as Manna and Pnueli's [33] with specialized proof rules and other deductive formalisms tailored for parametrized systems. In comparison with algorithmic techniques we sacrifice full automation to handle complex concurrency and data manipulation. We propose to start from a widely applicable method and improve its automation, instead of trying to improve the applicability of an automatic method.

In contrast with approaches that handle the data and the evolution of the computation altogether, the temporal deductive style of reasoning allows a clean separation in a proof between the temporal part—which explains why the interleaving of actions that a set of threads can perform satisfies the given property—and the underlying data being manipulated.

We aim for a general framework that enables the formal verification of safety and liveness properties of parametrized concurrent systems. Our framework is based on temporal deductive techniques. In [48] we introduced a collection of proof rules for proving safety properties of parametrized systems, that are adaptations of the classical invariance rules for non-parametrized systems. In this paper we present deductive techniques to prove temporal properties (not only safety but also liveness properties) of parametrized systems. Our methods take a parametrized program and a property, with additional annotations if required, and generate a finite number of verification conditions (VCs). The validity of these VCs implies that the parametrized program satisfies the property. In previous papers, we have explored the construction and implementation of decidable theories for a number of data heap layouts [43–45] common in concurrent data types. The decision procedures for these theories can be used to automatically check the validity of the verification conditions as long as these VCs are quantifier-free. We show in [43–45], using model-theoretic arguments, that many theories of data are decidable and can be used as state and data assertion languages in temporal deductive frameworks like the one introduced in this paper.

The method we propose here is called parametrized verification diagrams. A preliminary version of PVDs was introduced in [47]. PVDs are an extension of generalized verification diagrams [12, 50], specialized to deal with concurrent parametrized systems. Generalized verification diagrams [12, 32, 50] are a formalism to prove temporal properties of reactive systems, even if the state space of these systems is infinite. A diagram is essentially an abstraction of the system crafted specifically for the property under consideration. Diagrams can be precise enough to formally represent the temporal proof, and allow to check its correctness mechanically. All verification conditions can be checked automatically provided there are decision procedures for the state assertion language. GVDs are effective for the verification of non-parametrized concurrent and reactive systems. Although using GVDs directly for parametrized verification is possible (see [34] and Section 8 of [8]), it requires the use of quantifiers which in turn precludes the use of many decision procedures, like the ones mentioned above. The parametrized verification diagrams introduced in this paper are a specific temporal deductive technique which is general enough to effectively enable proving temporal properties. PVDs are designed for coping with parametrized systems without generating quantified verification conditions. In this paper, we present in full detail PVDs and we prove the soundness of this method. We illustrate how to use PVDs for the verification of parametrized liveness properties and we present some empirical results obtained using an implementation of PVDs in Leap [46], a tool under development at the IMDEA Software Institute for the verification of parametrized concurrent systems.

Verification Diagrams. Formal verification using generalized verification diagrams starts from a program and a specification in linear temporal logic (LTL) [33, 39]. These formulas are built from atomic predicates capturing properties of program states. These atomic predicates can be combined using Boolean connectives and temporal operators like $\square$ (always), $\diamond$ (eventually), $\bigcirc$ (next) or $\mathcal{U}$ (until).

The semantics of the program are represented as a *fair transition system* (FTS) that encompasses all executions of the program. Given a program P and a temporal specification φ, as shown in Fig. 1, a GVD $\mathcal{D}$ encodes a proof that all fair traces of program P satisfy the temporal specification φ. Checking the proof encoded by 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. The validity of all verification conditions guarantees that the diagram covers all (fair) executions of the system.
2. Check that every fair path of the diagram satisfies the temporal property.

The first part can be handled using suitable decision procedures (DPs) for the underlying data that the program manipulates, like Boolean, integers, lists in the heap, trees, etc. The second part can be fully automated using finite-state model

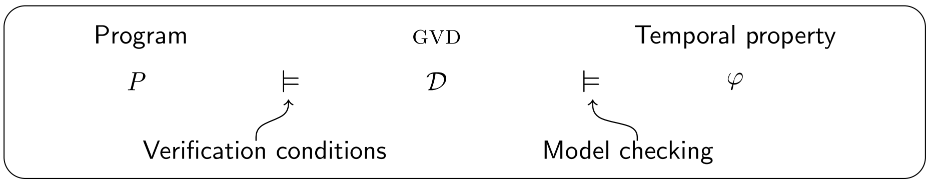

Fig. 1: Schematic representation of the use of a GVD $\mathcal{D}$ to prove $P \vDash \varphi$.

checkers. This way, GVDs cleanly separates two concerns: the temporal reasoning and the data manipulation. GVDs are complete in the following sense: if a reactive system satisfies a given temporal property then there is a diagram that encodes a correct proof. Unfortunately, GVDs cannot be used directly (without quantification) to verify concurrent programs executed by 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. Each closed system requires finding a different diagram, discharging a different collection of verification conditions, and solving a different model checking problem.

Parametrized Verification Diagrams. In order to verify parametrized temporal properties of systems composed by an unbounded number of threads we have developed *parametrized verification diagrams.* 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 handling a single finite collection of quantifier-free 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 correctness argument usually only requires to reason about a fix number of threads at each time instant. This finite collection includes: (a) the threads referred to in the property; (b) some other threads with a particularly significant characteristic in the given state (e.g., being a leader); and (c) one fresh thread identifier representing an arbitrary thread that executes a small step in the global execution.

The PVDs we present in this paper rely on the symmetry assumption. This assumption states that thread identifiers are interchangeable and are only compared using equality and inequality. 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, in particular concurrent data types [27], which are an efficient approach to exploit the parallelism of modern multiprocessor architectures. The assumption of symmetry is guaranteed when thread identifiers are only compared with equality and inequality, and when concurrency primitives do not depend on specific thread identifiers. This model of concurrency is quite pervasive. In fact, most modern programming languages like Java and the concurrency control found in modern processors and operating systems implement this model of concurrency. Concurrent data types, for example, can be modeled naturally as fully symmetric parametrized systems, where each thread executes in parallel a client of the data type. The results reported in this paper are part of a larger research effort aimed at the verification of complex properties of concurrent data types.

The rest of the paper is structured as follows. Section 2 presents a simple (infinite-state) mutual exclusion protocol based on tickets, which we will use as a running example. Section 3 includes the preliminaries and notation. Section 4 presents PVDs and shows the main result of this paper: that PVDs are sound as a proof system. Section 5 presents the experimental results we have obtained using PVDs to prove liveness properties of the mutual exclusion protocol and of an implementation of concurrent lock-coupling lists from [27]. Section 6 describes related work. Finally, Section 7 concludes.

2 Running Example: A Mutual Exclusion Protocol

For illustration purposes we use a simple programming language similar to SPL [33] to express programs. A parametrized program P is described by a sequence of SPL instructions. Each instruction in P is identified uniquely by a program location ranging from line 1 up to L. We use *Locs* for the set $\{1 \ldots L\}$ of lines in a given program. A program P also contains a number of typed variable identifiers, partitioned into *global* variables and *local* variables. We use V_{global} for the set of global variables and V_{local} for the set of local variables. A parametrized program is run in parallel by a collection of threads. The size of this collection is not known a priori. We consider in this paper asynchronous interleaving semantics as the semantics of parallel composition: at each point in time one of the threads executes an action, corresponding to the program statement pointed to by its program counter. Fig. 2 shows a symmetric protocol that solves the mutual exclusion problem using tickets, first proposed in [13] (see also [35]), and based on Lamport's bakery algorithm [31]. The version we present here is a slight modification of [13, 35], where we store the tickets in a set of pairs. Each pair consists of a ticket and a thread identifier, to capture the association between a thread and the ticket it holds.

The goal of program MutExc is to protect the critical section at line 5, using two global variables. The int variable *avail* stores the shared increasing counter. The set variable *bag* stores the ticket number and the thread identifier of all threads that are trying to access the critical section. In our example, we use the reserved word me to denote the identifier of the thread currently executing the program. When a thread wishes to enter the critical section (line 3), it grabs the current value of *avail* as its ticket and atomically performs the following two operations: (a) it increases the value of *avail*, and (b) it adds the value of its ticket as well as its thread identifier to the global storage *bag*. Then, the thread waits (line 4) until it holds the smallest ticket in the storage *bag*. Upon exiting the critical section, the thread removes its ticket from the bag (line 6).

In this paper we illustrate how to use PVDs to formally prove that if a thread wants to enter the critical section, then it will eventually enter the critical section. In Section 5 we also use PVDs to verify a liveness property of a concurrent lock-coupling list [27, 52] implementation that manipulates an unbounded heap.

```
global                                  procedure MutExc
  int $avail := 0$                        int $ticket := 0$
  set⟨int, tid⟩ $bag := \emptyset$           begin
                                     1: while $true$ do
                                     2:     noncritical
                                     3:     ⟨ $ticket := avail{+}{+}$
                                              $bag.add(ticket, \mathsf{me})$ ⟩
                                     4:     await $(bag.min == ticket)$
                                     5:     critical
                                     6:     $bag.remove(ticket, \mathsf{me})$
                                     7: end while
                                        end procedure
```

Fig. 2: MutExc: Mutual exclusion algorithm.

3 Model of Computation

In this section we introduce a general model of computation to reason about parametrized concurrent programs. We first revisit the notion of fair transition system. One of the first works to use similar transition systems in concurrent verification was [29]. We borrow the formalism from the monograph [33]. Second, we introduce parametrized programs and parametrized fair transition systems, which in turn provide the necessary 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.

3.1 Fair Transition Systems

A fair transition system, which models the executions of a non-parametrized system, is a tuple $\mathcal{S} : \langle \Sigma_{\mathsf{prog}}, V, \Theta, \mathcal{T}, \mathcal{J} \rangle$ where:

- **Signature**. The signature Σ_{prog} is a first-order signature modeling the data manipulated by the program, where a signature $\Sigma : (S, F, P)$ consists of a set of sorts S, a set F of function symbols, and a set P of predicate symbols. We use T_{prog} for the theory that allows to reason about formulas from Σ_{prog}. There are many theories that are useful as assertion languages in software verification, which have decidable satisfiability and validity problems, particularly when restricted to their quantifier-free fragments.
- **Program Variables**. V is a finite set of typed variables, whose types are taken from the set of sorts in Σ_{prog}. We use V^{σ} to denote all variables of sort σ in set V. In this paper we use type and sort interchangeably.
- **Initial Condition**. Θ is the initial condition, expressed as a first-ordered assertion from theory T_{prog} using only variables from V. Values of the variables V satisfying Θ correspond to initial states of the system.
- **Transition Relation**. $\mathcal{T}$ is a finite set of transitions. Each transition τ is expressed as a first-order formula $\tau(V, V')$ that can refer to program variables from V and to their primed version in V'. The set V' contains a fresh copy of v' of each variable v from V. The variable v' denotes the value of variable v after a transition is taken.
- **Fairness condition**. $\mathcal{J} \subseteq \mathcal{T}$ is the set of fair transitions.

A *state* is an interpretation of V, which assigns to each program variable a value of the corresponding type. A transition between two states s and s' satisfies a transition relation τ when the combined valuation which 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 $\tau(s, s')$. We will generally use the predicate *pres* over a set of variables to indicate that the transition preserves the values of the variables, that is, that the values of these variables are the same in s and in s'. Formally, if $U \subseteq V$ is a set of variables, then $\mathit{pres}(U)$ is a short for $\bigwedge_{u \in U} u' = u$. We borrow the notion of *pres* from [33] which is equivalent to the common notion of framing in program logics.

We use pc to denote the program counter, which in the case of non-parametrized systems is a variable of type loc. As usual, we use $pc \in \{i, j\}$ and $pc \in \{i_1 \dots i_n\}$ to denote $(pc = i \vee pc = j)$ and $\bigvee_{j=1}^{n} pc = i_j$, respectively. Note that pc captures the program counter of a sequential program (which fair transition systems intend to model). We will present later a generalization for concurrent systems and parametrized concurrent systems.

Example 1 Consider the following program statement:

$$1\colon\ x := x + 3$$

This is modeled by the following formula:

$$pc = 1 \quad \wedge \quad pc' = 2 \quad \wedge \quad x' = x + 3 \quad \wedge \quad pres(V \setminus \{pc, x\})$$

$\square$

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: $x := x + 3$" can always be taken if the program is at location 1.

A *run* of $\mathcal{S}$ is an infinite sequence $s_0\tau_0s_1\tau_1s_2\dots$ of states and transitions such that the following two conditions hold[1]:

(a) the first state satisfies the initial condition: $s_0 \vDash \Theta$.
(b) all steps are legal. That is, for all i, the relation $\tau_i(s_i, s_{i+1})$ holds.

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 $\pi : s_0\tau_0s_1\dots$ of a system $\mathcal{S}$, the corresponding run π^{AP} for a given propositional vocabulary AP is the sequence $P_1P_2\dots$ with $P_i \subseteq AP$, such that for all instants i:

$$s_i \vDash p \text{ for all } p \in P_i \quad \text{and} \quad s_i \vDash \neg p \text{ for all } p \notin P_i$$

We use $\mathcal{L}^{AP}(\mathcal{S})$ for the set of sequences of propositions from AP that results 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 $\mathcal{S} \vDash \varphi$.

[1] For notational convenience we have added the transitions in the sequence $s_0\tau_0s_1\tau_1s_2\dots$ instead of defining a run as a sequence of states $s_0s_1s_2\dots$, which is a more standard approach. The main reason is to be able to easily refer to the thread and transition that executed a given step. Both notions of run are equivalent, since one can equip states with a fresh variable to record the last transition that was executed from the previous state.

3.2 Parametrized Concurrent Programs

Given a parametrized program P, we associate P with an *instance family* $\{P[M]\}$, a collection of non-parametrized transition systems $P[M]$ 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 thread identifiers in the instance $P[M]$. Formally, for each M, the concrete non-parametrized transition system $P[M] : \langle \Sigma_{\mathsf{prog}}, V, \Theta, \mathcal{T}, \mathcal{J} \rangle$ consists of:

- **Signature**. The signature Σ_{prog} is as in non-parametrized fair transition systems.
- **Program Variables**. The set V of typed variables is:

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

  Note that "$v[k]$" and "$p[k]$" are indivisible variable names. We could have chosen alternative names like v_k or vk. The set $\{pc[k] \mid k \in [M]\}$ contains one variable of sort loc for each thread identifier k in $[M]$. The variable $pc[k]$ models 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 the following two predicates. First, Θ_g, that only refers to variables from V_{global}. Second, Θ_l, that can refer to variables in V_{global} and V_{local}. Both predicates are extracted from P using the semantics of the programming language. Given a thread identifier $a \in [M]$ for a concrete system $P[M]$, $\Theta_l[a]$ 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 $P[M]$ is:

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

- **Transition Relation**. $\mathcal{T}$ contains a transition $\tau_\ell[a]$ for each program location ℓ and thread identifier a in $[M]$, which are obtained again from P by 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**. We consider all transitions fair, that is $\mathcal{J} = \mathcal{T}$.

Example 2 Consider the program MutExc in Fig. 2, which corresponds to the instance family $\{\textsc{MutExc}[M]\}$. The instance consisting of two running threads, MutExc[2], contains the following variables:

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

The global variable *avail* has type int, and global variable *bag* has type $\mathsf{set}\langle\mathsf{int}, \mathsf{tid}\rangle$. 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 $\mathsf{loc} = \{1 \dots 7\}$. The initial condition of MutExc[2] is:

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

There are fourteen transitions in MutExc[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:

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

The transitions for thread 1 are analogous. The predicate *pres*, introduced in page 6, summarizes the preservation of the values of variables, that is, *pres* preserves all variables in the set it receives as argument. In MutExc[2], the predicate $pres(V \setminus \{bag, pc[0]\})$ is:

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

Note that each transition in MutExc[2] is expressed as a quantifier-free formula. These formulas are expressed using a combination of theories, including Presburger Arithmetic and a theory of finite sets of pairs with non-repeating first component and a minimum function according also to the first component. □

A plausible alternative model of computation to model parametrized executions would include only one transition per program location, independently of the number of threads. A transition in this model of computation would choose one thread that satisfies its enabling condition 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 non-parametrized FTS guarantees that a fair transition must be taken if enabled continuously. In the alternative model of computation, lifting 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 extending the temporal reasoning specifically for this purpose or to use quantification over threads to express the precise fairness condition.

3.3 Parametrized FTS and Parametrized Formulas

A parametrized transition system associated with a parametrized program P is a tuple $\mathcal{P}_P : \langle \Sigma_{\mathsf{param}}, V_{\mathsf{param}}, \Theta_{\mathsf{param}}, \mathcal{T}_{\mathsf{param}} \rangle$, 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, consisting of a finite collection of parametrized transitions. We assume all transitions in $\mathcal{T}_{\mathsf{param}}$ are fair. The intention of our notion of parametrized transition systems is not to define program runs directly (as in the case of non-parametrized FTSs and runs) 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 be able to refer symbolically to thread identifiers in an arbitrary instantiation of the parametrized system, we introduce a new sort tid interpreted as an unbounded discrete set of values. The signature Σ_{tid} contains only $=$ and $\neq$, and no constructor. Theory T_{tid} is the theory of thread identifiers defined over the signature Σ_{tid}. We also extend the theory T_{prog}—used to reason about the data in the program—with T_{A}, the theory of arrays from [10], with indices from tid and elements ranging over sorts σ of the local variables of program P. In this manner we can model local variables as arrays indexed by thread identifiers. We use T_{param} for the union of theories T_{prog}, T_{tid} and T_{A}, and we use 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 $\mathsf{array}\langle\sigma\rangle$. For example, we introduce a_{pc} of sort $\mathsf{array}\langle\mathsf{loc}\rangle$ to model 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 and write $v(k)$ instead of $a_v(k)$, and $v\{k \leftarrow e\}$ instead of $a_v\{k \leftarrow e\}$. Note that $v[0]$ is different than $v(k)$. The term $v[0]$ is an atomic term in V (for a concrete system $P[M]$) 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 T_{A} is very limited: we do not use arithmetic over indices and we do not allow nested arrays, so the conditions for decidability in [10] are trivially met. Variables of sort tid indexing arrays play a special role, so we classify formulas depending on the set 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 our formulas 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 ℓ ranges over location *Locs*, and *Locs* is finite.
- **Parametrized Initial Condition**. We similarly define the parametrized initial condition for a given set of thread identifiers X as follows:

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

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

Example 3 Consider again program MutExc. The parametrized transition $\tau_4^{(k)}$, for thread k executing line 4, is the following formula from $F_{\mathsf{param}}(\{k\})$:

$$pc(k) = 4 \;\wedge\; pc' = pc\{k \leftarrow 5\} \;\wedge\; \big(bag.min = ticket(k)\big) \;\wedge\; pres(ticket, bag, avail)$$

where $pc\{k \leftarrow 5\}$ denotes the array that results from pc by replacing the element at position k with 5, as we introduced in page 10. Additionally, $pres(bag, avail, ticket)$ 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 MutExc for two thread identifiers i and j is the formula $\Theta_{\mathsf{param}}(\{i, j\})$:

$$avail = 0 \;\wedge\; bag = \emptyset \;\wedge\; \begin{pmatrix} ticket(i) = 0 \;\wedge \\ pc(i) = 1 \end{pmatrix} \;\wedge\; \begin{pmatrix} ticket(j) = 0 \;\wedge \\ pc(j) = 1 \end{pmatrix} \qquad (2)$$

□

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 are not relevant.

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 ($\bigcirc$, $\mathcal{U}$, $\Box$, etc). For example, the following formula mutex is a 2-index safety formula which expresses mutual exclusion for program MutExc:

$$\mathsf{mutex}(i, j) \quad \stackrel{\text{def}}{=} \quad \Box(i \neq j \rightarrow \neg(pc(i) \in \{5, 6\} \wedge pc(j) \in \{5, 6\})) \qquad (3)$$

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

$$\mathsf{eventually\_critical}(i) \quad \stackrel{\text{def}}{=} \quad \Box\, \big(pc(i) = 3 \rightarrow \Diamond pc(i) = 5\big) \qquad (4)$$

3.4 Parametrized Temporal Verification

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

Definition 1 (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 $[M]$:

$$\alpha : \mathit{Voc}(\varphi) \to [M]$$

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 $P[M]$. We use $\mathit{Arr}^{\varphi}_{M}$ for the set of concretizations of φ and M. Note that $\mathit{Arr}^{\varphi}_{M}$ is a finite set because φ has a finite number of thread identifiers and M is a finite number, so the resulting set of functions from $\mathit{Voc}(\varphi)$ into $[M]$ is finite.

We now describe how a concretization α can be lifted inductively to convert Σ_{param} expressions (parametrized expressions) into Σ_{prog} expressions (non-parametrized expressions for $P[M]$). 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}:

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

To define the effect of the concretization on symbols in the theory of arrays, we first translate all literals of sort array in a formula φ to either:

(a) comparisons between variables of sort array ($w = v$ or $w \neq v$), 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. For example, consider the following formula that contains a literal $(A\{j \leftarrow 0\} \neq B\{k \leftarrow 1\})$ that compares two complex array expressions:

$$\varphi_1 \quad : \quad (x > y \vee A\{j \leftarrow 0\} \neq B\{k \leftarrow 1\})$$

This formula is simplified introducing the fresh variables v_1 and v_2:

$$\varphi_2 \quad : \quad v_1 = A\{j \leftarrow 0\} \wedge v_2 = B\{k \leftarrow 1\} \wedge (x > y \vee v_1 \neq v_2)$$

The formulas φ_1 and φ_2 are equivalent, and φ_2 is in the desired form.

Now, we lift α to expressions by considering the only two remaining array cases. First, variables of array types are mapped to their corresponding concrete variable, according to α:

$$\alpha(v(k_i)) \mapsto v[\alpha(k_i)]$$

Then, for comparisons between array variables (literals (a) above):

$$\alpha(w = v) \mapsto \bigwedge_{a \in M} w[a] = v[a]$$
$$\alpha(v \neq w) \mapsto v[k] \neq w[k] \text{ for a fresh } k$$

For array update literals (case (b) above):

$$\alpha(w = v\{k \leftarrow e\}) \mapsto \big(w[\alpha(k)] = e \wedge \bigwedge_{a \in M \setminus \alpha(k)} w[a] = v[a]\big)$$

Finally, this translation can be extended to formulas in the usual manner, extending it homomorphically to Boolean and temporal connectives.

Let ψ be a parametrized formula and $\alpha(\psi)$ a concretization of ψ for a given number of threads M. For fully symmetric systems, it is easy to see [48], that if ψ is valid then $\alpha(\psi)$ is also a valid formula.

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 a 1-index parametrized formula, then $\varphi(i)$ is the formula obtained from $\varphi(t)$ by replacing all occurrences of t by i. Note that $\varphi(i)$ is still a parametrized formula. Instead, $\varphi[3]$ denotes formula φ where all occurrences of t has been instantiated for the concrete thread identifier 3. The same notation applies for transitions. For instance, $\tau_\ell^{(t)}$ is the transition relation associated with program location ℓ for an arbitrary thread t, while $\tau_\ell^{[3]}$ is the same transition relation but instantiated for the concrete thread identifier 3.

Example 4 Consider the formula $\Theta_{\mathsf{param}}(\{i,j\})$ shown as (2) in Example 3 above:

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

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 (1) in Example 2:

$$\Theta_g : avail = 0 \ \wedge \ bag = \emptyset \qquad \begin{array}{l} \Theta_l[0] : ticket[0] = 0 \ \wedge \ pc[0] = 1 \\ \Theta_l[1] : ticket[1] = 0 \ \wedge \ pc[1] = 1 \end{array}$$

Similarly, if we consider the formula mutex from (3):

$$\mathsf{mutex}(i,j) = \Box(i \neq j \rightarrow \neg(pc(i) \in \{5,6\} \wedge pc(j) \in \{5,6\}))$$

The concretization of $\mathsf{mutex}(i,j)$ according to the map $\alpha_1 : \{i \rightarrow 0, j \rightarrow 1\}$ is:

$$\alpha_1(\mathsf{mutex}) = \Box\neg(pc[0] \in \{5,6\} \wedge pc[1] \in \{5,6\})$$

On the other, the concretization map $\alpha_2 : \{i \rightarrow 0, j \rightarrow 0\}$ maps mutex into:

$$\alpha_2(\mathsf{mutex}) = \Box\big(0 \neq 0 \rightarrow \neg(pc[0] \in \{5,6\} \wedge pc[1] \in \{5,6\})\big)$$

which is equivalent to $\Box true$. □

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

Definition 2 (Parametrized Temporal Verification Problem) Given a parametrized system P and parametrized temporal formula $\varphi(\overline{k})$—with parameters $\overline{k}$—we say that $P \vDash \varphi(\overline{k})$ whenever for all concrete instances $P[M]$ and concretizations α, $P[M] \vDash \alpha(\varphi(\overline{k}))$.

4 Parametrized Verification Diagrams

Parametrized verification diagrams are an effective method to solve the parametrized temporal verification problem, particularly for liveness temporal properties. In this section, we formally define PVDs. 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 fair runs of $P[M]$. In turn, fair runs of the diagram are covered by the executions allowed by the temporal formula.

4.1 Definition of Parametrized Verification Diagrams

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

- **Nodes**. N is a finite set of nodes, and $N_0 \subseteq N$ is the subset of initial nodes.
- **Boxes**. $\mathcal{B}$ is a finite collection of pairs $\{(\mathcal{B}_1, b_1), \ldots, (\mathcal{B}_q, b_q)\}$ where $\mathcal{B}_i \subseteq N$, $\mathcal{B}_i \cap \mathcal{B}_j = \emptyset$ for $i \neq j$, and b_i is a fresh variable of sort thread identifier. Each pair $(\mathcal{B}_i, b_i)$ is called a *box* and the set $V_{box} = \{b_1, \ldots, b_q\}$ is called the set of box variables. Boxes group nodes and label them with thread identifiers. The intended meaning of boxes is to capture intervals of the computation in which some thread, not necessarily a thread referred to in the formula φ, plays a significant role in the part of the proof corresponding to the interval. Finally, we use V_{tid} 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 the PVD. That is, $V_{tid} = \overline{k} \cup V_{box}$.
- **Edges**. E is a finite set of edges. Edges are equipped with the following supporting functions and predicates:
  - $in : E \to N$ and $out : E \to N$, which given an edge provide the incoming and outgoing node.
  - $within \subseteq E$, is a predicate which indicates whether a transition modeled by an edge which connects two nodes that belong to the same box must preserve the box variable. 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 $out(e) \in \mathcal{B}_i$. If an edge e is in *within*, the semantics of PVDs will force the preservation of the box variable for all transitions that correspond to edge e. If an edge e for which both the $in(e)$ and $out(e)$ are in the same box, but $e \notin within$, a transition that corresponds to edge e can assign different values to the box variable in the pre and post-states.

- **Node labeling**: μ is a function that assigns to each node n a formula $\mu(n)$ in $F_{\mathsf{param}}(\overline{k} \cup V_{box})$, with the restriction that $\mu(n)$ can only contain b_i whenever node n is in box $\mathcal{B}_i$. The intended meaning of the label of node n is to abstract the set of states of a concrete system that satisfy the formula $\mu(n)$.
- **Edge labeling**: $\eta : E \rightharpoonup \mathcal{T} \times V_{tid}$ is a partial function labeling some edges with transitions of the system to indicate that these edges label fair transitions. The intended meaning is that the transitions labeling an edge will be eventually taken due to fairness. In other words, an execution cannot get stuck in a strongly connected component of the diagram if there is an outgoing edge from all nodes of the component labeled by η with the same variable. In particular, a computation cannot get stuck in a self-loop if the node contains an outgoing labeled edge. There are some restrictions for the labels of edges. The map η can label an edge with a transition parametrized by a thread identifier taken from the thread identifiers in the formula. Alternatively, the transition can be parametrized by a box variable b_i only when the edge begins at a node belonging to the box $(\mathcal{B}_i, b_i)$.
- **Acceptance**. The intention of the acceptance condition is to rule out infinite loops in the execution whose termination is explained by the way the program manipulates data. $\mathcal{F}$ is the acceptance condition of the diagram, which consists of a finite collection of triplets:

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

  Each triplet in the acceptance condition $\langle B_j, G_j, \delta_j\rangle$ is formed by an edge-Streett condition $B_j, G_j \subseteq E$ and a ranking function $\delta_j : N \to \mathcal{O}$, where $\mathcal{O}$ is a well founded domain. The edge-Streett condition $\langle B_j, G_j, \delta_j\rangle$ indicates that in every path some edge in G_j is visited infinitely often or all edges in B_j are visited only finitely often. Without loss of generality we can assume $G_j \cap B_j = \emptyset$. Edges in G_j are called *good* edges, and edges in B_j are called *bad* edges.
- **Propositions**. The function f maps nodes into Boolean combinations of elementary propositions from $\varphi(k)$.

Informally, parametrized verification diagrams can model an infinite collection of non-parametrized generalized verification diagrams, one per parameter M, as follows. Once a parameter instance M is fixed, every box can be populated M times, assigning in each expansion one of the possible thread identifier values within $[M]$ to the box variable. Similarly, edges incoming or outgoing boxes are populated to connect the corresponding concretized instances, and every edge label is populated into the corresponding transitions of the concrete system. Following this intuition, PVDs concisely model an infinite family of GVDs, one per value of M. The proof of correctness of PVDs can be reduced to showing that for every M, the corresponding GVD proves that $P[M]$ satisfies all possible concretizations of the specification φ. Instead, we choose to follow a shorter and more intuitive proof of soundness, showing that finding a valid parametrized diagram solves the parametrized verification problem.

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

Given a concretization function $\alpha : \overline{k} \to [M]$ for some concrete system $P[M]$ and a path π of the diagram, we define an extended concretization of the path as a sequence of functions $\alpha_i : (\overline{k} \cup V_{box}) \to [M]$ that coincide with α on all $k \in \overline{k}$, and such that if $e_i \in \mathit{within}$ then $\alpha_{i+1}(b) = \alpha_i(b)$ where b is the box variable of n_i. Essentially, the basic concretization (that chooses concrete variables of the threads mentioned in the formula) is rigid (constant throughout the execution) and the extended concretizations choose concrete indices for the box variables whenever these are free to be chosen. The predicate *within* forces an edge to preserve the corresponding box variable.

Given a run $\pi : s_0\tau_0 s_1\tau_1 \ldots$ of a concrete instance $P[M]$ and a concretization $\alpha : \overline{k} \to M$, 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:

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

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 $P[M]$ accepted by $\mathcal{D}$).

A PVD encodes a proof of the parametrized temporal verification problem. Checking the proof encoded by the diagram requires two activities, described in detail in the next sub-section:

1. Check the validity of a finite collection of verification conditions, which are automatically generated from the program and the diagram. The validity of these VCs guarantees that the diagram covers all computations of the system.
2. Check that every computation of the diagram satisfies the temporal property.

We will show that the first activity implies that, for every M, all computations of $P[M]$ are in $\mathcal{L}^{[M]}(\mathcal{D})$. In other words, that the diagram is a (fair) abstraction of the system for all instantiations.

Given a concrete instance $P[M]$ and a concretization $\alpha : \overline{k} \to [M]$, a sequence $P_0 P_1 \ldots$ 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 $\pi : n_0 e_0 n_1 \ldots$ 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 $P[M]$). Again, we will show that checking all VCs implies that for all $P[M]$ and concretizations α, every sequence of elementary propositions of a run of $P[M]$ is included in $\mathcal{L}_p^{[M]}(\mathcal{D})$. We use $\mathcal{L}^{[M]}(\varphi)$ for $\cup_{\alpha:\overline{k}\to[M]}\mathcal{L}(\alpha(\varphi))$.

For the second activity we will show in Section 4.3 below how to construct a pair of non-deterministic Büchi automata on words. This construction allows to use finite-state model checking to check whether every trace in $\mathcal{L}_p^{[M]}(\mathcal{D})$ is a trace included in the language $\alpha(\varphi)$ for every concretization α. That is, $\mathcal{L}_p^{[M]}(\mathcal{D}) \subseteq \mathcal{L}^{[M]}(\varphi)$. In other words, that all traces in the diagram satisfy the temporal formula.

4.2 Verification Conditions

In this section we show how to discharge from $\mathcal{D}$ a *finite* collection of verification conditions whose validity guarantees that $\mathcal{L}(P[M]) \subseteq \mathcal{L}^{[M]}(\mathcal{D})$. Recall that the

Given $P[M]$, $\varphi(\overline{k})$ and $\mathcal{D}$, $\mathcal{D}$ shows that $P[M] \vDash \varphi(\overline{k})$ whenever all these conditions hold:

Initiation:

(Init) $$\Theta \rightarrow \mu(N_0)$$

Consecution: for every node $n \in N$, with $\mathcal{V} = NVoc(n)$:

(SelfConsec) $$\bigvee_{n \rightarrow_e m} \mu(n) \wedge \tau(i) \wedge boxed(e) \rightarrow \mu'(m) \qquad \text{for all } i \in \mathcal{V}$$

(OtherConsec) $$\bigvee_{n \rightarrow_e m} \mu(n) \wedge \tau(j) \wedge boxed(e) \wedge \bigwedge_{i \in \mathcal{V}} i \neq j \rightarrow \mu'(m) \qquad \text{for a fresh } j \notin \mathcal{V}$$

Acceptance: for each $(B, G, \delta) \in \mathcal{F}$ and edge $n \rightarrow_e m$. Let $\mathcal{V} = NVoc(n)$,

(SelfAcc) for all $i \in \mathcal{V}$

$$\Big(\mu(n) \wedge \tau(i) \wedge \mu'(m) \wedge boxed(e) \Big) \rightarrow \delta(n) \succ \delta(m) \qquad \text{if } e \in B$$
$$\Big(\mu(n) \wedge \tau(i) \wedge \mu'(m) \wedge boxed(e) \Big) \rightarrow \delta(n) \succeq \delta(m) \qquad \text{if } e \in E \setminus (G \cup B)$$

(OtherAcc) for a fresh $j \notin \mathcal{V}$

$$\Big(\mu(n) \wedge \tau(j) \wedge \bigwedge_{i \in \mathcal{V}} i \neq j \wedge \mu'(m) \wedge boxed(e) \Big) \rightarrow \delta(n) \succ \delta(m) \qquad \text{if } e \in B$$
$$\Big(\mu(n) \wedge \tau(j) \wedge \bigwedge_{i \in \mathcal{V}} i \neq j \wedge \mu'(m) \wedge boxed(e) \Big) \rightarrow \delta(n) \succeq \delta(m) \qquad \text{if } e \in E \setminus (G \cup B)$$

Fairness: for each edge $e = (n, m, p)$ and $\tau(i) = \eta(e)$:

(En) $$\mu(n) \rightarrow En(\tau(i))$$

(Succ) $$\mu(n) \wedge \tau(i) \rightarrow \bigvee_{\tau(i) = \eta(n \rightarrow_e m)} \mu'(m)$$

Satisfaction:

(Prop) $$\mu(n) \rightarrow f(n) \qquad \text{for all } n \in N$$

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

Fig. 3: Verification conditions for parametrized verification diagrams.

vocabulary of a formula φ, denoted by $Voc(\varphi)$ is the set of free variables of type tid appearing in φ. We use $Voc(x_1, \ldots, x_n)$ to denote $\bigcup_{i=1}^{n} Voc(x_i)$. 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 define the vocabulary of a node as: $NVoc(n) = \{b_i \mid n \in \mathcal{B}_i\} \cup \overline{k}$.

Given a parametrized transition system $P[M]$, a parametrized temporal formula $\varphi(\overline{k})$ and a parametrized verification diagram $\mathcal{D}$, Fig. 3 presents the verification conditions generated from the diagram. Given an edge $e \in E$ such that $in(e), out(e) \in \mathcal{B}_i$, we define the auxiliary predicate $boxed(e)$ as follows:

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

Recall that $within \subseteq E$, introduced in Section 4.1, captures the set of edges within a box that must preserve the variable parametrizing the box. The formula $boxed(e)$ captures this constraint logically. Also, $\tau(i)$ is the formula obtained from the transition relation τ by replacing all occurrences of local variables $v[i]$ by parameters $v(i)$, and all occurrences of $v[i]'$ by $v'(i)$.

We now describe the VCs discharged from the diagram:

- **Initiation**. Verification condition (Init), known as *initiation*, states that at least one initial node in N_0 satisfies the initial condition of $\mathcal{P}$.
- **Consecution**[2]. These VCs ensure that every node in the diagram has a τ-successor, consequently for every concrete run of a concrete system there is a trail in the diagram. Condition (SelfConsec), called *self-consecution*, establishes that if the system is in a state modeled by $\mu(n)$ and a thread mentioned in the formula or in the node takes a transition, a successor node of n models the resulting state. In other words, the diagram can always move when taking any enabled transition by any thread mentioned in the property or in the node. Condition (OtherConsec), called *others-consecution*, is analogous to condition (SelfConsec), with the difference that this condition considers transitions taken by an arbitrary thread not mentioned in the vocabulary of $\varphi(\overline{k})$ or used as argument in any of the boxes in the diagram. This condition is the key to guarantee that only a finite number of verification conditions are necessary because this condition encompasses all other threads.
- **Acceptance**. Conditions (SelfAcc) and (OtherAcc), called *self-acceptance* and *others-acceptance* respectively, guarantee the acceptance condition of the diagram using ranking functions. Intuitively, these verification conditions encode information about the way the program manipulates its data to ensure 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_j is (strictly) decreasing in B_j-edges, and non-increasing in edges $E \setminus (G_j \cup B_j)$. We use A_j to denote edges in $E \setminus (G_j \cup B_j)$, called *allowed edges.*
  If the verification conditions for δ are valid, infinite trails either traverse G_j edges infinitely often, or traverse B_j-edges only finitely often. This second case holds because: (a) the domain of δ is well-founded, (b) allowed edges are non-increasing, and (c) bad edges are decreasing.
- **Fairness**. For *fairness*, condition (En) establishes that any transition labeling an edge coming out from a node must be enabled in every state modeled by the node. On the other hand, condition (Succ) establishes that if a transition labeling an edge is taken at the incoming node, then the outgoing edges with the same label cover all possible effects of the transition. The combination of (En) and (Succ) guarantee that a label τ is always enabled at the incoming nodes and that the labeled edges cover all effects of the transition. These two conditions relate fairness in any concrete system with fairness in the diagram, showing that fair trails cover all fair paths.
- **Satisfaction**. Finally, *satisfaction* ensures that the diagram satisfies the temporal parametrized specification $\varphi(\overline{k})$. Condition (Prop) guarantees the correctness of the propositional models of the diagram, which map trails in the diagram with the corresponding sequences of atomic actions from the property, that is, the propositional models of the diagram. Condition (ModelCheck) ensures that all propositional models of the diagram are included in traces of the property $\varphi(\overline{k})$. The propositional label f of the diagram allows to use a single query to a finite-state model checker to automatically decide whether

[2] The term "*consecution*" was introduced by Zohar Manna in order to describe the relation between two consecutive states of a computation through a transition relation, see [33].

condition (ModelCheck) is satisfied. We show in Section 4.3 how to leverage finite-state model checkers to decide whether $\mathcal{L}_p^{[M]}(\mathcal{D}) \subseteq \mathcal{L}^{[M]}(\varphi)$.

Given a parametrized system $P[M]$, a formula φ and a PVD $\mathcal{D}$, if all verification conditions described above hold we say that $\mathcal{D}$ is (P, φ)-valid. Note that there is a finite number of verification conditions. In particular, we need to verify $|N|(|V_{tid}|+1)$ conditions for *consecution* and at most $|\mathcal{F}||E|(|V_{tid}|+1)$ conditions for *acceptance*. The number of conditions discharged for *fairness* is limited by the number of edges, program lines and thread identifiers in the vocabulary of the formulas labeling nodes in each box.

4.3 Model Checking that a PVD Satisfies a Temporal Property

All conditions described in Section 4.2, except from condition (ModelCheck), can be automatically verified using appropriate decision procedures. We now show how to check automatically that $\mathcal{L}_p^{[M]}(\mathcal{D}) \subseteq \mathcal{L}^{[M]}(\varphi)$, given a temporal property φ. The first step is to extract the propositional alphabet from φ. Then, we show how to construct two non-deterministic Büchi automata on words (NBW for short):

- $\mathcal{A}_{\mathcal{D}}$, which captures the propositional models 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 $\mathcal{L}_p^{[M]}(\mathcal{D}) \subseteq \mathcal{L}^{[M]}(\varphi)$. We now show how to build an NBW for the propositional traces of the diagram. The reason to choose an edge-Streett acceptance condition was to ease in the manual creation of PVDs for specific proofs. We show here an automata construction to translate an edge-Streett automaton into an NBW.

The Intended Meaning of $\mathcal{F}$. Conditions (SelfAcc) and (OtherAcc) imply that the ranking function δ_j is (strictly) decreasing in B_j-edges, and non-increasing in A_j- edges (that is, edges in $E \setminus (G_j \cup B_j)$).

Without loss of generality we assume that $G_j \cap B_j = \emptyset$. Otherwise consider the pair $\langle G_j, B'_j \rangle$ with $B'_j = B_j \setminus G_j$. The pair $\langle G_j, B'_j \rangle$ satisfies that:
- $G_j \cap B'_j = \emptyset$, and
- every trail π is accepting for (G_j, B_j) precisely when it is accepting for (G_j, B'_j). To see this last claim, observe that a good trail that traverses G_j-edges infinitely often is accepted for both. Also, if a trail visits all B_j-edges only finitely often, then the trail visits all B'_j-finitely often. For the other direction, consider the cases:
  1. if a trail visits all edges in B'_j only finitely often but some edge in $B_j \setminus B'_j$ infinitely often, then some edge in G_j is seen infinitely (because all edges in $B_j \setminus B'_j$ are G_j-edges), and the trail is accepting in both cases again.
  2. the last case is that all edges in B'_j are traversed only finitely and all edges in $B_j \setminus B'_j$ are also traversed only finitely often, which again is accepting for both.

Edge-Streett Automaton on Words. We define now ESNW (for edge-Streett non-deterministic automaton on words) 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, $F = \langle\langle B_1, G_1\rangle, \ldots, \langle B_m, G_m\rangle\rangle$ 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 $s_0t_0s_1t_1\ldots$ 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 $\mathit{inf}_T(\pi)$, and the set of states that occur infinitely often in π is $\mathit{inf}_Q(\pi)$. A trace π of an ESNW is *accepting* whenever for all $1 \leq j \leq m$, either:
- $\mathit{inf}_T(\pi) \cap G_j \neq \emptyset$, or
- $\mathit{inf}_T(\pi) \cap B_j = \emptyset$.

From ESNW into NBW. 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, and $F \subseteq Q$. An infinite trace π of the automaton is accepting if $\mathit{inf}_Q(\pi) \cap F \neq \emptyset$, that is, if some state in F is present infinitely often in π.

The translation from ESNW to NBW works as follows. Given an ESNW $\mathcal{E}$ we first generate an NBW $\mathcal{A}_j$ for each edge-Streett pair (B_j, G_j) separately. The automaton $\mathcal{A}_j$ is defined as a composition of two sub-automata $\mathcal{A}_1$ and $\mathcal{A}_2$, which we describe first:

1. **Automaton $\mathcal{A}_1$**. The automaton $\mathcal{A}_1 : \langle AP, Q_1, I_1, L_1, T_1, F_1\rangle$ is:
   - 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, and q_1 encodes that a good edge was not taken in the transition that reaches q in $\mathcal{E}$.
   - 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 $L_1(q_1^G) = L(q)$.

   The first sub-automaton $\mathcal{A}_1$ checks whether the corresponding trace in $\mathcal{E}$ traverses good edges infinitely often, because 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.
2. **Automaton $\mathcal{A}_2$**. The second component of $\mathcal{A}_j$ is $\mathcal{A}_2 : \langle AP, Q_2, I_2, L_2, T_2, F_2\rangle$:
   - Q_2 contains one state q_2 for each state q in Q.
   - T_2 contains an edge $p_2 \to q_2$ whenever there is a non-B edge $p \to q$ in T.
   - $I_2 = \emptyset$.
   - $L_2(q_2) = L(q)$.

  - $F_2 = Q_2$: all states are accepting.

  The automaton $\mathcal{A}_2$ captures all suffixes of traces that do not visit any bad edge.

3. **Automaton** $\mathcal{A}_j$. The automaton $\mathcal{A}_j : \langle AP, Q^j, I^j, L^j, T^j, F^j \rangle$ is defined as follows:
   - $Q^j = Q_1 \cup Q_2$,
   - The transitions T^j are:

$$T^j = T_1 \cup T_2 \cup \begin{array}{l} \{p_1 \to q_2 | \text{for every edge } p \to q \text{ in } T\} \\ \{p_1^G \to q_2 | \text{for every edge } p \to q \text{ in } T\} \end{array}$$

   The additional edges allow A_j to jump from $\mathcal{A}_1$ into $\mathcal{A}_2$.
   - $I^j = I_1$,
   - $L^j(q) = L_1(q)$ if $q \in Q_1$, and $L^j(q) = L_2(q)$ if $q \in Q_2$,
   - $F^j = F_1 \cup F_2$.

Note that in $\mathcal{A}_j$ there are no edges back into $\mathcal{A}_1$ from $\mathcal{A}_2$, so if one of the jump edges 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 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 trace.

The construction described so far allows to translate an ESNW with only one pair (B, G) of edge-Streett conditions into an NBW. The size of the generated automaton is $(|Q| + |G|) + |Q|$, where $|Q| + |G|$ corresponds to $\mathcal{A}_1$ and the last $|Q|$ to $\mathcal{A}_2$.

In order to create a single NBW that captures the general case of k edge-Streett conditions, it is tempting to construct an alternating automaton by merging all NBW computed for each of the individual accepting conditions and simply letting $I = \bigwedge_{j=1}^{k} I_j$. The resulting automaton is an alternating Büchi automaton that can be easily converted into an NBW using standard constructions. Unfortunately, this construction is not correct because the different copies of the alternating automaton can potentially move when reading the same symbol to different states q_i and q_j of the original ESNW automaton, which does not correspond to a simulation of a trace of the original automaton. Instead, we present here a translation where we build a unique NBW in which every component is forced to be in the same state in the automata for different ESNW accepting conditions.

The construction for the NBW $\mathcal{A}$ for the full ESNW starts from the NBWs $\mathcal{A}^j$ obtained by translating each of the edge-Streett conditions. The resulting automaton is $\mathcal{A}_\mathcal{E} : \langle AP, Q^f, I, L, T, F \rangle$ where:

- $Q^f = (Q \times 3^k \times 2^k)$. A state (q, v, o) represents the state of all sub-component automata:
  - every sub-automaton is in state q.
  - automaton $\mathcal{A}^j$ is in state q_1 (if $v[j] = 0$), in state q_1^G (if $v[j] = 1$) or in state q_2 (if $v[j] = 2$);
  - automaton $\mathcal{A}$ "owes" a visit to a final state (if $o[i] = 1$), or has already visited a final state since the last reset (see F below). Essentially, this field records which components have visited Büchi accepting states. When all

components have visited accepting states, the vector o is reset and the corresponding state declared accepting. This guarantees that all sub-automata visit accepting states infinitely often precisely when the vector is reset infinitely often.

- $L((q,v,o)) = L(q)$.
- I contains (q,v,o) for every $q \in I$, with $v[i] = 0$ and $o[i] = 1$ for all i.
- T contains a transition $(q,v,o) \rightarrow (p,v',o')$ whenever, for all i, one of the following hold:
  - $(q_1 \rightarrow p_1) \in T^i$, $v[i] = 0$ and $v'[i] = 0$.
  - $(q_1 \rightarrow p_1^G) \in T^i$, $v[i] = 0$ and $v'[i] = 1$.
  - $(q_1^G \rightarrow p_1) \in T^i$, $v[i] = 1$ and $v'[i] = 0$.
  - $(q_1^G \rightarrow p_1^G) \in T^i$, $v[i] = 1$ and $v'[i] = 1$.
  - $(q_1 \rightarrow p_2) \in T^i$, $v[i] = 0$ and $v'[i] = 2$.
  - $(q_1^G \rightarrow p_2) \in T^i$, $v[i] = 1$ and $v'[i] = 2$.
  - $(q_2 \rightarrow p_2) \in T^i$, $v[i] = 2$ 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 when $v'[i] = 1$ or $v'[1] = 2$.
- $F = \{(q,v,o) \mid \text{for all } i, o[i] = 0\}$

The choice of a single state q forces all sub-automata to be in the same state. The component v allows to distinguish, for each sub-automaton, the active version of state q (either q_1, q_1^G or q_2). The owing vector o is used to remember which sub-automaton has visited a final state since the last visit to a global final state. A global final state (i.e., a final state of the resulting NBW) occurs when all sub-automata have visited a local final state, which is captured by the owing vector being 0 at all positions. This guarantees that between two global final states every sub-automaton has visited a final state, and hence all sub-automata accept the trace. Also, if all sub-automata visit a final state infinitely often, it follows that a global final state will be visited infinitely often. The use of owe sets was first introduced in the Miyano-Hayashi construction [36] to translate alternating automata into non-deterministic automata. It is easy to check that $\mathcal{A}_{\mathcal{E}}$ and $\mathcal{E}$ accept the same language.

4.4 Soundness of Parametrized Verification Diagrams

We now present the main result of this paper.

Theorem 1 (Soundness) *Let P be a parametrized system and $\varphi(\overline{k})$ a temporal formula. If there exists a $(P,\varphi)-$valid* PVD*, then $P \vDash \varphi$.*

Proof We start by assuming that there is a $(\mathcal{P},\varphi)-$*valid* PVD $\mathcal{D}$, and show that $P \vDash \varphi$. This requires showing $P[M] \vDash \alpha(\varphi(\overline{k}))$ for an arbitrary M and concretization $\alpha : \overline{k} \rightarrow [M]$. In the proof, we will use repeatedly the following fact for symmetric systems. Given a parametrized non-temporal formula $\psi(\overline{k})$ and a concretization α, if $\psi(\overline{k})$ is valid, then $\alpha(\psi(\overline{k}))$ is also valid (see [48]).

Let M be an arbitrary bound and α an arbitrary concretization function. We consider an arbitrary run (that is, a fair computation) of $P[M]$: $\sigma : s_0\tau_0[i_0]s_1\tau_1[i_1]\ldots$ 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 α and making $\alpha(i) = k$. In this manner, all elements of M have at least one representative thread identifier (not necessarily in $\overline{k}$, but perhaps a fresh one).

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 $\alpha(j_k) = i_k$ and $s_i \vDash \alpha(\mu(n_i))$. The concretization α maps j_k into the concrete identifier of the thread taking the k-th step in the computation $s_k\tau_k[i_k]s_{k+1}$. It is enough to prove that there is a trail of nodes n_k of the diagram and an extended concretization α_k such that:

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

Note that the last formula implies that in every state that is captured by $\mu(n_k)$ (such as s_k) all moves of any instance of the system are considered by the diagram.

We build the trace by induction:

- Base case: The base case of induction follows from condition (Init). Since $\Theta \rightarrow \mu(N_0)$ is valid, then $\alpha(\Theta \rightarrow \mu(N_0))$ is valid, and $\alpha(\Theta) \rightarrow \alpha(\mu(N_0))$ 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:
  - Case: j_k is referred to in the property. From condition (SelfConsec) we have that the following is valid
  $$\bigvee_{n_k \rightarrow_e n_{k+1}} \mu(n_k) \wedge \tau(j_k) \wedge \mathit{boxed}(e) \rightarrow \mu'(n_{k+1})$$
  so the following is also valid
  $$\alpha_k\Big(\bigvee_{n_k \rightarrow_e n_{k+1}} \mu(n_k) \wedge \tau(j_k) \wedge \mathit{boxed}(e) \rightarrow \mu'(n_{k+1})\Big)$$
  and, finally, the following is also valid
  $$\bigvee_{n_k \rightarrow_e n_{k+1}} \alpha_k(\mu(n_k)) \wedge \tau[i_k] \wedge \mathit{boxed}(e) \rightarrow \alpha_k(\mu'(n_{k+1})))$$
  Now, $s_k \vDash \alpha_k(\mu(n_k))$ holds by inductive hypothesis, and (s_k, s_{k+1}) is a model of the last formula (possibly for a different value of the box variable if $\mathit{boxed}(e)$ is not true). It follows that for at least one of the disjuncts $s_{k+1} \vDash \alpha_{k+1}(\mu'(n_{k+1}))$. This disjunct 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.

Up to now we have shown that, provided that the verification conditions are valid, every path of the system is captured by a trail of the diagram. We now show that the trail $\pi : n_0e_0n_1\dots$ with transitions $\tau_k^{(j_k)}$ is a fair trail of the diagram. The proof is 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 the transitions taken at position k in the path (that is, $\tau_k^{(j_k)}$) 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, that s_k has a $\tau[\alpha(i)]$ successor in $P[M]$. However, $\tau[\alpha(i)]$ is not taken in the path. Hence, σ is not a fair run of $P[M]$, 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 concludes the proof. $\square$

5 Experimental Evaluation

We have implemented PVDs as part of Leap [46], a prototype theorem prover under development at the IMDEA Software Institute. The aim of Leap is to enable the verification of safety and liveness properties of parametrized concurrent systems and in particular concurrent data types. Leap and the examples presented in this section can be downloaded from the webpage `http://software.imdea.org/leap`.

For the verification of liveness properties, Leap receives as input a program description, an intended property and a PVD. Starting from the program and the PVD, Leap automatically generates the VCs described in Section 4.2. Leap also implements internally many decision procedures, including decision procedures for Presburger Arithmetic with sets and complex pointer-based concurrent data structures like lists with locks. With the assistance of these decision procedures, Leap can automatically verify the validity of the generated verification conditions for the programs described here.

We now present the experimental results we have obtained using Leap for the verification of liveness properties for the mutual exclusion protocol presented in Section 2 and for an implementation of concurrent lock-coupling lists [27, 52]. 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.

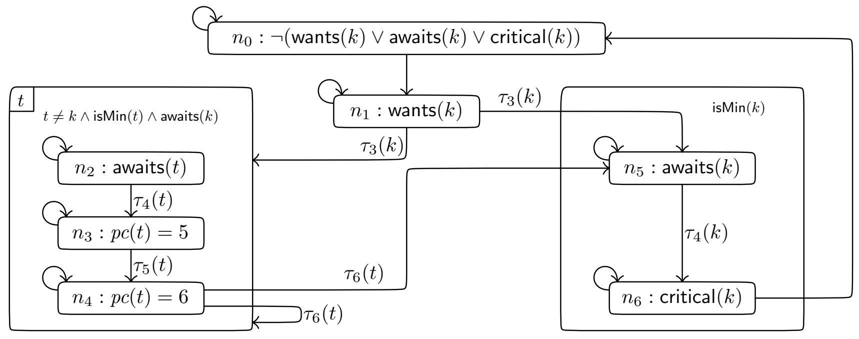

Fig. 4: PVD for the proof that MutExc satisfies eventually_critical(k).

5.1 Mutual Exclusion Protocol

First, we consider the mutual exclusion protocol MutExc introduced in Section 2. For this program, we verify the liveness property eventually_critical(k) presented in (4):

$$\text{eventually\_critical}(i) \overset{\text{def}}{=} \Box \big(pc(i) = 3 \to \Diamond pc(i) = 5\big)$$

For the verification, we use the theory of finite sets of pairs, where the elements of the pairs are integers and thread identifiers, with ordered comprehension and minimum value. Given a pair p, function $\pi_{int}(p)$ returns the integer component of p. 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 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 of being the lowest element in a set, then one of such pairs can be arbitrarily returned. It is easy to show that validity of quantifier-free formulas of this theory is decidable.

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. Note how the diagram proves the liveness property eventually_critical(k) thanks to the fact that the thread with the minimum ticket can freely progress, eventually leaving the critical section. Once the thread with the minimum ticket leaves the critical section, the thread owning the new minimum ticket is allowed to access the critical section. Since we use *bag* to keep pairs of tickets and threads, we can know the thread owning the new minimum ticket in the system. In the PVD We use the following auxiliary predicates:

$$\text{wants}(k) \overset{\text{def}}{=} pc(k) = 3$$
$$\text{awaits}(k) \overset{\text{def}}{=} pc(k) = 4$$
$$\text{critical}(k) \overset{\text{def}}{=} pc(k) = 5$$

The PVD is defined as follows:

$$
\begin{aligned}
N &\mathrel{\hat{=}} \{n_i \mid 0 \leq i \leq 6\} \\
N_0 &\mathrel{\hat{=}} \{n_0\} \\
E &\mathrel{\hat{=}} \{n_0 \to n_1, n_2 \to n_3, n_3 \to n_4, n_5 \to n_6, n_6 \to n_0\} \cup \\
&\qquad \{n_i \to n_j \mid i = 1,4 \text{ and } j = 2,3,4,5\} \cup \{n_i \to n_i \mid i = 0,1,2,3,4,5,6\} \\
\mathit{within} &\mathrel{\hat{=}} \{n_2 \to n_3, n_3 \to n_4\} \\
\mathcal{B} &\mathrel{\hat{=}} \{(\{n_2, n_3, n_4\}, t)\} \\
\mu(n_0) &\mathrel{\hat{=}} \neg(\mathsf{wants}(k) \vee \mathsf{awaits}(k) \vee \mathsf{critical}(k)) \\
\mu(n_1) &\mathrel{\hat{=}} \mathsf{wants}(k) \\
\mu(n_2) &\mathrel{\hat{=}} t \neq k \wedge \mathsf{isMin}(t) \wedge \mathsf{awaits}(k) \wedge \mathsf{awaits}(t) \\
\mu(n_3) &\mathrel{\hat{=}} t \neq k \wedge \mathsf{isMin}(t) \wedge \mathsf{awaits}(k) \wedge pc(t) = 5 \\
\mu(n_4) &\mathrel{\hat{=}} t \neq k \wedge \mathsf{isMin}(t) \wedge \mathsf{awaits}(k) \wedge pc(t) = 6 \\
\mu(n_5) &\mathrel{\hat{=}} \mathsf{isMin}(k) \wedge \mathsf{awaits}(k) \\
\mu(n_6) &\mathrel{\hat{=}} \mathsf{isMin}(k) \wedge \mathsf{critical}(k) \\
\eta(e) &\mathrel{\hat{=}} \begin{cases} (\tau_3, k) & \text{if } e \in \{n_1 \to n_i \mid i = 2,3,4,5\} \\ (\tau_4, t) & \text{if } e \in \{n_2 \to n_3\} \\ (\tau_4, k) & \text{if } e \in \{n_5 \to n_6\} \\ (\tau_5, t) & \text{if } e \in \{n_3 \to n_4\} \\ (\tau_6, t) & \text{if } e \in \{n_4 \to n_i \mid i = 2,3,4,5\} \end{cases} \\
\mathcal{F} &\mathrel{\hat{=}} \langle\langle \{n_4 \to n_i \mid i = 2,3,4,5\},\ \ \{n_6 \to n_0\},\ \ \lambda n \to \mathit{lower}(\mathit{bag}, \mathit{ticket}(k)) \rangle\rangle \\
f(n) &\mathrel{\hat{=}} \begin{cases} \neg(\mathsf{wants}(k) \vee \mathsf{critical}(k)) & \text{if } n = n_0 \\ \mathsf{wants}(k) & \text{if } n = n_1 \\ \mathsf{critical}(k) & \text{if } n = n_6 \\ \mathit{true} & \text{otherwise} \end{cases}
\end{aligned}
$$

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 has a self-loop edge to allow transitions that do not change the truth value of the node, and are not labeling any other outgoing edge. 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 $\subset$) every time the leader thread (captured by the box variable t) exits the critical section and removes its pair from the set. Note that this transition is not in *within*, which is graphically represented by an edge that leaves the box and returns back into the box. Table 1 in page 31 reports running times and the number of VCs generated using LEAP for verifying the PVD that represents the proof that program MUTEXC satisfies the $\mathsf{eventually\_critical}(k)$ property.

```
    procedure MGCList()
      elem $e$
    begin
1:  while true do
2:      $e :=$ havocListElem()
3:      nondet
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: $lock(prev \to lock)$
    $lockedSet := lockedSet \cup \{me\}$
    $lockedInsert := lockedInsert \cup \{me\}$
    if $(me = k)$ then
        $aheadSet := lockedSet$
        $aheadInsert := lockedInsert$
    endif
26: $curr := prev \to next$
27: $lock(curr \to lock)$
28: while $curr \neq null \wedge curr \to data < e$ do
29:     $aux := prev$
30:     $prev := curr$
31:     $unlock(aux \to lock)$
32:     $curr := curr \to next$
33:     $lock(curr \to lock)$
34: end while
35: if $curr \neq null \wedge curr \to data > e$ then
36:     $newnode := malloc(e)$
37:     $newnode \to next := curr$
38:     $prev \to next := newnode$
        $reg := reg \cup \{newnode\}$
        $elems := elems \cup \{e\}$
        $lockedInsert := lockedInsert \setminus \{me\}$
        $aheadInsert := aheadInsert \setminus \{me\}$
39: else
40:     skip
        $lockedInsert := lockedInsert \setminus \{me\}$
        $aheadInsert := aheadInsert \setminus \{me\}$
41: end if
42: $unlock(prev \to lock)$
43: $unlock(curr \to lock)$
    $lockedSet := lockedSet \setminus \{me\}$
    $aheadSet := aheadSet \setminus \{me\}$
44: return
    end procedure
```

```
    procedure Search(elem $e$)
      addr $prev,\ curr,\ aux$
      bool $found$
    begin
9:  $prev := head$
10: $lock(prev \to lock)$
    $lockedSet := lockedSet \cup \{me\}$
11: $curr := prev \to next$
12: $lock(curr \to lock)$
13: while $curr \to data < e$ do
14:     $aux := prev$
15:     $prev := curr$
16:     $unlock(aux \to lock)$
17:     $curr := curr \to next$
18:     $lock(curr \to lock)$
19: end while
20: $found := (curr \to data = e)$
21: $unlock(prev \to lock)$
22: $unlock(curr \to lock)$
    $lockedSet := lockedSet \setminus \{me\}$
    $aheadSet := aheadSet \setminus \{me\}$
23: return $found$
    end procedure
```

```
    procedure Remove(elem $e$)
      addr   $prev,\ curr, aux$
    begin
45: $prev := head$
46: $lock(prev \to lock)$
    $lockedSet := lockedSet \cup \{me\}$
47: $curr := prev \to next$
48: $lock(curr \to lock)$
49: while $curr \neq tail \wedge curr \to data < e$ do
50:     $aux := prev$
51:     $prev := curr$
52:     $unlock(aux \to lock)$
53:     $curr := curr \to next$
54:     $lock(curr \to lock)$
55: end while
56: if $curr \neq tail \wedge curr \to data = e$ then
57:     $aux := curr \to next$
58:     $prev \to next := aux$
        $reg := reg \setminus \{curr\}$
        $elems := elems \setminus \{e\}$
59: end if
60: $unlock(prev \to lock)$
61: $unlock(curr \to lock)$
    $lockedSet := lockedSet \setminus \{me\}$
    $aheadSet := aheadSet \setminus \{me\}$
62: return
    end procedure
```

Fig. 5: Implementation of procedures Search, Insert, Remove and MGCList for concurrent lock-coupling lists.

5.2 Concurrent Lock-Coupling Lists

The second case study is the formal verification of a progress property of lock-coupling concurrent lists. A lock-coupling concurrent list [27, 52] 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 only one thread can access any given node at the same time. However, different nodes can potentially be accessed by different threads concurrently. 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. Since new nodes can be inserted, the heap memory used can grow unboundedly.

Each node contains an element, a pointer to the next node and a lock that protects the node. These fields are accessible thorough field names *data*, *next* and *lock* respectively. Concurrent lock-coupling lists maintain two global pointers, *head* and *tail*, which point to the head and tail of the list. The data type is annotated with so-called "ghost variables" which are additional variables added for verification purposes that are removed during compilation. Ghost variables are used to store interesting aspects of the history of a given computation. These variables are updated with additional code annotated to the program, called ghost code, which is only allowed to update ghost variables. Ghost code appears in dashed boxes in the figure.

Concurrent lock-coupling lists maintain six ghost variables named *reg*, *elems*, *lockedSet*, *lockedInsert*, *aheadSet* and *aheadInsert*. Variable *reg* keeps track of the portion of the heap whose cells form the list. Variable *elems* stores the collection of elements stored in the list. Set *lockedSet* contains the set of thread identifiers that owns at least a lock in the list. In *lockedInsert* we keep the set of thread identifiers that own a lock and are executing procedure Insert but have not reached Insert's linearization point yet. Set *aheadSet* contains the set of thread identifiers that own at least one lock in the list and are ahead of thread k. Finally, *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 (at line 38).

Concurrent lock-coupling lists provide 3 main operations: Search, Insert and Remove. Procedure Search looks for an element in the list. Procedure Insert adds a new element in the list and procedure Remove deletes an element from the list. Finally, procedure MGCList acts as the most general client of the concurrent list. That is, MGCList is a procedure that non-deterministically performs calls to all the operations provided by the concurrent list data type. Fig. 5 presents the implementation of these procedures.

The progress property we want to verify is that if an arbitrary thread k attempts to insert an element in the concurrent lock-coupling list, then thread k eventually successfully inserts the element and returns. We can express this property as the following 1-index parametrized temporal formula:

$$\mathsf{eventually\_insert}(k) \quad = \quad \Box\,(pc(k) = 26 \rightarrow \Diamond pc(k) \in \{42..44\})$$

That is, whenever thread k gets its first lock in an attempt to insert an element (line 26) it eventually finishes the execution of the procedure (reaches the lines between 42 and 44 of Insert).

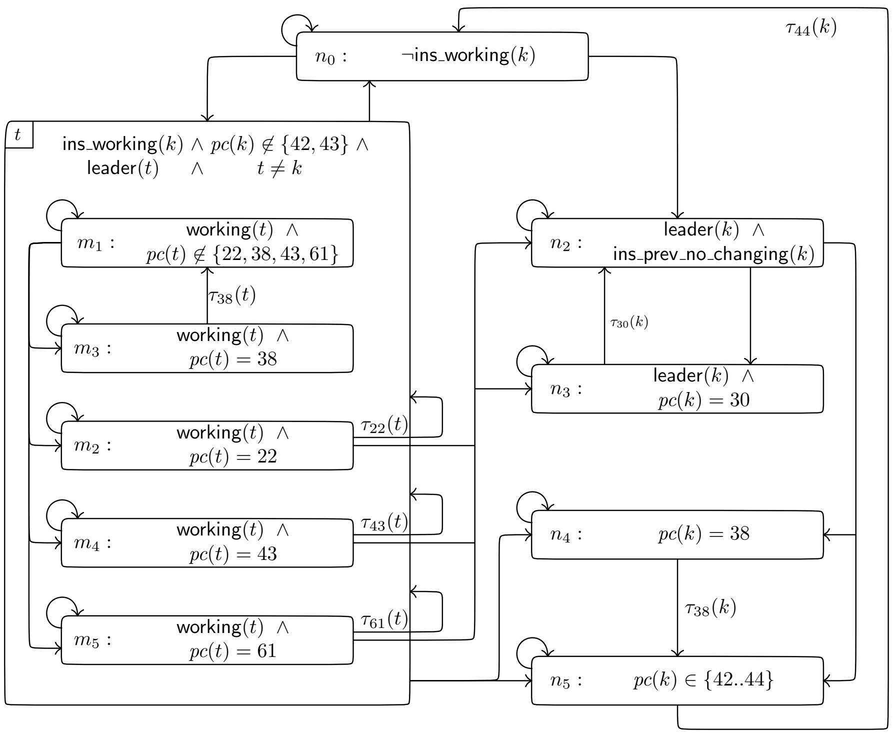
n_0 : ¬ins_working(k)
$\tau_{44}(k)$
t
ins_working(k) ∧ $pc(k) \notin \{42, 43\}$ ∧ leader(t) ∧ $t \neq k$
m_1 : working(t) ∧ $pc(t) \notin \{22, 38, 43, 61\}$
$\tau_{38}(t)$
m_3 : working(t) ∧ $pc(t) = 38$
m_2 : working(t) ∧ $pc(t) = 22$
$\tau_{22}(t)$
m_4 : working(t) ∧ $pc(t) = 43$
$\tau_{43}(t)$
m_5 : working(t) ∧ $pc(t) = 61$
$\tau_{61}(t)$
n_2 : leader(k) ∧ ins_prev_no_changing(k)
$\tau_{30}(k)$
n_3 : leader(k) ∧ $pc(k) = 30$
n_4 : $pc(k) = 38$
$\tau_{38}(k)$
n_5 : $pc(k) \in \{42..44\}$

Fig. 6: PVD for proving eventually_insert(k) in concurrent lock-coupling lists.

Fig. 6 shows the parametrized verification diagram that encodes the proof that any system composed by an unbounded number of threads executing the program shown in Fig. 5 satisfies property eventually_insert(k).

In the diagram, we use the following abbreviations:

$$\begin{aligned}
\mathsf{sch\_working}(i) &\overset{\text{def}}{=} pc(i) \in \{11..22\}\\
\mathsf{ins\_working}(i) &\overset{\text{def}}{=} pc(i) \in \{26..43\}\\
\mathsf{rem\_working}(i) &\overset{\text{def}}{=} pc(i) \in \{47..61\}\\
\mathsf{ins\_prev\_no\_changing}(i) &\overset{\text{def}}{=} pc(i) \in \{26..29, 31..37, 39..42\}\\
\mathsf{working}(i) &\overset{\text{def}}{=} (\mathsf{sch\_working}(i) \vee \mathsf{ins\_working}(i) \vee \mathsf{rem\_working}(i))
\end{aligned}$$

Labels sch_working, ins_working and rem_working refer to those program lines within procedures SEARCH, INSERT and REMOVE where the thread owns at least one lock and, thus, is performing some work on the list. Label ins_prev_no_changing refers to the program lines within procedure INSERT in which the field *next* of the node pointed by *prev* is not about to be modified. Finally, label working refers to any of the program locations labeled by sch_working, ins_working or rem_working.

Predicate 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*. When a thread i satisfies the predicate $\mathsf{leader}(i)$ we say that i is the "*leader*" thread, as it is the thread owning the lock closest to the tail of the list. The diagram is formally defined as follows:

$$
\begin{array}{rcl}
N & \hat{=} & \{n_i \mid i = 0,2,3,4,5\} \cup \{m_j \mid j = 1,2,3,4,5\} \\
N_0 & \hat{=} & \{n_0\} \\
E & \hat{=} & \{n_0 \to m_j \mid j = 1,2,3,4,5\} \cup \{m_j \to n_i \mid j = 1,2,3,4,5 \text{ and } i = 4,5\} \cup \\
 & & \{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, m_3 \to m_1\} \cup \\
 & & \{m_i \to m_j \mid i = 2,4,5 \text{ and } j = 1,2,3,4,5\} \cup \{n_i \to n_i \mid i = 0,2,3,4,5\} \cup \\
 & & \{m_j \to n_i \mid j = 2,4,5 \text{ and } i = 2,3\} \cup \{m_j \to m_j \mid j = 1,2,3,4,5\} \\
\mathit{within} & \hat{=} & \{m_1 \to m_j \mid j = 1,2,3,4,5\} \cup \{m_3 \to m_1\} \\
\mathcal{B} & \hat{=} & \Big\{(\{m_1, m_2, m_3, m_4, m_5\}, t)\Big\} \\
\mu(n_0) & \hat{=} & \neg\mathsf{ins\_working}(k) \\
\mu(m_1) & \hat{=} & \mathsf{ins\_working}(k) \wedge pc(k) \notin \{42,43\} \wedge \mathsf{leader}(t) \wedge t \neq k \wedge \mathsf{working}(t) \wedge pc(t) \notin \{22,38,43,61\} \\
\mu(m_2) & \hat{=} & \mathsf{ins\_working}(k) \wedge pc(k) \notin \{42,43\} \wedge \mathsf{leader}(t) \wedge t \neq k \wedge \mathsf{working}(t) \wedge pc(t) = 22 \\
\mu(m_3) & \hat{=} & \mathsf{ins\_working}(k) \wedge pc(k) \notin \{42,43\} \wedge \mathsf{leader}(t) \wedge t \neq k \wedge \mathsf{working}(t) \wedge pc(t) = 38 \\
\mu(m_4) & \hat{=} & \mathsf{ins\_working}(k) \wedge pc(k) \notin \{42,43\} \wedge \mathsf{leader}(t) \wedge t \neq k \wedge \mathsf{working}(t) \wedge pc(t) = 43 \\
\mu(m_5) & \hat{=} & \mathsf{ins\_working}(k) \wedge pc(k) \notin \{42,43\} \wedge \mathsf{leader}(t) \wedge t \neq k \wedge \mathsf{working}(t) \wedge pc(t) = 61 \\
\mu(n_2) & \hat{=} & \mathsf{leader}(k) \wedge pc(k) \in \{26..29, 31..37, 39..41\} \\
\mu(n_3) & \hat{=} & \mathsf{leader}(k) \wedge pc(k) = 30 \\
\mu(n_4) & \hat{=} & \mathsf{leader}(k) \wedge pc(k) = 38 \\
\mu(n_5) & \hat{=} & \mathsf{leader}(k) \wedge pc(k) \in \{42..44\} \\
\eta(e) & \hat{=} & \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} & \hat{=} & \Big\langle \langle \{m_i \to m_j \mid i = 2,4,5 \text{ and } j = 1,2,3,4,5\} \cup \{n_3 \to n_2, m_3 \to m_1\}\ , \\
 & & \quad \{n_0 \to m_j \mid j = 1,2,3,4,5\} \cup \{m_j \to n_i \mid j = 1,2,3,4,5 \text{ and } i = 4,5\} \cup \\
 & & \quad \{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\}\ , \\
 & & \quad \lambda n \to \langle \mathit{aheadSet}, \mathit{aheadInsert}, \mathit{addr2set}(\mathit{heap}, \textsc{Insert}::\mathit{prev}(k))\rangle\rangle \Big\rangle \\
f(n) & \hat{=} & \begin{cases} pc(k) = 26 & \text{if } n = m_1, m_2, m_3, m_4, m_5, n_2 \\ pc(k) \in \{42..44\} & \text{if } n = n_5 \\ \mathit{true} & \text{otherwise} \end{cases}
\end{array}
$$

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 acquired a lock within procedure INSERT. All nodes n_i with $i = 2, 3, 4, 5$ represent the situation in which thread k is executing procedure INSERT and thread k is the leader. In this case, thread k has no obstacle for advancing through the list.

The other case, when a 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 the leader is a thread t different from k. Note that thread t can be executing any of the procedures 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 $\tau_{61}^{(\tau)}$. When these transitions are taken, thread k becomes the new leader or a new thread becomes the leader.

We use the ghost variables *aheadSet* and *aheadInsert* for proving acceptance. The ranking function associated to the acceptance condition is a triple that follows a lexicographic order. The first component is *aheadSet*, which contains the set of thread identifiers owning a lock ahead of thread k. The idea is that *aheadSet* decreases with respect to $\subset$ every time a thread t stops being the leader (following the execution of transitions $\tau_{22}^{(t)}$, $\tau_{43}^{(t)}$ or $\tau_{61}^{(t)}$). The second component of the 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 at the third component of the ranking function, which is the distance between INSERT::*prev* and the tail of the list. As thread k progresses through the list, this distance will decrement. However, if another thread j inserts an element between the current location of INSERT::*prev* and *tail*, the distance between INSERT::*prev* and *tail* can increase when thread j executes $\tau_{38}^{(j)}$, which is the transition that effectively inserts the node. However, in this case, the

	#VC	#solved VC		single VC time(s.)		DP	LEAP
		pos	num	slowest	average	time(s)	time(s)
mutex (init)	1	0	1	0.01	0.01	0.01	0.01
mutex (consec)	153	144	9	2.66	0.03	4.22	0.06
mutex (accept)	195	132	63	1.46	0.08	15.28	0.05
mutex (fair)	24	20	4	0.03	0.01	0.10	0.02
lists (init)	1	0	1	0.01	0.01	0.01	0.01
lists (consec)	1550	1343	207	3.80	0.05	78.12	3.42
lists (accept)	5404	4352	1052	191.61	0.12	647.04	1.61
lists (fair)	48	20	28	0.42	0.16	7.82	0.14

Table 1: Running times, in seconds, for the verification eventually_critical(k) and eventually_insert(k). Each row corresponds to an activity in the proof (initiation, consecution, acceptance and fairness). The first column shows the number of generated VCs. The second shows the number of VCs solved using a very simple decision procedure that can only reason about *pc*. The third column reports the number of remaining VCs, which are proven valid using a specialized decision procedure. The forth and fifth column report the slowest and average solving time for a single VC. The sixth column shows the total time used by the DPs to solve all VCs. Finally, the last column shows the running time taken to generate all VCs.

set *aheadInsert* decrements, making the ranking function strictly decrease in the lexicographic order.

Table 1 reports the results obtained using Leap to verify eventually_insert(k) and eventually_insert(k). All verification conditions are automatically checked as valid. The verification of (ModelCheck) is reduced to a simple model checking query which is performed according to the process described in Section 4.3.

6 Related Work

Most of the work in formal verification of sequential pointer programs is based on program logics following the Hoare tradition [9, 30, 55, 56]. In this sense, separation logic [38, 42] is the best known and extensively used general framework for describing dynamically allocated mutable data structures in the heap. The success of these logics to deal with sequential heap manipulating programs has influenced much research [11, 28, 37, 52] to extend these logics for concurrent programs. However, handling unbounded unstructured concurrency is still very challenging. Also, virtually all these approaches are restricted to safety properties. In spite of some recent partial success [18, 25], extending separation logic techniques to build a general framework for liveness properties is still an open problem.

The main advantage of separation logic is its ability to describe concisely compositional proofs, thanks to the frame rule enabled by the hiding of the heap regions implicit in the separation conjunction operator. We advocate the use of explicit heap regions [2] represented as finite sets of object references. Explicit region manipulation provides the user full control over the heap partitioning and allows the use of classical first-order assertion languages to reason about heaps, including mutation and disjointness of memory regions. Unlike separation logic, the theory of sets [54] can be easily combined with other classical theories to build more powerful decision procedures, and is also amenable to integration into SMT solvers [3].

The problem of verifying parametrized systems is in general undecidable [1], even for finite-state components [51]. Some algorithmic methods proposed are restricted to classes of finite-state processes [15, 16, 20] and finite-state shared data to regain decidability. For instance, methods for automatically verifying infinite-state programs over unboundedly many threads such as [22] focus only on numerical and Boolean programs and tackle only safety properties. Model checking techniques [14, 49, 53, 57] are not readily applicable to systems composed of an unbounded number of threads, and cannot even scale to concrete systems with large number of threads due to the state space explosion problem. Methods such as [40] present an incomplete solution to the problem of uniform verification of finite-state parametrized systems, in particular for liveness, using symbolic model checking. Baukus et al. [4, 5] proposed and applied [6] a method which enabled the verification of parameterized networks of finite-state processes by modeling an infinite family of networks by a single transition system expressed in WS1S. However, their approach is limited to programs that manipulate simple data types while we advocate the use of specialized decision procedures, which allows the verification of programs that manipulate non-trivial data structures. An intrinsic limitation of algorithmic methods arises from the approach of handling data and temporal reasoning together.

Instead of considering algorithmic methods, we follow an alternative approach by extending temporal deductive methods such as Manna and Pnueli's [33] 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 to handle separately in a proof the temporal part and the underlying data being manipulated. Temporal deductive methods, like ours, are very powerful for reasoning 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. In [43–45] we describe some decision procedures for heap data structures that can be used in combination with PVDs.

In [48] we studied the problem of verifying parametrized safety properties in systems composed by an unbounded number of threads proposing specialized proof rules. The proof rules presented in [48] are the safety counterpart of the work presented in this paper. For safety, starting from the program and a safety specification, we generate a bounded number of verification conditions that can then be automatically checked using specialized decision procedures [43–45]. For tackling the problem of verifying parametrized temporal properties we have proposed in this paper the new formalism of PVDs. Leap [46] includes an implementation of (1) proof rules for parametrized safety properties (see [48]) (2) the PVDs presented in this paper, and (3) several decision procedures to automatically prove VCs (e.g., [43–45]). Leap is inspired by the STeP theorem prover [8] developed by Zohar Manna's group at Stanford in the 1990s.

For constructing PVDs, we extend generalized verification diagrams [12, 50] with capabilities to deal with concurrent parametrized systems. The work that is closest to ours, considering the verification of parametrized properties, is [34] and Section 8 of [8] (both of them from Zohar Manna's group), in which the authors use diagrams to verify temporal properties of parametrized reactive systems. However, their methods require 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. The PVD formalism we present in this paper generates quantifier-free verification conditions for parametrized systems that can be handled automatically by SMT solvers.

Another relevant work is the Deductive Verification Framework (DVF) [24], 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, DVF is based on Hoare-style reasoning, does not tackle parametrized verification and is restricted to safety properties.

Environment abstraction [17] and thread quantification abstractions [7] are abstraction-based techniques that deal with parametrized systems by abstracting processes and data altogether, making them 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 state assertion language. This is also a key difference between our approach and parametrized model checking for symmetric systems [21]. Similarly, [26] presents a verification approach which uses abstract transition systems to simulate lock-free algorithms. However, their method is limited to safety, and the use of simulation obscures the verification. Our framework, on the other hand, is capable of dealing with temporal properties of lock-free algorithms.

Apart from generalized verification diagrams, the other general technique for the verification of temporal properties is transition invariants [41], which characterize the validity of 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. Even though transition invariants are more suitable for automation, they are more difficult to understand and craft by human engineers, and hence less suitable as a semi-automatic verification method. Moreover, the transition invariant approach is not immediately applicable to parametrized systems. Even though we plan to attack the extension of transition invariants to parametrized systems in future work, this is out of the scope of this paper. Other techniques for checking program termination include [19] and [23], but all these techniques are restricted to non-parametrized systems.

7 Conclusion

In this paper we presented parametrized verification diagrams, an extension of generalized verification diagrams which is specifically designed to tackle the verification of temporal properties of concurrent systems executed by an unbounded number of threads. An advantage of PVDs is that they can encode in a single object the formal proof that all instances of the parametrized system satisfy a given temporal specification. The proof that the PVD encodes can be automatically checked solving a finite-state model checking problem, and proving a finite number of verification conditions, which are generated automatically from the program and the PVD. The generated verification conditions are guaranteed to be quantifier-free and can be automatically verified if there exists an appropriate decision procedure which can deal with the underlying theories of the data types manipulated by the program.

We have implemented PVDs as part of Leap, a theorem prover for parametrized systems under development at the IMDEA Software Institute. Using Leap we successfully verified liveness properties of mutual exclusion protocols and concurrent data types.

Future work includes studying the completeness of the PVD technique and relaxations of the symmetry requirement to expand the class of systems for which PVDs can be used.

Finally, the quest for generalized verification diagrams in the 1990s was pursuing two objectives: to find a unifying deductive framework for all temporal properties, and to serve as a humanly readable graphical formalism. In this paper we have focused on the effective applicability of PVDs in the verification of concurrent data types. Future work also includes studying and improving the applicability of PVDs as a graphical formalism amenable for human use.

Acknowledgment

We would like to acknowledge the anonymous reviewers for their feedback and numerous suggestions about how to improve this paper.

References

1. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Inf. Proc. Letters 22(6), 307–309 (1986)
2. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Proc. of ECOOP'08. pp. 387–411. Springer (2008)
3. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Handbook of Satisfiability, chap. Satisfiability Modulo Theories. IOS Press (2008)
4. Baukus, K., Bensalem, S., Lakhnech, Y., Stahl, K.: Abstracting WS1S systems to verify parameterized networks. In: Proc. of TACAS'00. LNCS, vol. 1785, pp. 188–203. Springer (2000)
5. Baukus, K., Lakhnech, Y., Stahl, K.: Verifying universal properties of parameterized networks. In: Proc. of FTRTFT'00. LNCS, vol. 1926, pp. 291–303. Springer (2000)
6. Baukus, K., Lakhnech, Y., Stahl, K.: Parameterized verification of a cache coherence protocol: Safety and liveness. In: Proc. of VMCAI'02. LNCS, vol. 2294, pp. 317–330. Springer (2002)
7. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, S.: Thread quantification for concurrent shape analysis. In: Proc. of CAV'08. LNCS, vol. 5123, pp. 399–413. Springer (2008)
8. Bjørner, N., Browne, A., Colón, M., Finkbeiner, B., Manna, Z., Sipma, H., Uribe, T.E.: Verifying temporal properties of reactive systems: A STeP tutorial. Form. Meth. in Sys. Design 16(3), 227–270 (2000)
9. Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: A logic-based framework for reasoning about composite data structures. In: Proc. of CONCUR'09. pp. 178–195. Springer (2009)
10. Bradley, A.R., Manna, Z., Sipma., H.B.: What's decidable about arrays? In: Proc. of VMCAI'06. LNCS, vol. 3855, pp. 427–442. Springer (2006)
11. Brookes, S.D.: A semantics for concurrent separation logic. In: Proc. of CONCUR'04. LNCS, vol. 3170, pp. 16–34. Springer (2004)
12. Browne, A., Manna, Z., Sipma, H.B.: Generalized temporal verification diagrams. In: Proc. of FSTTCS'95. LNCS, vol. 1206, pp. 484–498. Springer (1995)
13. Bultan, T., Gerber, R., Pugh, W.: Symbolic model checking of infinite state systems using Presburger Arithmetic. In: Proc. of CAV'97. LNCS, vol. 1254, pp. 400–411. Springer (1997)
14. Cerný, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: Proc. of CAV'10. LNCS, vol. 6174, pp. 465–479. Springer (2010)
15. Clarke, E.M., Grumberg, O.: Avoiding the state explosion problem in temporal logic model checking. In: Proc. of PODC'87. pp. 294–303. ACM (1987)
16. Clarke, E.M., Grumberg, O., Browne, M.C.: Reasoning about networks with many identical finite-state processes. In: Proc. of PODC'86. pp. 240–248. ACM (1986)
17. Clarke, E.M., Talupur, M., Veith, H.: Proving Ptolemy right: The environment abstraction framework for model checking concurrent systems. In: Proc. of TACAS'08. LNCS, vol. 4963, pp. 33–47. Springer (2008)
18. Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., Vardi, M.Y.: Proving that programs eventually do something good. In: Proc. of POPL'07. pp. 265–276. ACM (2007)
19. Dershowitz, N., Lindenstrauss, N., Sagiv, Y., Serebrenik, A.: A general framework for automatic termination analysis of logic programs. Applicable Algebra in Engineering, Communication and Computing 12(1/2), 117–156 (2001)
20. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: Proc. of CADE'00. LNAI, vol. 1831, pp. 236–254. Springer (2000)
21. Emerson, E.A., Kahlon, V.: Model checking large-scale and parameterized resource allocation systems. In: TACAS. LNCS, vol. 2280, pp. 251–265. Springer (2002)
22. Farzan, A., Kincaid, Z.: Verification of parameterized concurrent programs by modular reasoning about data and control. In: Proc. of POPL'12. pp. 297–308. ACM (2012)
23. Giesl, J., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Proving termination of programs automatically with AProVE. In: Proc. of IJCAR'14. LNCS, vol. 8562, pp. 184–191. Springer (2014)
24. Goel, A., Krstic, S., Leslie, R., Tuttle, M.R.: SMT-based system verification with DVF. In: Proc. of SMT'12. EPiC Series, vol. 20, pp. 32–43. EasyChair (2012)
25. Gotsman, A., Cook, B., Parkinson, M.J., Vafeiadis, V.: Proving that non-blocking algorithms don't block. In: Shao, Z., Pierce, B.C. (eds.) Proc. of POPL'09. pp. 16–28. ACM (2009)

26. Groves, L.: Verifying Michael and Scott's lock-free queue algorithm using trace reduction. In: CATS. CRPIT, vol. 77, pp. 133–142. Australian Computer Society (2008)
27. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan-Kaufmann (2008)
28. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: Proc. of ESOP'08. LNCS, vol. 4960, pp. 353–367. Springer (2008)
29. Keller, R.M.: Formal verification of parallel programs. Commun. ACM 19(7), 371–384 (1976)
30. Lahiri, S.K., Qadeer, S.: Back to the future: Revisiting precise program verification using smt solvers. In: Proc. of POPL'08. pp. 171–182. ACM (2008)
31. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM 17(8), 453–455 (1974)
32. Manna, Z., Browne, A., Sipma, H., Uribe, T.E.: Visual abstractions for temporal verification. In: Proc. of AMAST'98. LNCS, vol. 1548, pp. 28–41. Springer (1998)
33. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer (1995)
34. Manna, Z., Sipma, H.: Verification of parameterized systems by dynamic induction on diagrams. In: Proc. of CAV'99. LNCS, vol. 1633. Springer (1999)
35. Marco Bozzano, G.D.: Beyond parameterized verification. In: Proc. of TACAS'02. LNCS, vol. 2280, pp. 221–235. Springer (2002)
36. Miyano, S., Hayashi, T.: Alternating finite automata on ω-words. Theor. Comput. Sci. 32, 321–330 (1984)
37. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: Proc. of CONCUR'04. LNCS, vol. 3170, pp. 49–67. Springer (2004)
38. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Proc. of CSL'01. LNCS, vol. 2142, pp. 1–19. Springer (2001)
39. Pnueli, A.: The temporal logic of programs. In: Proc. of FOCS'77. pp. 46–57. IEEE Computer Society Press (1977)
40. Pnueli, A., Shahar, E.: Liveness and acceleration in parameterized verification. In: Proc. of CAV'00. vol. 1855, pp. 328–343. Springer (2000)
41. Podelsky, A., Rybalchenko, A.: Transition invariants. In: Proc. of LICS'04. pp. 32–41. IEEE Computer Society Press (2004)
42. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. of LICS'02. pp. 55–74. IEEE Computer Society Press (2002)
43. Sánchez, A., Sánchez, C.: Decision procedures for the temporal verification of concurrent lists. In: Proc. of ICFEM'10. LNCS, vol. 6447, pp. 74–89. Springer (2010)
44. Sánchez, A., Sánchez, C.: A theory of skiplists with applications to the verification of concurrent datatypes. In: Proc. of NFM'11. LNCS, vol. 6617, pp. 343–358. Springer (2011)
45. Sánchez, A., Sánchez, C.: Formal verification of skiplists with arbitrary many levels. In: Proc. of ATVA'14. vol. 8837, pp. 314–329. Springer (2014)
46. Sánchez, A., Sánchez, C.: LEAP: A tool for the parametrized verification of concurrent datatypes. In: Proc. of CAV'14. vol. 8559, pp. 620–627. Springer (2014)
47. Sánchez, A., Sánchez, C.: Parametrized verification diagrams. In: Proc. of TIME'14. pp. 132–141. IEEE Computer Society (2014)
48. Sánchez, A., Sánchez, C.: Parametrized invariance for infinite state processes. Acta Inf. 52(6), 525–557 (2015)
49. Sethi, D., Talupur, M., Schwartz-Narbonne, D., Malik, S.: Parameterized model checking of fine grained concurrency. In: Proc. of SPIN'12. pp. 208–226. Springer (2012)
50. Sipma, H.B.: Diagram-Based Verification of Discrete, Real-Time and Hybrid Systems. Ph.D. thesis, Stanford University (1999)
51. Suzuki, I.: Proving properties of a ring of finite-state machines. Inf. Proc. Letters 28, 213–214 (1988)
52. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: Proc. of PPOPP'06. pp. 129–136. ACM (2006)
53. Vechev, M.T., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Proc. of SPIN'09. LNCS, vol. 5578, pp. 261–278. Springer (2009)
54. Wies, T., Piskac, R., Kuncak, V.: Combining theories with shared set operations. In: Proc. of FROCOS'09. LNCS, vol. 5749, pp. 366–382. Springer (2009)
55. Yorsh, G., Rabinovich, A.M., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. In: Proc. of FOSSACS'06. pp. 94–110 (2006)
56. Yorsh, G., Rabinovich, A.M., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. J. Log. Algebr. Program. 73(1-2), 111–142 (2007)
57. Zhang, S.J.: Scalable automatic linearizability checking. In: Proc. of ICSE'11. vol. 5578, pp. 1185–1187. ACM (2011)