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

Alejandro Sánchez · César Sánchez

Parametrized Invariance for Infinite State Processes

the date of receipt and acceptance should be inserted later

Abstract We study the uniform verification problem for infinite state processes. This problem consists of proving that the parallel composition of an arbitrary number of processes running the same program (or a finite collection of programs) satisfies a temporal property. Our practical motivation is to build a general framework for the temporal verification of concurrent datatypes.

In this paper we propose a general method for the verification of safety properties of parametrized programs that manipulate complex local and global data, including mutable state in the heap. Our method is based on a clear division between the following two dimensions of the problem: the interaction between executing threads—handled by novel parametrized invariance proof rules—, and the data being manipulated—handled by specialized decision procedures. Our proof rules discharge automatically a finite collection of verification conditions. The size of this collection depends only on the size of the program and the specification, but not on the number of processes in any given instance or on the kind of data manipulated. Moreover, all verification conditions are quantifier free, which eases the development of decision procedures for complex data-types on top of off-the-shelf SMT solvers.

We prove soundness of our proof rules and illustrate their application in the formal verification of (1) two infinite-state mutual exclusion protocols; (2) shape and functional correctness properties of several concurrent datatypes, including fine-grained and non-blocking concurrent lists and queues.

Alejandro Sánchez
IMDEA Software Institute, Madrid, Spain
and Facultad de Informática, UPM, Spain
E-mail: alejandro.sanchez@imdea.org

César Sánchez
IMDEA Software Institute, Madrid, Spain
and Institute for Information Security, CSIC, Spain
E-mail: cesar.sanchez@imdea.org

We report empirical results using a prototype implementation of the proof rules and decision procedures.

1 Introduction

In this paper we present a general method to verify concurrent programs. The concurrent programs that we consider are executed by an arbitrary number of parallel threads which manipulate complex data, including unbounded local and share state. Our solution consists of a method that cleanly *separates* two concerns: (1) the data and its changes, handled by specialized decision procedures; and (2) the concurrent thread interactions, which is handled by novel proof rules called *parametrized invariance* presented in this paper. The method of parametrized invariance solves the *uniform verification* problem for safety properties:

> Given a parametrized system $\mathcal{S}[N] : P(1) \parallel P(2) \parallel \ldots \parallel P(N)$ and a safety property φ, establish whether $\mathcal{S}[M] \vDash \varphi$ for all instances $M \geq 1$.

in particular, for systems processes that manipulate arbitrary *infinite data*.

Our method is a generalization of the inductive invariance proof rules from temporal deductive verification [30], in which each verification condition corresponds to a small-step (a single transition) in the execution of a system. The applicability of these proof rules (without adding quantifiers) is restricted to non-parametrized systems. Non-parametrized systems can be described by a finite number of transitions, so one can generate one verification condition per transition. However, in parametrized systems, the number of transitions depends on the concrete number of processes in each particular instantiation, which is unbounded.

The main contribution of this paper is the principle or parametrized invariance, presented as proof rules that capture the effect of single steps of both:

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

Our parametrized invariance rules automatically discharge a *finite* collection of verification conditions. The validity of these verification conditions imply the correctness of all concrete system instantiations. In the rest of the paper we will use VC as a short for verification condition. We show that all VCs generated are *quantifier-free* as long as transition relations and specifications are quantifier-free, which is the case in conventional system descriptions, for example programs. For simplicity we present the rules for fully symmetric systems in which thread identifiers are only compared with equality, which encompasses many real systems. Other topologies, like rings of processes or totally ordered collections of processes can be handled with variations of our proof rules. To prove the generated verification conditions we use specific decision procedures for each manipulated data-type. For many data-types one can use directly SMT solvers [21, 33], or specialized decision procedures built on top of these solvers. We illustrate in this paper the use of a decision procedure for a quantifier-free theory of single linked list layouts with locks [37] to verify several fine-grained and non-blocking concurrent datatypes. Other powerful logics and tools for building similar decision procedures include [27,29].

1.1 Related Work

The problem of uniform verification of parametrized systems has received a lot of attention in recent years. This problem is, in general, undecidable [5], even for finite state components [41]. There are two general ways to overcome this limitation: deductive proof methods as the one we propose here, and (necessarily incomplete) algorithmic approaches.

Most algorithmic methods are restricted to finite state processes [11,12,16] to regain decidability. Examples are synchronous systems with guards [22]; interleaving systems with pairwise rendezvous [18]; systems with only conjunctive guards or only disjunctive guards [16]; implicit induction [17]; network invariants [28]; etc. A related technique, used in parametrized model checking, is symmetry reduction [13,19]. There also exists some automatic approaches designed to verify automatically specific properties such as linearizability [46]. Our approach is not automatic, but can be used to prove many other properties other than linearizability. Other approaches are based on shape analysis, but in general they are limited to a fixed number of threads [4] or limited to fixed data structures or shapes, like simple linked-list data structures [45]. Our approach can be applied to any theory of data with an available decision procedure. These aforementioned automatic approaches based on shape analysis can alleviate the human intervention needed in our approach by generating intermediate invariants.

A different tradition of automatic (incomplete) approaches is based on abstracting control and data altogether, for example representing configurations as words from a regular language [1,3,26,31]. Property directed techniques can be used to automatically prove invariants without manual effort [25], but they are in general restricted to Boolean programs. Other approaches use abstraction, like thread quantification [7] and environment abstraction [14] are based on similar principles as the full symmetry presented here, according to which all threads identifiers are interchangeable. However, these approaches rely on building specific abstract domains that abstract symbolic states instead of using decision procedures based on SMT solvers, as in our work. A very powerful method is invisible invariants [6,35,47], which works by heuristically generating invariants on small instantiations and trying to generalize these to parametrized invariants. However, this method is so far also restricted to finite state processes.

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

Our target application is the verification of concurrent datatypes [24], where the main difficulty arises from the mix of *unstructured unbounded* concurrency and heap manipulation. We use the term unstructured concurrency to refer to programs that are not structured in sections protected by locks but that allow a more liberal pattern of shared memory accesses, including fine-grain locking or lock-free algorithms. Unbounded refers to the lack of bound on the number of running threads. Concurrent datatypes can be modeled naturally as fully symmetric parametrized systems, where each thread executes in parallel a client of the data-type. There exist results [2] proving decidability for systems with finite control flow and infinite domain provided the infinite domain contains a well-founded preorder. On the contrary, our approach can be applied to any finite or infinite data domain for which there is a decision procedure.

In this paper we focus only in verification of safety properties using parametrized invariance proof rules. The main focus of our earlier work [37] was on decision procedure for concurrent lists. In [37] we preliminarily sketched a method for the verification of liveness properties of parametrized system later formally developed in [38], currently under review. These methods build on the proof rules presented here. Temporal deductive methods [30], like ours, are very powerful to reason about (structured or unstructured) concurrency, but they have been traditionally restricted to non-parametrized systems and scalar data.

The rest of the paper is structured as follows. Section 2 includes the preliminaries. Section 3 introduces the parametrized invariance rules. Section 4 contains the examples, a description of our tool and empirical evaluation results. Finally, Section 5 concludes.

2 Preliminaries and Running Examples

2.1 Running Examples

Throughout the paper we illustrate the concepts and application of parametrized invariance using the running example programs shown in Fig. 1. We use a simple programming language similar to SPL [30].

Fig. 1(a) contains SetMutex, a parametrized mutual exclusion protocol based on tickets. Each thread that intends to access the critical section at line 5, acquires a ticket with a unique and increasing number and—atomically—announces its intention to enter the critical section by adding the ticket to a shared global set of tickets (line 3). Then, the thread waits (line 4) until its ticket becomes the lowest value in the set before entering the critical section. After a thread leaves the critical section it removes its ticket from the global set (line 6). SetMutex uses two global variables: *avail*, of type *Int*, which stores the shared counter; and *bag*, of type $Set\langle Int\rangle$, which stores the set of tickets owned by those threads that are trying to access the critical section. Any concrete instance of the parametrized system, obtained by fixing

```
global
  $Int\ avail := 0$
  $Set\langle Int\rangle\ bag := \emptyset$

procedure SetMutex
  $Int\ ticket := 0$
begin
loop
    noncritical
    $\left\langle \begin{array}{l} ticket := avail++ \\ bag.add(ticket) \end{array} \right\rangle$
    await $(bag.min == ticket)$
    critical
    $bag.remove(ticket)$
end loop
end procedure
```

(a) SetMutex, using a set of integers

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

procedure IntMutex
  $Int\ ticket$
begin
loop
    noncritical
    $ticket := avail++$
    await $(min == ticket)$
    critical
    $min := min + 1$
end loop
end procedure
```

(b) IntMutex, using two counters

Fig. 1: Two implementations of a ticket based mutual exclusion protocol

the number of running threads, is an infinite state system because the available ticket is ever increasing. Program IntMutex in Fig. 1(b) implements a similar version of the protocol in which only the minimum value among all given tickets is maintained, in a global variable of type *Int*. This program is also infinite state for any concrete instantiation since tickets are also ever increasing. We include these two very similar programs to illustrate that our method is not wired for a specific theory of data, but it allows to be applied to every program as long as there is a procedure to reason about the data.

2.2 Preliminaries

The formal verification problem for non-parametrized systems takes a system described as a program and a specification of a safety property expressed as a state predicate. A system satisfies its specification if all states reachable in all the traces of the transition system that models the set of executions of the program satisfy the specified property.

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

- *Signature.* The signature Σ_{prog} is a first-order signature modeling the data manipulated in a given 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 in Σ_{prog}.
- *Program Variables.* V is a finite set of (typed) variables, whose types are taken from sorts in Σ_{prog}.
- *Initial Condition.* Θ is the initial condition, expressed as a first-ordered assertion over the variables V. Values of V satisfying Θ correspond to initial states of the system.

To show that P satisfies $\Box\varphi$:

$$\begin{array}{lcl}
\mathsf{B1.} & \Theta \to \varphi & \\
\mathsf{B2.} & \varphi \wedge \tau \to \varphi' & \text{for all } \tau \\
\hline
 & \Box\varphi &
\end{array}$$

(a) The basic invariance rule B-INV

To show that P satisfies $\Box\varphi$, find q:

$$\begin{array}{lcl}
\mathsf{I1.} & \Theta \to q & \\
\mathsf{I2.} & q \wedge \tau \to q' & \text{for all } \tau \\
\mathsf{I3.} & q \to \varphi & \\
\hline
 & \Box\varphi &
\end{array}$$

(b) The invariance rule INV

Fig. 2: Rules B-INV and INV for non-parametrized systems.

- *Transition Relation.* $\mathcal{T}$ is a finite set of transitions. Each transition τ in $\mathcal{T}$ is expressed as a first-order formula $\tau(V, V')$ that can refer to program variables from 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. We assume that every system is equipped with an idle transition whose transition relation is $\tau_\epsilon(V, V')$ describes the preservation of all system variables ($v = v'$ for all $v \in V$). This idle transition allows to reason only about infinite runs even for deadlocked systems.

A *state* is an interpretation of V that 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 (that assigns values to variables in V according to s and to variables in V' according to s') satisfies the formula $\tau(V, V')$. In this case, we write $\tau(s, s')$, and we say that the system reaches state s' from state s by taking transition τ. We say that a transition τ is *enabled* in state s if there is a state s' for which $\tau(s, s')$. The *enabling condition* of transition τ is then the formula $\exists V'.\tau(V, V')$.

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

- the first state is initial: $s_0 \models \Theta$; and
- all steps are legal. For all i, the following relation holds: $\tau_i(s_i, s_{i+1})$. We say that τ_i is taken at s_i, leading to state s_{i+1}.

A system $\mathcal{S}$ satisfies a safety property p, which we write $\mathcal{S} \models \Box p$, whenever all runs of $\mathcal{S}$ satisfy p at all states. For non-parametrized systems, invariants can be proven using the classical invariance rules [30], shown in Fig. 2. The formula φ' in the consequent of premise B2 refers to the formula obtained from φ by replacing every variable v in φ by v'. The formula q' in the consequent of I2 is obtained from q similarly. The basic rule B-INV establishes that if the candidate invariant φ holds initially and is preserved by every transition, then φ is indeed an invariant. In this case we call φ an *inductive invariant.* Rule INV uses an intermediate strengthening invariant q. If q implies φ and q is an invariant, then φ is also an invariant. An alternative characterization of rule INV requires finding q and proving that $(q \wedge \varphi)$ is an inductive invariant using rule B-INV.

For non-parametrized systems, premises B1 and I1—called *initiation*—discharge one verification condition, and premises B2 and I2—called *consecution*—discharge a collection verification conditions whose size is linear in the number of transitions. To use these invariance rules directly for parametrized systems, one either needs to use quantification (as in [35]) or apply the rules once for each concrete system instantiation, which requires to discharge and prove unbounded number of verification conditions. In this paper we present novel proof rules that allow to tackle parametrized system discharging only a finite number of VCs.

2.3 Parametrized Concurrent Programs

Programs manipulate data during their execution. In our approach, the reasoning about the program data and its manipulation is handled by specialized decision procedures for specific theories of data. We use Σ_{prog} for the first-order signature and T_{prog} for the theory. For example, for program SetMutex in Fig. 1, T_{prog} is the combined theory of Presburger arithmetic, finite sets of integers with minimum, and finite values (to reason about locations).

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

When building verification arguments it is sometimes convenient to enlarge the set of variables with auxiliary variables, called *ghost variables*, to store interesting information about the history of the computation. These variables are not allowed in the enabling condition of statements occurring in the actual program, and are only used to update other ghost variables using statements added to the program. We call these statements ghost code, which is executed atomically with the real code they annotate. In particular, ghost variables do not occur in the code in expressions or assignments to program variables.

Given a parametrized program P, we associate to P an *instance family* $\{\mathcal{S}_P[M]\}$, a collection of non-parametrized transition systems indexed by $M \geq 1$ the number of running threads. We use $[M]$ to denote the set $\{0, \ldots, M-1\}$ of concrete thread identifiers. For each M, the concrete non-parametrized transition system $\mathcal{S}_P[M] : \langle \Sigma_{\mathsf{prog}}, V, \Theta, \mathcal{T} \rangle$ consists of:

- *Signature.* The signature Σ_{prog} for reasoning about data, including program locations.

- *Program Variables.* The set V of variables is:

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

  Note that "$v[k]$" is an indivisible variable name. Alternative names could have been v_k or vk.
- *Initial Condition.* The initial condition Θ is described by two predicates Θ_g (that only refers to variables from V_{global}) and Θ_l (that can refer to variables in V_{global} and V_{local}). These expressions are extracted from the program using the semantics of the programming language. Given a thread identifier $a \in [M]$ for a concrete system $\mathcal{S}[M]$, $\Theta_l[a]$ is the initial condition for thread a, obtained by replacing in Θ_l every occurrence of a local variable v from V_{local} for $v[a]$. The initial condition of the concrete transition system $\mathcal{S}[M]$ 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 obtain from the semantics of the programming language. The formula $\tau_\ell[a]$ is obtained from τ_ℓ by replacing every occurrence of a local variable v for $v[a]$, and v' for $v[a]'$. Note again that that "$v[a]'$" is an indivisible variable name, denoting the primed version of $v[a]$.

Example 1 Consider program SetMutex in Fig. 1(b). The instance consisting of two running threads, SetMutex[2], contains the following variables:

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

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

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

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

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

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

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

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

3 Parametrized Transition Systems, Formulas and Proof Rules

We show in this section how to specify and prove invariant properties of parametrized systems. Unlike in [35] we generate quantifier-free verification conditions, enabling the development of decision procedures for complex datatypes.

3.1 Parametrized Transition Systems

We introduce here the notion of parametrized transition system. A parametrized transition system associated with a 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. The intention of parametrized transition systems is not to define program runs directly but to serve as a modeling language for the definition of parametrized formulas and to enable the definition of proof rules for parametrized systems. We describe each component separately:

- *Parametrized Program Signature.* To capture thread identifiers in an arbitrary instantiation of the parametrized system we introduce a new sort tid interpreted as an unbounded but finite set. The signature Σ_{tid} contains only $=$ and $\neq$, and no constructor. Then, we extend the theory T_{prog}— used to reason about the data in the program— with the theory of arrays T_{A} from [9], taking indices from tid and elements ranging over sorts t of the local variables of program P. We use T_{param} for the union of theories T_{prog}, T_{tid} and T_{A}, and Σ_{param} for the combined signature.
- *Parametrized Program Variables.* For each local variable v of type t in the program, we introduce a variable name a_v of sort $\mathsf{array}\langle t \rangle$, including a_{pc} for the program counter pc. Using the theory of arrays, the expression $a_v(k)$ denotes the element of sort t stored in array a_v at position given by expression k of sort tid. The expression $a_v\{k \leftarrow e\}$ corresponds to an array update, and denotes the array that results from a_v by replacing the element at position k with e. For clarity, we abuse notation using $v(k)$ for $a_v(k)$, and $v\{k \leftarrow e\}$ for $a_v\{k \leftarrow e\}$. Note that $v[0]$ is different from $v(k)$: the term $v[0]$ is an atomic term in V (for a concrete system $\mathcal{S}_P[M]$) referring to the local program variable v of a concrete thread with id 0. On the other hand, $v(k)$ is a non-atomic term built using the signature of arrays, where k is a variable (logical variable, not program variable) of sort tid serving as index of array v. The use we make of T_{A} is very limited: we do not use arithmetic over indices or nested arrays, so the conditions for decidability in [9] are trivially met. Variables of sort tid indexing arrays play a special role, so we classify formulas depending on the number of free variables of sort tid. The parametrized set of program variables with index variables X of sort tid is:

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

  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 $Var(\varphi)$ to refer to the set of variables of sort tid that occur free in φ. Since we restrict to the quantifier-free fragment of $F_{\mathsf{param}}(X)$ then $Var(\varphi)$ 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 $Var(\varphi)$ 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.

- *Parametrized Initial Condition.* We similarly define the parametrized initial condition for a given set X of index variables (of sort tid) as:

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

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

Example 2 Consider program SetMutex. The parametrized transition $\tau_4^{(k)}$, for thread k in line 4, is the following formula from $F_{\mathsf{param}}(\{k\})$:

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

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

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

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

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

□

3.2 Parametrized Formulas

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

Let us fix a program P, and let $\{\mathcal{S}_P[M]\}$ be its instance family and $\mathcal{P}_P$ be the parametrized transition system.

Definition 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 $[M]$:

$$\alpha : Var(\varphi) \rightarrow [M]$$

We use Arr_M^φ for the set of concretizations of φ and M.

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

$$\alpha(F(t_1,\ldots,t_n)) \mapsto F(\alpha(t_1),\ldots,\alpha(t_n))$$
$$\alpha(P(t_1,\ldots,t_m)) \mapsto P(\alpha(t_1),\ldots,\alpha(t_m))$$

For symbols in the theory of arrays, we first translate all literals of sort array in a formula φ to (a) either variables of sort array, or (b) array updates in the right of equalities $w = v\{k \leftarrow e\}$. This translation can be easily achieved by introducing a new array variable v for a more complex term t occurring in φ, conjoining $v = t$ to the root of φ and substituting in φ all occurrences of t for v. Then, α can be defined for the remaining array cases:

$$\begin{array}{rcl}
\alpha(v(k_i)) & \mapsto & v[\alpha(k_i)] \\
\alpha(w = v\{k \leftarrow e\}) & \mapsto & \big(w[\alpha(k)] = e \wedge \bigwedge\limits_{a \in M \setminus \alpha(k)} w[a] = v[a]\big) \\
\alpha(w = v) & \mapsto & \bigwedge\limits_{a \in M} w[a] = v[a]
\end{array}$$

Essentially, a concretization computes the predicate $\alpha(\varphi)$ for system $\mathcal{S}_P[M]$ that results from φ by instantiating its variables $Var(\varphi)$ according to the map α. For example, consider the formula $\Theta_{\mathsf{param}}(\{i,j\})$ in (2) above. The concretization of $\Theta_{\mathsf{param}}(\{i,j\})$ by the map $\alpha : \{i \leftarrow 0, j \leftarrow 1\}$ is the concrete initial condition expressed by (1) in Example 1.

We can now formulate the uniform verification problem in terms of concretizations.

Definition 2 (Uniform Verification Problem) Given a program P and parametrized formula $\varphi(\overline{k})$ we say that P satisfies the universal safety property $\big(\forall \overline{k}\, .\, \Box\varphi(\overline{k})\big)$ whenever for every M and substitution $\alpha : \overline{k} \to [M]$, the concrete closed system $\mathcal{S}_P[M]$ satisfies $\mathcal{S}_P[M] \vDash \Box\alpha(\varphi(\overline{k}))$. In this case we write $P \vDash \forall\overline{k}.\Box\varphi(\overline{k})$, or simply $P \vDash \Box\varphi$ and say that φ is a *parametrized invariant* of P.

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

We prove first an important intermediate result, that will ease later to prove soundness theorems.

Lemma 1 *Let ψ be a parametrized formula and α a concretization of ψ for a given number of threads M. Then,*

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

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

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

- The *domains* of all sorts in $\mathcal{A}$ and $\mathcal{B}$ coincide, except for arrays, in which case indices are $\mathbb{Z}$ and values are the corresponding domain in $\mathcal{A}$. In particular, the domain of the program counter variable is that of arrays indexed by $\mathbb{Z}$ with values ranging over locations.
- For *terms*: the only terms of sort tid occurring in ψ are variables. For these, $\mathcal{B}$ assigns the integer within range $[M]$ given by α:

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

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

  $$v^{\mathcal{B}}(n) = \begin{cases} (v[n])^{\mathcal{A}} & \text{if } n \in [M] \\ d_\sigma & \text{if } n \notin [M] \end{cases} \tag{3}$$

  Note, in particular, that for tid variable k,

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

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

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

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

  $$(w = v)^{\mathcal{B}} \quad \iff \quad (\alpha(w = v))^{\mathcal{A}} \tag{4}$$

  Since for all $n \notin [M]$, by (3), $w^{\mathcal{B}}(n) = d_\sigma = v^{\mathcal{B}}(n)$, it follows that:

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

  Therefore, (4) holds.

Second, we show that:

$$(w = v\{k \leftarrow e\})^{\mathcal{B}} \iff (\alpha(w = v\{k \leftarrow e\}))^{\mathcal{A}} \tag{5}$$

Given array variables v and w, tid variable k, and term e (of the appropriate sort of elements stored in arrays v and w) by (3) all indices not in $[M]$ are mapped to the same value d_σ. Then, $w^{\mathcal{B}}(n) = v^{\mathcal{B}}(n)$ for all $n \notin [M]$. It follows that:

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

Therefore, (5) holds as well.
Finally, for all common predicates P, that is, for all predicates except those in the theory of arrays we let:

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

Hence, by (3), (4) and (5) it follows that for all predicates, including those in the theory of arrays:

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

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

3.3 Parametrized Proof Rules

We introduce now specialized proof rules for parametrized systems, which allow to prove parametrized invariants discharging only a finite number of verification conditions. The simplest proof rule is BP-INV, the *basic parametrized invariance rule*, which appears in Fig. 3.

Premise P1 guarantees that the initial condition holds for all instantiations. Premise P2 guarantees that φ is preserved under transitions taken by all threads referred in the formula and considering all possible transitions of the system. Finally, P3 guarantees that φ is preserved for all transitions taken by *any other thread* (this is achieved by taking a *fresh* thread identifier in P3. A *fresh* variable of type thread identifier refers to a variable not appearing in φ. Premise P1 discharges only one verification condition, P2 discharges one VC per transition in the description of the system (statement in the program) and per index variable in the formula φ. Finally, P3 generates one extra VC per transition in the system. All these VCs are quantifier-free provided that φ is quantifier-free. Rules P2 and P3 must be verified for all

To show that P satisfies $\square\varphi$ (where $\overline{k} = \mathit{Var}(\varphi)$):

$$
\begin{array}{lrcl}
\mathsf{P1.} & \Theta_{\mathsf{param}}(\overline{k}) & \rightarrow \varphi & \\
\mathsf{P2.} & \varphi \wedge \tau^{(i)} & \rightarrow \varphi' & \text{forall } \tau \text{ and all } i \in \overline{k} \\
\mathsf{P3.} & \varphi \wedge \big(\bigwedge\limits_{x \in \mathit{Var}(\varphi)} j \neq x \wedge \tau^{(j)} \big) & \rightarrow \varphi' & \text{forall } \tau \text{ and one fresh } j \notin \overline{k} \\
\hline
 & & \square\varphi &
\end{array}
$$

Fig. 3: The basic parametrized invariance rule BP-INV

possible system transitions. Moreover, rule P2 requires each transition to be checked for every thread identifier appearing in the formula φ. Corollary 1 below justifies the soundness of rule BP-INV.

There are cases in which premise P3 cannot be proven, even if φ is initial and preserved by all transitions of all threads.

Example 3 Consider the following program POSITIVE and the 1-index property $\varphi_{\text{POS}} : (x > 0 \wedge c(i) > 0)$.

```
   global
     Int $x > 0$
   procedure POSITIVE
     Int $c > 0$
1:     $x = x + c$
2:
   end procedure
```

This property is trivially a parametrized invariant, but premise P3 is not valid, when fresh thread id j takes transition at line 1:

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

An example counter-model is:

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

Essentially, the formula φ_{POS} does not imply that $c(j) > 0$ before the transition $\tau_1^{(j)}$ is taken, and the counter-example assigns $c(j) = -1$. A transition for which the corresponding verification condition is not valid is known as an *offending transition* (see [30]), or more modernly as a *counter-example to induction* [8]. □

The problem exposed in Example 3 is that in the antecedent of premise P3, φ does not refer to the fresh arbitrary thread introduced. In other words, BP-INV tries to prove a property for the threads referred to in the formula, without assuming anything about any other thread. It is sound, however, to assume that in the pre-state the property one intends to prove holds *for all processes*, and not only for the processes explicitly mentioned in the formula. Intuitively speaking, the justification of this assumption is based on $\forall k. \square\varphi(k)$ being equivalent to $\square\forall k. \varphi(k)$. However, we want to avoid quantification in all verification conditions. Instead, we propose to instantiate the formula φ in the antecedent of premises, not only to threads in the formula itself, but also to other threads, in particular in the transition relation. The notion of *support* allows to formally capture this intuition. We use the conventional notion of *substitution* in first-order-logic (as a map from variables to terms), and restrict our attention to maps from a set of tid variables X into set of tid variables Y. Substitutions can be extended to maps from terms to terms and (formulas to formulas) homomorphicaly in the usual way, preserving all symbols except the replaced variables. A partial substitution is a partial map.

Definition 3 (Support) Let ψ, A and B be parametrized formulas, and let S be the set of partial substitutions from $\mathit{Var}(\psi)$ into $\mathit{Var}(A \rightarrow B)$. We say that ψ supports $(A \rightarrow B)$, whenever

$$\big(\big(\bigwedge_{\sigma \in S} \sigma(\psi)\big) \wedge A\big) \rightarrow B \qquad\qquad \text{is valid}$$

We use $\psi \rhd (A \rightarrow B)$ as a short notation for $\big((\bigwedge_{\sigma \in S} \sigma(\psi)) \wedge A\big) \rightarrow B$.

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

$$\big(\big(\bigwedge_{\sigma \in S'} \sigma(\psi)\big) \wedge A\big) \rightarrow B \qquad\qquad \text{is valid}$$

then

$$\big(\big(\bigwedge_{\sigma \in S} \sigma(\psi)\big) \wedge A\big) \rightarrow B \qquad\qquad \text{is also valid}$$

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

Example 4 Consider the program Positive in Example 3 above, and let A and B be the formulas:

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

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

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

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

The main motivation for introducing the notion of support is to instantiate a formula ψ (an assumed fact in the pre-state) to strengthen the antecedent of an implication (the VC) without extending the vocabulary (the free tid variables) in the resulting strengthened implication. We can strengthen premise $\mathsf{P3}$, so the target invariant candidate φ can be assumed in the pre-state *for every thread*, in particular for the fresh thread that takes the transition:

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

For example, let $\varphi(i)$ be a candidate invariant with one thread variable (an index 1 invariant candidate). Premise S3 is $\big(\varphi \rhd (j \neq i \wedge \tau^{(j)} \rightarrow \varphi'(i))\big)$, or equivalently

$$\big(\varphi(j) \wedge \varphi(i) \wedge j \neq i \wedge \tau^{(j)}\big) \rightarrow \varphi'(i).$$

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

Unfortunately, rule P-INV can still fail to prove invariants if they are not inductive.

Example 5 Consider the following property of program SETMUTEX:

$$\big(i \neq j \wedge \mathsf{active}(i) \wedge \mathsf{active}(j)\big) \rightarrow \mathit{ticket}(i) \neq \mathit{ticket}(j) \qquad (6)$$

where $\mathsf{active}(i)$ is a short notation for $(pc(i) = 4 \vee pc(i) = 5 \vee pc(i) = 6)$. This property is a 2-index parametrized invariant of SETMUTEX, but it cannot be proven by BP-INV. Premise P2 fails by taking $\tau_3^{(j)}$ (this transition is an offending transition for proving the property invariant) as witnessed by a model from a pre-state in which:

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

A true fact of the program that eliminates this spurious counter-example is that $\mathit{ticket}(i) < \mathit{avail}$ is invariant, but neither the goal invariant (6) nor the transition relation for $\tau_3^{(j)}$ directly imply this fact. □

Using support we can rewrite the basic parametrized invariance rules into the parametrized invariance rule P-INV in Fig. 4.

To show that P satisfies $\Box\varphi$ (where $\overline{k} = \mathit{Var}(\varphi)$):

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

Fig. 4: The parametrized invariance rule P-INV

Theorem 1 (Soundness of P-INV) *Let $\mathcal{S}$ be a parametrized system and $\Box\varphi$ a parametrized safety property. If* S1, S2 *and* S3 *hold, then $\mathcal{S} \vDash \Box\varphi$.*

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

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

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

We need to show that both premises of B-INV are valid.

- Premise $\mathsf{B1}$: Since $\mathsf{S1}$ is valid, then $\Theta_{\mathsf{param}}(\overline{k}) \rhd \varphi$ is valid, or equivalently $\Theta_{\mathsf{param}}(\overline{k}) \to \varphi$, where $\overline{k} = Var(\varphi)$. Consequently, by Lemma 1, $\alpha(\Theta_{\mathsf{param}}(\overline{k}) \to \varphi)$ is valid for an arbitrary α, and then, $\alpha(\Theta_{\mathsf{param}}(\overline{k})) \to \alpha(\varphi))$ is valid for an arbitrary α. Then,

$$\Theta_g \wedge \bigwedge_{n \in [M]} \Theta_l[n] \to \bigwedge_{\alpha \in Arr_M^\varphi} \alpha(\varphi) \qquad \text{is valid}$$

  and, finally,

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

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

$$\Psi \wedge \tau[n] \to \Psi' \qquad \text{is valid} \tag{7}$$

  Let α be an arbitrary concretization in Arr_M^φ. We will show that:

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

  which implies (7) because α is arbitrary. We consider two cases depending on whether the concrete n is in the image of α or not:

  1. $n \in Img\,\alpha$, i.e., there is a $i \in Var(\varphi)$ for which $\alpha(i) = n$. Then, since $\mathsf{S2}$ for $\tau^{(i)}$ is valid, by Lemma 1,

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

     or, equivalently, for the set S of partial substitutions

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

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

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

which implies that

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

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

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

Now, since premise S3 is valid, by Lemma 1 for α_3:

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

Then, for the set of substitutions S

$$\big(\alpha_3(\bigwedge_{\sigma \in S} \sigma(\varphi)) \wedge \alpha_3\big(\bigwedge_{x \in \mathit{Var}(\varphi)} j \neq x\big) \wedge \alpha_3(\tau^{(j)})\big) \rightarrow \alpha_3(\varphi') \quad \text{is valid}$$

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

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

and hence

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

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

The following corollary establishes the soundness of rule BP-INV, and follows immediately from Theorem 1 by observing that if $A \rhd B \rightarrow C$ is valid then $(A \wedge B) \rightarrow C$ is also valid.

Corollary 1 (Soundness of BP-INV) *Let $\mathcal{S}$ be a parametrized system and $\Box\varphi$ a parametrized safety property. If* P1, P2 *and* P3 *hold, then $\mathcal{S} \vDash \Box\varphi$.*

As for closed systems, there are two reasons that explain the failure to prove, using the inductive rules, that a candidate invariant is indeed invariant: (1) the candidate is actually not an invariant; (2) the candidate is invariant but not inductive, so one needs to use strengthening invariants, or to prove the candidate is inductive relative to other invariants. However, in parametrized systems it is not necessary the case that by simply conjoining the candidate and its strengthening one obtains a BP-INV inductive invariant, because one may need to instantiate the candidate formulas for all thread identifiers in their shared vocabulary. One solution is to prove the invariants incrementally, and use support to instantiate to freshly introduced thread identifiers. This idea is captured by rule SP-INV in Fig. 5.

Theorem 2 *Let $\mathcal{S}$ be a parametrized system and $\Box\varphi$ a parametrized safety property. If* R0*,* R1*,* R2 *and* R3 *hold, then $\mathcal{S} \vDash \Box\varphi$.*

Proof Assume that $\mathit{Var}(\varphi) \cap \mathit{Var}(\psi) = \emptyset$ which can be easily achieved by renaming tid variables. The proof is very similar to the proof of Theorem 1 showing that:

$$\Psi \overset{\text{def}}{=} \bigwedge_{\alpha_1 \in Arr_M^{\varphi}} \alpha_1(\varphi) \wedge \bigwedge_{\alpha_2 \in Arr_M^{\psi}} \alpha_2(\psi)$$

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

3.4 The Parametrized Graph Proof Rule

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

A naïve solution attempt would be to write down all necessary candidates in a single large formula and prove this formula invariant using P-INV. In parametrized systems, this approach quickly leads to formulas with many

To show that $\mathcal{S}$ satisfies $\Box\varphi$ (where $\overline{k} = \mathit{Var}(\varphi)$). Find ψ with:

$$\begin{array}{llrl}
\mathsf{R0.} & & \Box\psi & \\
\mathsf{R1.} & & \Theta_{\mathsf{param}}(\overline{k}) \rhd \varphi & \\
\mathsf{R2.} & \psi, \varphi \rhd & \tau^{(i)} \rightarrow \varphi' & \text{forall } \tau \text{ and all } i \in \overline{k} \\
\mathsf{R3.} & \psi, \varphi \rhd & \bigwedge_{x \in \mathit{Var}(\varphi)} j \neq x \wedge \tau^{(j)} \rightarrow \varphi' & \text{forall } \tau \text{ and one fresh } j \notin \overline{k} \\
\hline
 & & \Box\varphi &
\end{array}$$

Fig. 5: The general strengthening parametrized invariance rule SP-INV for proving relative inductive parametrized invariants.

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

$$\begin{array}{lrl}
\mathsf{G1.} & \Theta_{\mathsf{param}}(\overline{k}) \;\rhd\; \psi & \text{forall } \psi \in \mathit{Invs}, \text{ where } \overline{k} = \mathit{Var}(\psi) \\
\mathsf{G2.} & \Phi, \psi \rhd \qquad\qquad \tau^{(k)} \to \psi' & \text{forall } \psi \in \mathit{Invs}, \text{ forall } \tau, \\
 & & \quad \text{and all } j \in \mathit{Var}(\psi), \\
 & & \quad \text{and } \Phi = \{\psi_i \mid (\psi_i, \psi) \in \mathit{Supp}\} \\
\mathsf{G3.} & \Phi, \psi \rhd \displaystyle\bigwedge_{x \in v} k \neq x \wedge \tau^{(k)} \to \psi' & \text{forall } \psi \in \mathit{Invs}, \text{ forall } \tau, \\
 & & \quad \text{one fresh } k \notin v = \mathit{Var}(\psi), \\
 & & \quad \text{and } \Phi = \{\psi_i \mid (\psi_i, \psi) \in \mathit{Supp}\} \\
\hline
 & \Box\varphi &
\end{array}$$

Fig. 6: The graph parametrized invariance rule G-INV.

duplications due to thread renaming which in turn jeopardizes the scalability of the decision procedures for sophisticated data by requiring to prove large formulas, which requires to search for large models. A more efficient approach consists on building the proof modularly, splitting invariants into meaningful sub-formulas to be used only when required. This sort of proof modularity is captured by rule G-INV shown in Fig. 6. This rule handles cases in which invariants that mutually dependent on each other need to be verified.

A proof graph is a finite directed graph $(\mathit{Invs}, \mathit{Supp})$ whose nodes in *Invs* are labeled with candidate invariant formulas. An edge in *Supp* between two nodes indicates that in order to prove the formula pointed by the edge it is useful to use the formula at the origin of the edge as support. As a particular case, a formula with no incident edges is inductive and can be shown directly using P-INV. Note that a proof graph can be (and in practice it is) a cyclic graph. A proof graph encodes the proof that *all* the formulas labeling nodes are invariants of the system. The edges encode the information of which sub-formulas (the set of predecessor nodes) are needed to prove a particular node.

Theorem 3 (Soundness of Proof Graphs) *Let $\mathcal{P}_P$ be a parametrized system and $(\mathit{Invs}, \mathit{Supp})$ a proof graph. If* G1, G2, *and* G3 *hold, then $P \vDash \Box\psi$ for all $\psi \in \mathit{Invs}$.*

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

$$\Psi \overset{\text{def}}{=} \bigwedge_{\psi \in \mathit{Invs}} \; \bigwedge_{\alpha \in \mathit{Arr}^{\psi}_{M}} \alpha(\psi)$$

The argument to show that premise B1 follows from G1 is identical to the argument that B1 follows from S1, in the proof of Theorem 1 above.

For B2, we consider an arbitrary ψ in *Invs* and an arbitrary concretization α from Arr^{ψ}_{M}. We need to show the following:

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

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

1. $n \in \mathit{Img}\,\alpha$. Let k in $\mathit{Var}(\psi)$ be such that $\alpha(k) = n$. In this case, by premise $\mathsf{G2}$ with $\mathit{\Phi} = \{\psi_i | (\psi_i, \psi) \in E\}$,

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

   and hence

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

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

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

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

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

   or, for α_2 by Lemma 1,

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

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

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

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

4 Implementation and Empirical Evaluation

We illustrate the use of our parametrized invariance rules showing that the infinite state protocols SetMutex and IntMutex satisfy mutual exclusion. Additionally, we also verify list shape preservation and some functional properties of an implementation of concurrent lock-coupling lists, a course-grain unbounded total concurrent queue, a lock-free queues and a lock-free stack [24]. The proof rules presented in this paper are implemented in the temporal theorem prover tool Leap[1]. Leap parses a temporal specification and a program description in a C-like language, and automatically generates VCs applying the parametrized invariance rules. Each VC is then discharged and automatically verified using a suitable decision procedure for each theory.

We report here the use of three decision procedures built on top of the SMT solvers Z3 [33] and Yices [15]:

1. a simple decision procedure that can reason only about program locations, and treats all other predicates as uninterpreted;
2. a decision procedure based on $\mathsf{TLL3}$ (see [37]) capable of reasoning about single-linked lists layouts in the heap with locks to aid in the verification of fine grain locking algorithms;
3. a decision procedure that reasons about program locations, integers and finite sets of integers with minimum and maximum functions, for the mutual exclusion protocols.

All these decision procedures are implemented as part of Leap. Some of these decision procedures and their corresponding implementations are based on small model theorems. The satisfiability of a quantifier free formula is reduced to the search for a model of a sufficiently large size (see e.g., [37] for an example of the calculation of the upper-bound). The theories involved in each decision procedures can also be combined through Nelson-Oppen [34] provided the theories to be combined meet some requirements [20,42,43]. Leap also implements some heuristic optimizations, called *tactics*, like attempting first to use a simpler decision procedure, or instantiating support formulas lazily. These optimizations aid the solvers to speed the proof of validity of many VCs by reducing the formulas obtained by partial assignments in the application of rules sp-inv or g-inv.

4.1 Mutual Exclusion for IntMutex

We first introduce the following abbreviations for IntMutex and SetMutex. We use $\mathsf{active}(k)$ and $\mathsf{critical}(k)$ defined as follows:

$$\begin{aligned}\mathsf{active}(k) &\overset{\text{def}}{=} \big(pc(k) = 4 \lor pc(k) = 5 \lor pc(k) = 6\big)\\ \mathsf{critical}(k) &\overset{\text{def}}{=} \big(pc(k) = 5 \lor pc(k) = 6\big).\end{aligned}$$

[1] Available at `http://software.imdea.org/leap`

Mutual exclusion is specified as the following 2 index formula:

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

Using the P-INV rule to prove $\square\mathsf{mutex}$ fails for $\tau_4^{(i)}$, because the VC discharged is not valid:

$$\mathsf{mutex}(i,j) \wedge \begin{pmatrix} \mathit{pc}(i) = 4 \wedge \mathit{pc}' = \mathit{pc}\{i \leftarrow 5\} & \wedge \\ \mathit{ticket}(i) = \mathit{min} & \wedge \\ \mathit{pres}(\mathit{avail}, \mathit{min}, \mathit{ticket}(i), \mathit{ticket}(j)) & \end{pmatrix} \rightarrow \mathsf{mutex}'(i,j)$$

The SMT Solver reports two counter models:

$$\mathit{pc}(j) = 5 \wedge \mathit{min} = 1 \wedge \mathit{avail} = 2 \wedge \mathit{ticket}(i) = 1 \wedge \mathit{ticket}(j) = 3 \qquad (8)$$

and

$$\mathit{pc}(j) = 5 \wedge \mathit{min} = 1 \wedge \mathit{avail} = 2 \wedge \mathit{ticket}(i) = 1 \wedge \mathit{ticket}(j) = 1 \qquad (9)$$

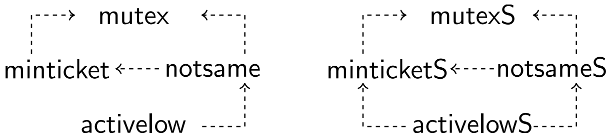
Each of these models illustrate that the VC is not valid. Hence, mutex is not inductive. The formula $\mathsf{mutex}(i,j)$ by itself does not include information about two important facts of the program. First, if a thread is in the critical section, then it owns the minimum announced ticket, unlike in the counter-model (8). Second, the same ticket cannot be given to two different threads, unlike in the second counter-model (9). Two new auxiliary support invariants encode these facts:

$$\begin{aligned} \mathsf{minticket}(i) &\overset{\text{def}}{=} \big(\mathsf{critical}(i) \rightarrow \mathit{min} = \mathit{ticket}(i)\big) \\ \mathsf{notsame}(i,j) &\overset{\text{def}}{=} \big(i \neq j \wedge \mathsf{active}(i) \wedge \mathsf{active}(j) \rightarrow \mathit{ticket}(i) \neq \mathit{ticket}(j)\big) \end{aligned}$$

Using SP-INV, we can prove that mutex is invariant, using minticket and notsame as support, except for the fact that minticket is not inductive. When trying to use P-INV to prove minticket invariant the solver reports that if two different threads i and j are in the critical section with the same ticket and $\tau_6^{(j)}$ is taken, then $\mathsf{minticket}(i)$ does not hold in the post state. Using notsame as support allows to prove minticket, but notsame is also not inductive. In this case, the offending transition is τ_3 when an existing ticket is reused. The following invariant rules out this spurious case:

$$\mathsf{activelow}(i) \overset{\text{def}}{=} \big(\mathsf{active}(i) \rightarrow \mathit{ticket}(i) < \mathit{avail}\big)$$

The formula activelow is inductive and can be proved directly using P-INV. Also, activelow is enough to support notsame, so the proof is completed. The dependencies between invariants are shown in Fig. 7(a).



(a) IntMutex (b) SetMutex

Fig. 7: Proof graph showing the dependencies between invariants

4.2 Mutual Exclusion for SetMutex:

To prove mutual exclusion for SetMutex we proceed in a similar way as for IntMutex. The invariants mutexS, $\mathsf{notsameS}$ and $\mathsf{activelowS}$ are identically to mutex, $\mathsf{notsame}$ and $\mathsf{activelow}$. The invariant $\mathsf{minticketS}$ is now defined as follows:

$$\mathsf{minticketS}(i) \overset{\text{def}}{=} \mathsf{critical}(i) \rightarrow \mathit{bag}.\mathit{min} = \mathit{ticket}(i)$$

To prove that mutexS is invariant it is enough to use $\mathsf{minticketS}$ and $\mathsf{notsameS}$ as support. This time, to prove the invariance of $\mathsf{minticketS}$ requires the use of $\mathsf{activelowS}$ in addition to $\mathsf{notsameS}$ as support. This extra support is required to encode that when a thread takes transition τ_3, it adds to *bag* a value strictly greater than any other previously assigned ticket. Finally, $\mathsf{notsameS}$ relies on $\mathsf{activelowS}$, which again, is an inductive invariant. Fig 7(b) contains the proof graph that describes the dependencies between these invariants.

4.3 List Preservation and Set Representation for Lock-coupling Concurrent Lists

We now present a larger real-world example of a concurrent data-type verified using the parametrized proof rules presented in this paper: a concurrent lock-coupling list, whose pseudo-code is shown in Fig. 8.

Concurrent lock-coupling lists implement a set [24] by maintaining an ordered list of non-repeating elements. Each node in the list consists of three fields: (1) an element, (2) a pointer to the next node in the list, and (3) a lock used to protect concurrent accesses to the node. To search an element, a thread advances through the list acquiring a lock before visiting a node. This lock is only released after the lock of the next node has been acquired. Concurrent lists also maintain two sentinel nodes, *head* and *tail*, with phantom values representing the lowest and highest possible values, $-\infty$ and $+\infty$ respectively. Sentinel nodes are not modified at runtime. We define two "ghost" variables that aid the verification: *reg*, a set of addresses of the memory locations of nodes in the list; and *elems*, a set of elements we use to keep track of elements contained in the list. Ghost variables are compiled away and are only used in the verification process. In Fig. 8 ghost variables (*reg* and *elems*) and ghost code (program lines 37 and 55) appear inside a box.

```
global
  $Addr\ head;\ Addr\ tail;$
  $\boxed{Set\langle Addr\rangle\ reg;\ Set\langle Elem\rangle\ elems;}$
assume
  $reg = \{head, tail, null\}$
$\wedge\ elems = \{head\!\rightarrow\!data, tail\!\rightarrow\!data\}$
$\wedge\ head \neq tail \wedge head \neq null \wedge tail \neq null$
$\wedge\ head\!\rightarrow\!data = -\infty \wedge tail\!\rightarrow\!data = +\infty$
$\wedge\ head\!\rightarrow\!next = tail \wedge tail\!\rightarrow\!next = null$

procedure MGC
  $Elem\ e$
begin
while true do
    $e :=$ havocListElem()
    nondet
        call SEARCH($e$)
    or
        call INSERT($e$)
    or
        call REMOVE($e$)
end while
end procedure

procedure INSERT($e$)
  $Addr\ \ prev$
  $Addr\ \ curr$
  $Addr\ \ aux$
begin
$prev := head$
$prev\!\rightarrow\!lock()$
$curr := prev\!\rightarrow\!next$
$curr\!\rightarrow\!lock()$
while $curr\!\rightarrow\!data < e$ do
    $aux := prev$
    $prev := curr$
    $aux\!\rightarrow\!unlock()$
    $curr := curr\!\rightarrow\!next$
    $curr\!\rightarrow\!lock()$
end while
if $curr \neq null \wedge curr\!\rightarrow\!data > e$ then
    $aux := malloc(e, null, \#)$
    $aux\!\rightarrow\!next := curr$
    $prev\!\rightarrow\!next := aux$
    $\boxed{\begin{array}{l} reg := reg \cup \{aux\} \\ elems := elems \cup \{e\} \end{array}}$
end if
$prev\!\rightarrow\!unlock()$
$curr\!\rightarrow\!unlock()$
return
end procedure
```

```
procedure SEARCH($e$)
  $Addr\ \ prev$
  $Addr\ \ curr$
  $Addr\ \ aux$
  $Bool\ \ found$
begin
$prev := head$
$prev\!\rightarrow\!lock()$
$curr := prev\!\rightarrow\!next$
$curr\!\rightarrow\!lock$
while $curr\!\rightarrow\!data < e$ do
    $aux := prev$
    $prev := curr$
    $aux\!\rightarrow\!unlock()$
    $curr := curr\!\rightarrow\!next$
    $curr\!\rightarrow\!lock()$
end while
$found := (curr\!\rightarrow\!data = e)$
$prev\!\rightarrow\!unlock()$
$curr\!\rightarrow\!unlock()$
return $found$
end procedure

procedure REMOVE($e$)
  $Addr\ \ prev$
  $Addr\ \ curr$
  $Addr\ \ aux$
begin
$prev := head$
$prev\!\rightarrow\!lock()$
$curr := prev\!\rightarrow\!next$
$curr\!\rightarrow\!lock()$
while $curr\!\rightarrow\!data < e$ do
    $aux := prev$
    $prev := curr$
    $aux\!\rightarrow\!unlock()$
    $curr := curr\!\rightarrow\!next$
    $curr\!\rightarrow\!lock()$
end while
if $\begin{pmatrix} curr \neq tail\ \wedge \\ curr\!\rightarrow\!data = e \end{pmatrix}$ then
    $aux := curr\!\rightarrow\!next$
    $prev\!\rightarrow\!next := aux$
    $\boxed{\begin{array}{l} reg := reg \setminus \{curr\} \\ elems := elems \setminus \{e\} \end{array}}$
end if
$prev\!\rightarrow\!unlock()$
$curr\!\rightarrow\!unlock()$
return
end procedure
```

Fig. 8: Concurrent lock-coupling list implementation

In particular, ghost code at program locations 37 and 55 is simply used for updating the values of *reg* and *elems* at the exact point in which a new cell is added or removed from the skiplist. When invoking *malloc* in program line 35 we use the # symbol to denote that the lock is initially available, i.e., no thread has initially acquired the lock. Hence, $malloc(e, null, \#)$ returns a cell which stores element e, whose *next* pointer points to *null* and whose lock is unlocked. Concurrent lock-coupling lists provide three main operations: (a) Search: finds an element in the list; (b) Insert: adds a new element to the list; and (c) Remove: deletes an element from the list. For verification purposes we define the most general client MGC—shown also in Fig. 8—which non-deterministically chooses a method and its parameters. We verify a parametrized system whose processes run the MGC.

For lock-coupling concurrent lists we prove that the most general client of the concurrent lock-coupling list implementation in Fig. 8 satisfies:

1. the layout in the heap is always that of a list;
2. the data-type implements a set, whose elements correspond to the set of elements *elems*.

We use the theory TLL3 (see [37]) to describe the property of list shape preservation. This theory allows to reason about addresses, elements, locks, sets, order, cells (i.e., list nodes), memory and list reachability. A cell is a structure containing an element, a pointer to next node in the list and a lock to protect the cell. A lock is associated with operations *lock* and *unlock* to acquire and release. The memory (*heap*) is modeled as an array of cells indexed by addresses (which is equivalent to a map from addresses to cells). List shape preservation is modeled as the following formula:

$$\mathsf{list} \overset{\text{def}}{=} \begin{cases} null \in reg \wedge reg = addr2set(heap, head) \wedge head \neq tail & \wedge \quad (\text{L1}) \\ heap[tail].next = null \wedge tail \neq null \wedge head \neq null & \wedge \quad (\text{L2}) \\ heap[head].data = -\infty \wedge heap[tail].data = +\infty & \wedge \quad (\text{L3}) \\ elems = set2elemset(heap, reg \setminus \{null\}) & \wedge \quad (\text{L4}) \\ Ordered(heap, head, tail) & \qquad (\text{L5}) \end{cases}$$

The formula list is 0-index because it only refers to global program variables. (L1) establishes that *null* belongs to *reg* and that *reg* is exactly the set of addresses reachable in the *heap* starting from *head*, which ensures that the list is acyclic. (L2) and (L3) express some sanity properties of the sentinel nodes *head* and *tail*. (L4) establishes that *elems* is the set of elements in cells referenced by addresses in *reg*, except for the element at the cell pointed by *null*. Finally, (L5) express the fact that the list is ordered.

Using p-inv, Leap can establish that list holds initially, but fails to prove that list is preserved by all transitions. As in the previous examples, the use of decision procedures for proving verification conditions allows to obtain counter-examples as models of an execution step that leads to a violation of the desired invariant. Leap parses the counterexample returned by the SMT solver, which is usually very small, involves only few threads and allows to understand the missing information. In practice, these models alleviate the human ingenuity required to produce intermediate support invariants. We

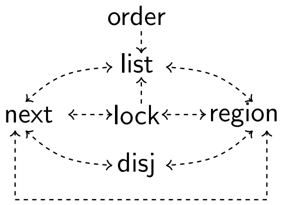


Fig. 9: Invariant dependencies to prove that list is an invariant for concurrent lock-coupling lists

introduce some support invariants that allow to prove list. Here we just sketch the support invariants used in the verification process. The full description of each invariant can be downloaded from the LEAP web site[2].

The formula region is a 1-index formula that describes that local variables *prev*, *curr* and *aux* point to cells contained in region *reg*. The auxiliary invariant next captures the relative position in the list of the cells pointed by *head* and *tail* and local variables *prev*, *curr* and *aux*. This invariant is needed to prove (L2). To prove (L3) and (L4) we need to show that the order is preserved. The invariant candidate order captures the increasing order between the data in cells pointed by *curr*, *prev* and *aux* among themselves, and with respect to the element e used as a parameter to the SEARCH, INSERT and REMOVE functions. The auxiliary invariant lock identifies those program locations at which a thread owns a cell in the heap by means of acquiring a lock. Finally, we introduce the formula disj, which encodes that invocations to *malloc* by different threads return different fresh cells. The formula disj is a 2-index formula, because it needs to refer to local variables of two different threads:

$$\mathsf{disj}(i,j) \overset{\text{def}}{=} (i \neq j \wedge pc(i) = 36, 37 \wedge pc(j) = 36, 37) \ \rightarrow \ aux(i) \neq aux(j)$$

Fig. 9 shows the proof graph encoding the proof of list shape preservation.

We also verify other properties of the concurrent lock-coupling list implementation, including the following functional specifications expressed as invariants:

- funSchL, which establishes that the result of SEARCH corresponds to whether the searched element e is present at the linearization point of SEARCH:

  $$\mathsf{funSchL}(i) \overset{\text{def}}{=} pc(i) = 19 \rightarrow (heap[curr(i)].data = e(i) \leftrightarrow e(i) \in elems)$$

  This specification states that after the loop, if the element stored at *curr* is e, then e belongs to the set of elements represented by the list. That is, the element is found by the SEARCH procedure if and only if the element is in the list.

[2] Full examples available at `http://software.imdea.org/leap/examples.html`

- $\mathsf{funSchI}$, which states that if a search is successful then e was inserted earlier in the history. For this specification we need to slightly modify the program presented in Fig. 8. First we need to declare two new global ghost variables of sort $\mathit{Set}\langle \mathit{Elem}\rangle$ named *histIns* and *histRem*. Variable *histIns* initially contains the same elements as *elems* and it is updated using ghost code at line 37 of program Insert by inserting element e. Variable *histRem* initially is assigned to the empty set and it is updated as ghost code at line 55 of program Remove by removing element e. Then, the specification of $\mathsf{funSchL}$ can be defined as:

$$\mathsf{funSchI}(i) \overset{\text{def}}{=} \begin{cases} \mathsf{funSchL}(i) & \wedge \quad (\text{FI1}) \\ (pc(i) = 20..22 \wedge \mathit{found}) \ \rightarrow \ e(i) \in \mathit{histIns} & \wedge \quad (\text{FI2}) \\ \mathit{elems} \subseteq \mathit{histIns} & \qquad (\text{FI3}) \end{cases}$$

Sub-formula (FI2) establishes that if element e was found by Search then e must have been previously inserted by some thread in the list, and (FI3) describes the fact that all elements present in the list were previously inserted.

- $\mathsf{funSchR}$, which captures the fact that if a search is unsuccessful then either e was never inserted or it was removed, but in any case was not present at the linearization point of Search. To describe this specification, we need to additionally declare two local ghost variables of sort $\mathit{Set}\langle \mathit{Elem}\rangle$ named *histICopy* and *histRCopy*. These variables are updated at line 19 with the values of *histIns* and *histRem* respectively and their purpose is just to keep a copy of the state of *histIns* and *histRem* at the linearization point of Search. Then, the specification of $\mathsf{funSchR}$ is:

$$\mathsf{funSchR}(i) \overset{\text{def}}{=} \begin{cases} \mathsf{funSchL}(i) & \wedge \quad (\text{FR1}) \\ \begin{pmatrix} pc(i) = 20..22 \\ \wedge \neg \mathit{found} \end{pmatrix} \rightarrow \begin{pmatrix} e(i) \notin \mathit{histICopy} \ \vee \\ e(i) \in \mathit{histRCopy} \end{pmatrix} & \wedge \quad (\text{FR2}) \\ \mathit{histIns} \subseteq (\mathit{elems} \cup \mathit{histRem}) & \qquad (\text{FR3}) \end{cases}$$

Formula (FR2) establishes that, at the end of the loop, if element e was not found in the list then either e has never been inserted into the list or e has been removed by a call to Remove. Finally, (FR3) describes that all elements that have been historically inserted into the list by a call to Insert are still in the list or they have been removed from the list using a call to Remove.

Additionally, we prove that $\mathsf{funRem}(i)$, $\mathsf{funIns}(i)$ and $\mathsf{funSch}(i)$ are invariants. These formulas consider the case in which one thread handles different elements than all other threads. In this case, the specification is similar to a sequential functional specification: an element is found if and only if it is in the list, an element is not present after removal, and an element is present after insertion.

4.4 Unbounded Total Concurrent Queue

We also verify an implementation of unbounded concurrent queue from [24] that uses internally a single-linked list. The implementation is shown in

```
global
  $Addr\ head,\ tail;$
  $Lock\ queueLock;$
  $Set\langle Addr\rangle\ reg;$
  $Set\langle Elem\rangle\ enqueueSet = \emptyset;$
  $Set\langle Elem\rangle\ dequeueSet = \emptyset;$

procedure ENQUEUE($Elem\ e$)
  $Addr\ n$
begin
  $queueLock.lock()$
  $n := malloc(e)$
  $n \rightarrow next := null$
  $tail \rightarrow next := n$
  $reg := reg \cup \{n\}$
  $enqueueSet := enqueueSet \cup \{e\}$
  $tail := n$
  $queueLock.unlock()$
  return()
end procedure

procedure DEQUEUE()
  $Elem\ result$
begin
  $queueLock.lock()$
  if $head \rightarrow next = null$ then
    $queueLock.unlock()$
    raise($EmptyException$)
  end if
  $result := head \rightarrow next \rightarrow data$
  $head := head \rightarrow next$
  $reg := reg \setminus \{head\}$
  $dequeueSet := dequeueSet \cup \{result\}$
  $queueLock.unlock()$
  return($result$)
end procedure
```

Fig. 10: Concurrent unbounded queue implementation

Fig. 10. The queue contains two sentinel nodes, named *head* and *tail*. An element e is inserted in the queue using procedure ENQUEUE, which appends a cell containing e them to *tail* of the list. Procedure DEQUEUE removes the cell pointed by *head* and returns the element that stored in that cell. The implementation we analyze uses a single lock called *queueLock* to protect both procedure body, which then execute atomically. This is an example of a coarse-grain concurrent data-type.

For this implementation we prove queue shape preservation expressed as the formula UQPres:

$$\mathsf{UQPres} \overset{\text{def}}{=} \begin{cases} null \in reg \ \wedge \ tail \in reg \ \wedge \ tail \neq null \ \wedge \\ reg = addr2set(heap, head) \ \wedge \ head \neq null \end{cases}$$

We also prove that UQInc is invariant This formula states that all elements that have been inserted at some point in the queue are still in the queue or they have been removed by a call to DEQUEUE.

$$\mathsf{UQInc} \overset{\text{def}}{=} \begin{pmatrix} set2elemset(heap, reg \setminus \{head, null\}) \\ \cup \\ deqSet \end{pmatrix}$$

Function *set2elemset* is defined of theory TLL3. Given a heap h and a set of addresses s, the term $set2elemset(h, s)$ corresponds to the set of elements stored in cells pointed by addresses in s according to h. Note that we do not consider the addresses *null* and *head* since in this implementation *head* is used as a sentinel node. We also define two auxiliary formulas named UQNext and UQLock to describe the relation between pointers and the behavior of the global lock.

```
global
  $Addr\ top$;
  $Set\langle Addr\rangle\ reg = \emptyset$;
  $Set\langle Elem\rangle\ pushSet = \emptyset$;
  $Set\langle Elem\rangle\ popSet = \emptyset$;

procedure PUSH($Elem\ e$)
  $Addr\ oldTop,\ newTop,\ n$;
begin
$n := malloc(e)$
$n \rightarrow next := null$
while true do
    $oldTop := top$
    $n \rightarrow next := oldTop$
    if $CAS(top, oldTop, n)$ then
        if $top = oldTop$ then
          $reg := reg \cup \{n\}$
          $pushSet := pushSet \cup \{e\}$
        endif
        return()
    end if
end while
end procedure

procedure POP()
  $Addr\ oldTop,\ newTop$;
begin
while true do
    $oldTop := top$
    if $oldTop = null$ then
        raise($EmptyException$)
    end if
    $newTop := oldTop \rightarrow next$
    if $CAS(top, oldTop, newTop)$ then
        if $top = oldTop$ then
          $reg := reg \setminus \{oldTop\}$
          $popSet := popSet \cup \{oldTop \rightarrow data\}$
        endif
        return($oldTop \rightarrow data$)
    end if
end while
end procedure
```

Fig. 11: Lock-free implementations of a stack.

4.5 Lock-free queues and stacks

Finally, we also verify two more concurrent datatypes: an implementation of a lock-free stack [24] and the lock-free non-blocking implementation of a queue known as Michael-Scott queue [32].

In both cases we verify similar specifications to those for lock-coupling lists in Section 4.3 and unbounded queues in Section 4.4. More specifically, for the lock-free stack implementation presented in Fig. 11 we prove the following invariants:

- LFSPres, which states that the PUSH and POP operations preserve the list shape of the stack.
- LFSRegion, which captures the structure of the heap region where the nodes of the stack are located.
- LFSReach, that describes which nodes are reachable from the head of the stack.
- LFSNext, which states the relation between nodes *top*, *oldTop*, *newTop* and n, in terms of their *next* field, during the execution of PUSH and POP.
- LFSInc, which establishes that all elements inserted into the stack using a call to PUSH are either still in the stack or they have been removed by a call to POP.
- LFSDisj, which encodes the fact that invocations to *malloc* performed by different threads return different fresh cells.
- LFSVals, which establishes that the procedure arguments are precisely the elements inserted or removed.

```
global
  $Addr\ head,\ tail;$
  $Set\langle Addr\rangle\ reg = \emptyset;$
  $Set\langle Elem\rangle\ enqSet = \emptyset;$
  $Set\langle Elem\rangle\ deqSet = \emptyset;$
```

```
procedure ENQUEUE($Elem\ e$)
  $Addr\ last,\ nextptr,\ n;$
begin
$n := malloc(e)$
$n \rightarrow next := null$
while true do
    $last := tail$
    $nextptr := last \rightarrow next$
    if $last = tail$ then
        if $nextptr = null$ then
            if $\underline{CAS(last \rightarrow next, nextptr, n)}$ then
                ┌─────────────────────────────────────┐
                │ if $last \rightarrow next = nextptr$ then │
                │   $reg := reg \cup \{n\}$                 │
                │   $enqSet := enqSet \cup \{e\}$           │
                │ endif                                      │
                └─────────────────────────────────────┘
                break
            end if
        else
            $CAS(tail, last, nextptr)$
        end if
    end if
end while
$CAS(tail, last, n)$
return()
end procedure
```

```
procedure DEQUEUE()
  $Addr\ first,\ last,\ nextptr;$
  $Elem\ value;$
begin
while true do
    $first := head$
    $last := tail$
    $nextptr := first \rightarrow next$
    if $first = head$ then
        if $first = last$ then
            if $nextptr = null$ then
                raise($EmptyException$)
            end if
            $CAS(tail, last, nextptr)$
        else
            $value := nextptr \rightarrow data$
            if $\underline{CAS(head, first, nextptr)}$ then
                ┌──────────────────────────────────────┐
                │ if $head = first$ then                      │
                │   $reg := reg \setminus \{first\}$          │
                │   $deqSet := deqSet \cup \{value\}$         │
                │ endif                                        │
                └──────────────────────────────────────┘
                break
            end if
        end if
    end if
end while
return($value$)
end procedure
```

Fig. 12: Lock-free implementations of a queue.

Similarly, for the lock-free queue implementation shown in Fig. 12 we prove the following invariants:

- LFQPres, which states that calls to ENQUEUE and DEQUEUE preserve the list shape of the queue data structure.
- LFQRegion, which describes the composition of the region of the heap containing the nodes in the queue.
- LFQReach, that establishes the reachability relation between *head*, *tail* and the nodes traversed during the execution of ENQUEUE and DEQUEUE.
- LFQNext, which establishes the relation between *head*, *tail*, *first*, *last*, *nextptr* and *n* in terms of their *next* field.
- LFQInc, that states that all elements inserted into the queue using a call to ENQUEUE are still in the queue of have been removed as a result of a call to DEQUEUE.
- LFQDisj, which encodes the fact that invocations to *malloc* performed by different threads return different fresh cells.

4.6 Experimental Results

Fig. 13 contains the results of our empirical evaluation, executed on a computer with a 2.8 GHz processor and 8GB of memory. Each row reports the empirical results obtained when proving a single invariant. Rows 1 to 4 correspond to the mutual exclusion protocol based on integers presented in Section 4.1. Rows 5 to 8 contain the invariants for the mutual exclusion protocol based on sets presented in Section 4.2. Rows 9 to 20 present the invariants for the concurrent lock-coupling single-linked lists presented in Section 4.3. Rows 21 to 24 contain the results for unbounded concurrent queues, presented in Section 4.4. Finally, rows 25 to 31 and rows 32 to 37 correspond to the invariants for the lock-free implementation of a stack and a lock-free implementation of a queue respectively, presented in Section 4.5.

In the table, the first two columns enumerate the examples and present the name of the formulas. The following four columns show:

- "id": The index of the formula, that is, the number of threads which parameterizing the formula. For instance, list is a 0-index formula because it only uses global variables. On the other hand, a specification like mutexS describes a relation involving the program counter of two different threads, and thus it is a 2-index formula.
- "#vc": The total number of generated verification conditions.
- "pos": The number of VCs successfully proved by a position decision procedure, which can only reasoning about program locations and consider all other program predicates uninterpreted. This decision procedure is very fast but can only solve simple verification conditions.
- "dp": The number of the remaining VCs proved by an specialized decision procedure. For invariants in lines 1 to 8 we use the decision procedure for Presburguer arithmetic with sets. For invariants in lines 9 to 37 we use the TLL3 decision procedure.

When discharging a VC, Leap first tries to use a positional decision procedure, which can quickly verify VCs that are provable valid using simple reasoning about program locations. The specialized decision procedure is invoked only for those VCs for which the positional decision procedure fails. The total number of VCs proved for a given candidate invariant is then the sum of columns "pos" and "dp". Consequently, if this sum equals the total number of VCs discharged (column "#vc") all VCs are valid and the formula is an invariant.

The final four columns in the table show the total running time required by the specialized decision procedures, with trying four different approaches:

- "Full": which corresponds to naively instantiating all support invariants for all VCs. This is equivalent to trying to solve the VCs by brute force, passing the resulting formula directly to the decision procedure.
- "Supp": which corresponds to instantiate only the necessary support.
- "Offend": which corresponds to instantiating the support as in "Supp" but only in potentially offending transitions, which are those transitions that modify a program variable in the formula.
- "Tactics": which reports the running time required after using some standard first-order tactics like lazy instantiation and formula normalization

		form. info		#solved vc		Solving time for all VCs (sec.)			
		id	#vc	pos	dp	Full	Supp	Offend	Tactics
1	mutex	2	28	26	2	0.32	0.23	0.10	0.01
2	minticket	1	19	18	1	0.04	0.04	0.01	0.01
3	notsame	2	28	26	2	0.13	0.13	0.10	0.02
4	activelow	1	19	17	2	0.01	0.01	0.01	0.01
5	mutexS	2	28	26	2	0.44	0.38	0.14	0.04
6	minticketS	1	19	18	1	0.31	0.18	0.08	0.01
7	notsameS	2	28	26	2	0.14	0.13	0.10	0.02
8	activelowS	1	19	17	2	0.02	0.02	0.02	0.01
9	list	0	61	38	23	*TO*	*TO*	*TO*	12.85
10	order	1	121	62	59	998.35	7.56	2.69	1.20
11	lock	1	121	76	45	778.15	4.82	1.44	0.50
12	next	1	121	60	61	*TO*	*TO*	26.58	1.76
13	region	1	121	95	26	*TO*	*TO*	85.27	25.67
14	disj	2	181	177	4	121.74	1.29	1.29	0.22
15	funSchL	1	121	97	24	*TO*	*TO*	82.13	4.63
16	funSchI	1	121	93	28	*TO*	*TO*	80.20	5.00
17	funSchR	1	121	93	28	*TO*	*TO*	110.84	5.49
18	funSch	1	208	198	10	*TO*	*TO*	6.14	4.55
19	funIns	1	208	200	8	*TO*	*TO*	2.04	0.51
20	funRem	1	208	200	8	*TO*	*TO*	2.73	1.56
21	UQPres	0	23	18	5	1.13	0.90	0.35	0.18
22	UQNext	1	45	35	10	0.98	0.20	0.18	0.11
23	UQLock	1	45	33	12	0.18	0.08	0.07	0.07
24	UQInc	0	23	19	4	0.74	0.64	0.34	0.30
25	LFSPres	0	37	30	7	29.43	0.70	0.32	0.05
26	LFSRegion	1	73	69	4	1.54	0.16	0.10	0.05
27	LFSReach	1	109	90	19	*TO*	*TO*	16.81	24.35
28	LFSNext	1	73	63	10	1.13	0.50	0.31	0.22
29	LFSInc	0	37	30	7	3.78	2.38	1.12	0.05
30	LFSDisj	2	109	105	4	0.63	0.23	0.20	0.19
31	LFSVals	1	73	62	11	0.42	0.16	0.15	0.06
32	LFQPres	1	103	78	25	*TO*	*TO*	*TO*	2.85
33	LFQRegion	1	103	99	4	113.97	0.30	0.29	0.07
34	LFQReach	1	103	81	22	*TO*	*TO*	22.78	96.55
35	LFQNext	1	103	76	27	622.41	53.74	6.12	8.20
36	LFQInc	0	52	40	12	438.30	41.24	27.96	0.20
37	LFQDisj	2	154	150	4	1.89	0.46	0.51	0.30

Fig. 13: VCs proven using each decision procedure and running times.

and propagation. These tactics allow to simplify the formula before invoking the decision procedures, sometimes at the price of requiring several invocations. These simplifications lead to smaller cut-offs and faster search times for the decision procedure.

TO represents a timeout of 30 minutes. Our results suggest that, in practice, tactics are essential for efficiency when handling non-trivial examples such as concurrent lists. Also, our decision procedures have room for implementation improvements which would lead to faster running times.

An alternative proof method for generating and proving invariants is to compute an over-approximation of the reachable state space by iteratively computing formulas in sophisticated logics for heap shapes. However, our results suggest that this approach is not likely to be feasible for complicated heap manipulating programs. Instead, we propose to use proof rules like the ones presented in this paper and improve automation via researching decision procedures, combining automated first-order reasoning and decision procedures to improve the efficiency in proving each VC, and invariant generation techniques to alleviate the human intervention.

5 Concluding Remarks

This paper has introduced a temporal deductive technique for the uniform verification problem of safety properties of parametrized infinite state processes, in particular for the verification of concurrent datatypes that manipulate data in the heap. Our proof rules automatically discharge a finite collection of verification conditions. The size of this collection depends on the program description and the index of the formula to prove, but not on the number of threads in a particular instance. Each VC describes a small-step in the execution of all corresponding instances. The VCs are quantifier-free as long as the formulas are quantifier free. We use the theory of arrays [9] to encode the local variables of a system with an arbitrary number of threads, but the dependencies with arrays can be eliminated, under the assumption of full symmetry. It is immediate to extend our framework to a finite family of process classes, for example to model client/server systems.

An interesting research direction is to relax the requirement of full symmetry to cover other process topologies, like for example process rings or totally order processes. Handling these topologies requires to specialize the proof rules in this paper by adapting the premises that refer to threads not in the formula (premises P3, S3, R3 and G3) to consider all cases according to the topology. For example, totally order processes would require to split the case $i \neq j$ into $(i < j \vee i > j)$.

Our main goal is the development of a framework for the deductive verification of temporal properties of parametrized systems. In this paper we tackle the verification of safety properties through the introduction of the parametrized invariance proof rules. For liveness properties we propose the use of parametrized verification diagrams (PVD), an extension of general verification diagrams [10], amenable for the verification over parametrized systems. PVD are studied in a companion paper [38] based on the results presented here.

Future work includes invariant generation to simplify or even automate proofs. We are studying how to apply the decision procedures with the calculation of precondition formulas (like [27]), extended to parametrized systems, to effectively infer candidate invariants from the target specification. We are also studying how to extend the "invisible invariant" approach [6, 35, 47] to processes that manipulate infinite state, not only by instantiating small systems with a few threads (like in invisible invariants) but also by limiting the counter-model exploration to a bounded size, heuristically determined. The candidate invariants produced this way must then be verified with the proof rules presented in this paper for the unrestricted system. We envision this method to be a smart exploration of the space of candidate invariants.

We are also extending our previous work on abstract interpretation-based invariant generation for parametrized systems [39] to handle complex datatypes. Our work in [39] was restricted to numerical domains.

As our empirical evaluation suggests, the instantiation of support is critical to the efficiency of the decision procedure and hence to the effectiveness of our verification method. This is because the size of the formula passed to the decision procedure depends heavily on the instantiation of support. Our current tactics for instantiating support are rather heuristic. We plan to research more rigorous and sophisticated methods for instantiation, or even to develop decision procedures that include instantiation. Promising directions for this study are local theory extensions [40] and the search for natural proofs [36]. This line can also potentially lead to complete methods for some class of programs and theories of data [44].

Finally, another approach that we are currently investigating is to use the proof rules presented here to enable a Horn-Clause Verification engine [23] to automatically generate parametrized invariants guided by the invariant candidate goal. Our preliminary results are promising but out of the scope of this paper.

Acknowledgements This work was funded in part by the Spanish Ministry of Economy under project "TIN2012-39391-C04-01 STRONGSOFT" and by the Madrid Regional Government under project "S2013/ICE-2731 N-Greens Software-CM".

References

1. Abdulla, P.A., Bouajjani, A., Jonsson, B., Nilsson, M.: Handling global conditions in parametrized system verification. In: Proc. of CAV'99. pp. 134–145 (1999)
2. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: Proc. of LICS'96. pp. 313–321. IEEE Computer Society (1996)
3. Abdulla, P.A., Delzanno, G., Rezine, A.: Approximated parameterized verification of infinite-state processes with global conditions. FMSD 34(2), 126–156 (2009)
4. Amit, D., Rinetzky, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Proc. of CAV'07. LNCS, vol. 4590, pp. 477–490. Springer (2007)
5. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Information Processing Letters 22(6), 307–309 (1986)
6. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.D.: Parameterized verif. with automatically computed inductive assertions. In: Proc. of CAV'01. LNCS, vol. 2102, pp. 221–234. Springer (2001)
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. Bradley, A.R.: SAT-based model checking without unrolling. In: In Proc. of VMCAI'11. LNCS, vol. 6538, pp. 70–87. Springer (2011)
9. 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)

10. Browne, A., Manna, Z., Sipma, H.B.: Generalized verification diagrams. In: Proc. of FSTTCS'95. LNCS, vol. 1206, pp. 484–498. Springer (1995)
11. 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)
12. Clarke, E.M., Grumberg, O., Browne, M.C.: Reasoning about networks with many identical finite-state processes. In: PODC'86. pp. 240–248. ACM (1986)
13. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. FMSD 9(1/2), 77–104 (1996)
14. 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)
15. Dutertre, B.: Yices 2.2. In: Proc. of CAV'14. LNCS, vol. 8559, pp. 737–744. Springer (2014)
16. 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)
17. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: Proc. of POPL'95. pp. 85–94. ACM (1995)
18. Emerson, E.A., Namjoshi, K.S.: Automatic verification of parameterized synchronous systems. In: Proc. of CAV'96. LNCS, vol. 1102, pp. 87–98. Springer (1996)
19. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. FMSD 9(1/2), 105–131 (1996)
20. Fontaine, P., Ranise, S., Zarba, C.G.: Combining lists with non-stably infinite theories. In: Proc. of LPAR'04. pp. 51–66. Springer (2004)
21. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast decision procedures. In: Proc. of CAV'04. LNCS, vol. 3114, pp. 175–188. Springer (2004)
22. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. of the ACM 39(3), 675–735 (1992)
23. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Proc. of PLDI'12. pp. 415–416. ACM (2012)
24. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgran-Kaufmann (2008)
25. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Proc. of SAT'12. LNCS, vol. 7317, pp. 157–171. Springer (2012)
26. Kesten, Y., Pnueli, A., on Raviv, L.: Algorithmic verification of linear temporal logic specifications. In: Proc. of ICALP'98. LNCS, vol. 1443, pp. 1–16. Springer (1998)
27. 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)
28. Lesens, D., Halbwachs, N., Raymond, P.: Automatic verification of parameterized linear networks of processes. In: Proc. of POPL'97. pp. 346–357. ACM (1997)
29. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: Proc. of POPL'11. pp. 611–622. ACM (2011)
30. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. springer-Verlag (1995)
31. Marco Bozzano, G.D.: Beyond parameterized verification. In: TACAS'02. LNCS, vol. 2280, pp. 221–235. Springer (2002)
32. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. of PODC'96. pp. 267–275 (1996)
33. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. of TACAS'08. LNCS, vol. 4963, pp. 337–340. Springer (2008)
34. Nelson, C.G., Oppen, D.C.: A simplifier based on efficient decision algorithms. In: Proc.of POPL'78. pp. 141–150. ACM (1978)
35. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Proc. of TACAS'01. LNCS, vol. 2031, pp. 82–97. Springer (2001)
36. Qiu, X., Garg, P., Stefanescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: Proc. of PLDI'13. pp. 231–242. ACM (2013)

37. 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)
38. Sánchez, A., Sánchez, C.: Parametrized verification diagrams. In: Proc. of TIME'14. pp. 132–141. IEEE Computer Society Press (2014)
39. Sánchez, A., Sankaranarayanan, S., Sánchez, C., Chang, B.Y.E.: Invariant generation for parametrized systems using self-reflection. In: Proc. of SAS'12. LNCS, vol. 7460, pp. 146–163. Springer (2012)
40. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Proc. of CADE'05. LNCS, vol. 3632, pp. 219–234. Springer (2005)
41. Suzuki, I.: Proving properties of a ring of finite-state machines. Information Processing Letters 28, 213–214 (1988)
42. Tinelli, C., Zarba, C.G.: Combining decision procedures for sorted theories. In: Proc. Logic in Artificial Intelligence (JELIA'04). LNCS, vol. 3229, pp. 641–653. Springer (2004)
43. Tinelli, C., Zarba, C.G.: Combining nonstably infinite theories. Journal of Automatic Reasoning 34, 209–238 (2005)
44. Totla, N., Wies, T.: Complete instantiation-based interpolation. In: Proc. of POPL'13. pp. 537–548. ACM (2013)
45. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: VMCAI. Lecture Notes in Computer Science, vol. 5403, pp. 335–348. Springer (2009)
46. Vafeiadis, V.: Automatically proving linearizability. In: Proc. of CAV'10. LNCS, vol. 6174, pp. 450–464. Springer (2010)
47. Zuck, L.D., Pnueli, A.: Model checking and abstraction to the aid of parameterized systems (a survey). Computer Languages, Systems & Structures 30, 139–169 (2004)