Parametrized Verification Diagrams

Alejandro Sánchez
IMDEA Software Institute, Madrid, Spain
Email: alejandro.sanchez@imdea.org

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

***Abstract*—This paper introduces parametrized verification diagrams (PVDs), a formalism that allows to prove temporal properties of parametrized concurrent systems, in which a given program is executed by an unbounded number of processes.**

PVDs extend general verification diagrams (GVDs). GVDs encode succinctly a proof that a non-parametrized reactive system satisfies a given temporal property. Even though GVDs are known to be sound and complete for non-parametrized systems, proving temporal properties of parametrized systems potentially requires to find a different diagram for each instantiation of the parameter (number of processes). In turn, each diagram requires to discharge and prove a different collection of verification conditions.

PVDs allow a *single* diagram to represent the proof that all instances of the parametrized system for an arbitrary number of threads running concurrently satisfy the temporal specification. Checking the proof represented by a PVD requires proving only a finite collection of quantifier-free verification conditions.

The PVDs we present here exploit the symmetry assumption, under which process identifiers are interchangeable. This assumption covers a large class of concurrent systems, including concurrent datatypes. We illustrate the use of PVDs in the verification of an infinite state mutual exclusion protocol.

***Keywords*-temporal logic; formal verification; formal methods; liveness properties; parametrized systems; concurrent datatypes**

I. Introduction

Verification diagrams [1], [2] allow to prove temporal specifications of reactive systems (finite or infinite state), in particular of concurrent programs. A diagram is essentially an abstraction of the system built specifically for the temporal property under consideration. A successful diagram is precise enough to formally represent the temporal proof, and allows to check mechanically the correctness of the proof.

Formal verification using general verification diagrams starts from a program and a specification given in linear temporal logic. The semantics of the program are represented as a *fair transition system* (FTS) that encompasses all executions of the program. A verification diagram encodes a proof that all the executions covered by the FTS satisfy the given temporal property. Checking the proof encoded in the diagram requires two activities. First, the validity of a finite collection of verification conditions (VCs) – automatically generated from the FTS of the program – guarantees that the diagram covers all (fair) executions of the system. Second, a finite state model checking algorithm ensures that every fair path of the diagram satisfies the temporal property. This second problem can be fully automated. The first part can be handled with decision procedures for the underlying data that the program manipulates, like Boolean, integers, lists in the heap, etc. This way, GVDs cleanly separates two concerns: the temporal reasoning and the data-manipulation.

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

To solve this issue we introduce in this paper Parametrized Verification Diagrams (PVDs). PVDs enrich verification diagrams with capabilities to reason about executions with an arbitrary number of symmetric threads. Checking the proof represented by a PVD requires to handle a single finite collection of verification conditions and to solve a single finite-state model-checking problem. Success in proving each of these obligations guarantees that the property holds for *all* parameter instances. The key idea behind PVDs is that a proof usually only requires to reason about a finite number of processes at each point. This finite collection includes (1) the processes referred to in the property, (2) other processes with particular remarkable characteristic for the proof in the given state (e.g., a leader), and (3) one fresh thread id representing an arbitrary process. The PVDs we present in this paper rely on the symmetry assumption. This assumption states that process identifiers are interchangeable and are only compared for equality and inequality. Swapping identifiers in a given legal execution produces another legal execution. Even though some protocols are not symmetric, full symmetry covers an important class of concurrent systems: concurrent datatypes [3], which are an efficient approach to exploit the parallelism of modern multiprocessor

architectures. Concurrent datatypes allow the simultaneous access to shared data, and are very hard to design, implement and prove correct. In this line of work we aim at verifying liveness properties of concurrent datatypes.

The main difficulty in reasoning about concurrent datatypes comes from the interaction of *unstructured unbounded* concurrency, and heap manipulation. "Unstructured" concurrency refers to programs that are not structured in sections protected by locks and with clear memory footprints, but to programs that allow a more liberal pattern of memory accesses. "Unbounded" refers to the aforementioned lack of a-priori bound on the number of threads.

The success of separation logic [4] and pointer logics [5], [6] to deal with sequential heap manipulating programs has influenced much research [7]–[10] to extended these logics for concurrent programs, but unbounded unstructured concurrency is still very challenging. Moreover, virtually all these approaches are restricted to safety properties.

Our approach to the verification of concurrent datatypes is complementary. We start from a very powerful technique to reason about concurrent systems: the Manna-Pnueli [11] temporal deductive approach, and in particular general verification diagrams. Then, we extend verification diagrams for parametrized systems and equip these diagrams with capabilities to reason about the heap (see [12], [13] for some examples of decision procedures for heap data-types).

The work that is closest to ours is [14] and chapter 8 of [15], which also use diagrams to verify temporal properties of reactive systems. However, they use quantification in the diagram nodes and hence generate quantified VCs. In many cases, using quantifiers sacrifices the automation in the proof of verification conditions (and in the model-checking problem). In this paper, we generate quantifier-free VCs that can be handled automatically by SMT solvers.

Environment abstraction [16] and thread quantification abstractions [17] are abstraction based techniques to deal with parametrized systems, but these systems abstract the number of processes and the data altogether, and are much more difficult to extend to arbitrary data and memory layouts. In contrast, our approach can be applied to any datatype as long as there is a decision procedure for its memory state. This is also a key difference between our approach and paramerized model checking for symmetric systems [18]. Finally, [19] shown a verification approach which uses abstract transition systems to simulate lock-free algorithms, however it is limited only to safety and the simulation obscures the verification. Besides, our approach is also suitable for lock-free algorithms.

The rest of the paper is structured as follows. Section II presents a simple mutual exclusion protocol based on tickets, which we will use as a running example. Section III includes the preliminaries and notation. Section IV presents parameterized verification diagrams and shows they are sound as a proof system. Section V illustrates the use of PVDs to prove a liveness property of a simple mutual exclusion protocol for unbounded and infinite state processes. Finally, Section VI presents the conclusion and future work.

```
   global
     int avail := 0
     set⟨int, tid⟩ bag := ∅

   procedure MutExc
     int ticket
   begin
1: loop
2:     noncritical
3:     ⟨ ticket := avail++
       ⟨ bag.add(ticket, myId) ⟩
4:     await (bag.min == ticket)
5:     critical
6:     bag.remove(ticket, myId)
7: end loop
   end procedure
```

Figure 1: MutExc: Mutual exclusion algorithm

II. Running Example

For illustration purposes we use a simple programming language similar to SPL [11]. A parametrized program P is described by a sequence of instructions. Each instruction is identified uniquely using a finite set of program locations loc ranging from 1 up to L. A program P also contains a number of typed variable identifiers, partitioned into *global* variables and *local* variables. We use V_{global} for the set of global variables and V_{local} for the set of local variables. A parametrized program is run in parallel by a collection of processes. The size of this collection is not known a-priori. We consider in this paper asynchronous interleaving semantics as the semantics of parallel composition: at each point in time one of the threads execute an action, corresponding to the program statement pointed by its control location. Fig. 1 shows a simple symmetric mutual exclusion protocol based on tickets. Program MutExc implements a mutual exclusion protocol that protects the critical section at line 5, using two global variables. The int variable $avail$ stores the shared increasing counter. The set variable bag stores the ticket number and the thread identifier of all threads that are trying to access the critical section. When a process wishes to enter the critical section (line 3), it grabs the current value of *avail* as its ticket and atomically performs the following two operations: (1) it increases the value of *avail*, and (2) it adds the value of its ticket and its thread id to the global storage bag. Then, the process waits (line 4) until it holds the smaller ticket in the storage bag. Upon exiting the critical section, the process removes its ticket from the bag (line 6).

We will illustrate in this paper how to formally prove using PVDs that if a process wants to enter the critical section, then it will eventually enter the critical section.

III. Model of Computation

In this section we introduce a general model of computation to reason about parametrized concurrent programs. We first revisit the notion of fair transition system. Then, we introduce parametrized programs and parametrized fair transition systems, which in turn provide the vocabulary to define parametrized temporal formulas. Finally, we define our notion of correctness of concurrent programs by associating parametrized fair transition systems with parametrized temporal formulas.

A. Fair Transition Systems

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

- *Signature*. The signature Σ_{prog} is a first-order signature modeling the data manipulated 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 from Σ_{prog}.
- *Program Variables*. V is a finite set of (typed) variables, whose types are taken from sorts in Σ_{prog}. We use V^t to denote all variables of sort t in set V.
- *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.
- *Transition Relation*. $\mathcal{T}$ is a finite set of transitions. Each transition τ is expressed as a first-order formula $\tau(V, V')$ that can refer to program variables from V (the set V' contains a fresh copy of v' of each variable v from V). The variable v' denotes the value of variable v after a transition is taken.
- *Fairness condition*. $\mathcal{J} \subseteq \mathcal{T}$ is the set of fair transitions.

A *state* is an interpretation of V, which assigns to each program variable a value of the corresponding type. We use S to denote the set of all possible states. A transition between two states s and s' satisfies a transition relation τ when the combined valuation (that assigns valuations to variables in V as s and to variables in V' as 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 $\tau(s, s')$. For example, the program statement:

```
1 :  x:=x + 1
```

is modeled by the following formula:

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

where we use $pres(U)$ as a short for $u' = u$ for all $u \in U$.

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

A *run* of $\mathcal{S}$ is an infinite sequence $s_0\tau_0 s_1\tau_1 s_2 \dots$ of states and transitions such that (a) the first state satisfies the initial condition: $s_0 \models \Theta$; (b) any two consecutive states s_i and s_{i+1} and the connecting transition τ_i satisfy $\tau_i(s_i, s_{i+1})$. We say that τ_i is taken at s_i, producing state s_{i+1}. A *computation* of $\mathcal{S}$ is a run of $\mathcal{S}$ such that for each transition $\tau \in \mathcal{J}$, if τ is continuously enabled after some point, then τ is taken infinitely many times. We use $\mathcal{L}(\mathcal{S})$ to denote the set of computations of $\mathcal{S}$. Given an LTL formula φ over a propositional vocabulary AP, $\mathcal{L}(\varphi)$ denotes the set of sequences of elements of 2^{AP} satisfying φ. Given a computation $\pi : s_0\tau_0 s_1 \dots$ of a system $\mathcal{S}$, the corresponding run π^{AP} for a given propositional vocabulary AP is the sequence $P_1P_2 \dots$ with $P_i \subseteq AP$, such that for all instants i:

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

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

B. Parametrized Concurrent Programs

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

- *Signature*. Σ_{prog}, as in FTSs.
- *Program Variables*. The set V of typed variables is:

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

  Note that "$v[k]$" is an indivisible variable name. Alternative names could have been v_k or vk. The set $\{pc[k] \mid k \in [M]\}$ contains one variable of sort loc for each thread id k in $[M]$. The variable $pc[k]$ stores the program counter of thread k. Similarly, for each local program variable v and thread k there is one variable $v[k]$ of the appropriate sort in the set $\{v[k] \mid v \in V_{local} \text{ and } k \in [M]\}$.
- *Initial Condition*. The initial condition Θ is described by two predicates Θ_g (that only refers to variables

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

Figure 2: Transition relations for thread 0 running program MUTEXC[2]

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

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

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

Example 1: Consider program MUTEXC in Fig. 1. The instance consisting of two running threads, MUTEXC[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 $\mathsf{set}\langle\mathsf{int}, \mathsf{tid}\rangle$. The instances of local variable $ticket$ for threads 0 and 1, $ticket[0]$ and $ticket[1]$, have type int. The program counters $pc[0]$ and $pc[1]$ have type $\mathsf{loc} = \{1 \ldots 7\}$. The initial condition of MUTEXC[2] is:

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

There are fourteen transitions in MUTEXC[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 shown in Fig. 2. The transitions for thread 1 are analogous. The predicate $pres$ summarizes the preservation of the values of variables. For example, in MUTEXC[2], the predicate $pres(V \setminus \{bag, pc[0]\})$ is simply:

$$
\begin{array}{lll}
avail' = avail & \wedge & ticket[0]' = ticket[0] \ \wedge \\
pc[1]' = pc[1] & \wedge & ticket[1]' = ticket[1]
\end{array}
$$

Note that each transition in MUTEXC[2] is quantifier free, and involves a combination of theories, including Presburger arithmetic and a theory of finite sets of pairs with non-repeating first component and minimum according also to the first component. ■

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

C. Parametrized FTS and Parametrized Formulas

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 and verification diagrams for parametrized systems. We describe each component separately:

- *Parametrized Program Signature*. To capture thread identifiers in an arbitrary instantiation of the parametrized system we introduce a new sort tid interpreted as

an unbounded discrete set. The signature Σ_{tid} contains only $=$ and $\neq$, and no constructor. Then, we extend the theory T_{prog}—used to reason about the data in the program— with the theory of arrays T_{A} from [20], with 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 [20] are trivially met. Variables of sort tid indexing arrays play a special role, so we classify formulas depending on the number of free variables of sort tid. The parametrized set of program variables with index variables X of sort tid is defined as:

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

  We use $F_{\mathsf{param}}(X)$ for the set of first-order formulas constructed using predicates and symbols from T_{param} and variables from $V_{\mathsf{param}}(X)$. Given a formula φ from $F_{\mathsf{param}}(X)$ we use $Var(\varphi)$ to refer to the set of variables of type tid 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 of thread identifiers X 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 MUTEXC. The parametrized transition $\tau_4^{(k)}$, for thread k in line 4, is the following formula from $F_{\mathsf{param}}(\{k\})$:

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

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 MUTEXC for two thread ids i and j is the formula $\Theta_{\mathsf{param}}(\{i, j\})$:

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

■

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

We are interested in verifying temporal properties of parametrized programs, so we extend parametrized formulas to *temporal parametrized formulas*, by taking predicates from $F_{\mathsf{param}}(\{k_0 \ldots, k_n\})$ and combining them using temporal operators from LTL ($\bigcirc$, $\mathcal{U}$, $\Box$, etc). For example, the following formula (a 2-index safety formula) expresses mutual exclusion for MUTEXC:

$$\Box i \neq j \rightarrow \neg(pc(i) = 6 \wedge pc(j) = 6)$$

Progress of each individual thread is expressed by the following 1-index temporal formula:

$$\Box \left(pc(i) = 3 \rightarrow \Diamond pc(i) = 6\right)$$

D. Parametrized Temporal Verification

In order to define the parametrized temporal verification problem we need to introduce an auxiliary notion. Let $\mathcal{S}$ be a parametrized FTS, $\varphi(\overline{k})$ a parametrized temporal formula, and M be a parameter value (a finite collection of thread ids). A concretization is a map $\alpha : \overline{k} \rightarrow [M]$. This map can be extended to formulas in the usual manner (by assigning $v[\alpha(i)]$ to $a_v(i)$) and extending to Boolean and temporal connectives. In this manner, elementary propositions from

the parametrized formula φ are in T_{param} but the corresponding elementary propositions of the concrete $\alpha(\varphi)$ are in T_{prog} using the variables of the concrete system $\mathcal{S}[M]$. For example, the concretization of mutual exclusion

$$\Box i \neq j \rightarrow \neg(pc(i) = 6 \wedge pc(j) = 6)$$

for MUTEXC[2] and $\alpha : \{i \rightarrow 0, j \rightarrow 1\}$ is

$$\Box\neg(pc[0] = 6 \wedge pc[1] = 6)$$

The concretization for $\alpha : \{i \rightarrow 0, j \rightarrow 0\}$ is $\Box$T. We are now ready to define the *parametrized temporal verification problem*.

Definition 1: Given a parametrized system $\mathcal{S}$ and parametrized temporal formula $\varphi(\overline{k})$ we say that $\mathcal{S} \vDash \varphi(\overline{k})$ whenever for all concrete instances $\mathcal{S}[M]$ and concretizations α, $\mathcal{S}[M] \vDash \alpha(\varphi(\overline{k}))$.

IV. PARAMETRIZED VERIFICATION DIAGRAMS

We introduce parametrized verification diagrams in this section as an effective method to solve the parametrized temporal verification problem. The aim of PVDs is to capture formally the proof that *all instances* of a parametrized program satisfy a temporal specification. Essentially, for each value of M, the diagram over-approximates the set of runs of $\mathcal{S}[M]$, while in turn being covered by the executions allowed by the temporal formula.

A. Definition of PVD

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

- N is a finite set of nodes.
- $N_0 \subseteq N$ is the subset of initial notes.
- $\mathcal{B}$ is a finite collection of pairs $\{(\mathcal{B}_1, b_1), \ldots, (\mathcal{B}_q, b_q)\}$, where $\mathcal{B}_i \subseteq N$ are disjoint set of nodes ($\mathcal{B}_i \cap \mathcal{B}_j = \emptyset$ for $i \neq j$), and each b_i is a fresh tid variable. Each pair $(\mathcal{B}_i, b_i)$ is called a box and the set $V_{box} = \{b_1, \ldots, b_q\}$ is called the set of box variables. We use V_{tid} for $\overline{k} \cup V_{box}$.
- E is a finite set of edges each connecting two nodes. Edges are equipped with the following functions and predicates:
  - *in* : $E \rightarrow N$ and *out* : $E \rightarrow N$ to indicate the incoming and outgoing node. We use $n \rightarrow_e m$ for an edge $e \in E$ with $in(e) = n$ and $out(e) = m$.
  - *within* $\subseteq E$, with the restriction that for all $e \in$ *within*, there is a box $\mathcal{B}_i$ with $in(e), out(e) \in \mathcal{B}_i$. This predicate indicates whether a transition modeled by e that connects two nodes within the same box must preserve the box variable, if $e \in$ *within*, or can change the box variable arbitrarily.
- μ is a labeling function for nodes which assigns to each node n a formula $\mu(n)$ in the theory $\mathcal{F}_T(V_{tid})$, with the restriction that $\mu(n)$ can only contain b_i whenever node n is in box $\mathcal{B}_i$.
- $\eta : E \rightharpoonup \mathcal{T} \times V_{tid}$ is a partial function labeling some edges with transitions to indicate that these edges label fair transitions (transitions that must be taken because of fairness).
- $\mathcal{F}$ is the acceptance condition of the diagram, described by a finite collection $\langle\langle B_1, G_1, \delta_1\rangle \ldots \langle B_m, G_m, \delta_m\rangle\rangle$. Each triplet acceptance condition $\langle B_i, G_i, \delta_i\rangle$ is formed by an edge Streett condition $B_i, G_1 \subseteq E$ and a ranking function $\delta : N \rightarrow \mathcal{O}$, where $\mathcal{O}$ is a well founded domain. Without loss of generality we can assume $G_i \cap B_i = \emptyset$. Edges in G_i are called *good* edges, and edges in B_i are called *bad* edges.
- f is a map from nodes into Boolean combinations of elementary propositions from $\varphi(k)$.

The following are restriction of node and edge labelings:

1) for each $n \in N$, b_i is not free in $\mu(n)$ unless $n \in \mathcal{B}_i$.
2) if $\eta(e) = (\tau, b_i)$ then $in(e) \in B_i$.

Restriction 1) establishes that the labeling of a node n that belongs to box (B_i, b_i) can use $\overline{k}$ and b_i as free variables. Restriction 2) indicates that the labeling of edges e for which $in(e)$ belongs to box (B_i, b_i), can be transitions of the form $\tau(k)$ for a $k \in \overline{k}$, or $\tau(b_i)$. Boxes can be understood as a compact representation of a section of a computation for all possible thread values. Conceptually, if a parameter instance M is fixed, every box can be populated M times, assigning in each expansion one of the possible tid values to the box variable that occurs in nodes and edges within the box. The resulting diagram is a classical non-parametrized GVD.

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

A *path* in the diagram is a sequence of states and edges $n_0e_0n_1e_1\ldots$ such that for every i, $n_i \rightarrow_{e_i} n_{i+1}$. A path is *fair* whenever if after some point i all nodes n_i have an outgoing edge labeled with (τ, v) then edges labeled (τ, v) are taken infinitely often. A path is *accepting* whenever for every acceptance condition (B_i, G_i, δ_i) either all edges from B_i are traversed finitely often, or some edge from G_i is traversed infinitely often.

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

Given a run $\pi : s_0\tau_0s_1\tau_1\ldots$ of a concrete instance $\mathcal{S}[M]$ and a concretization $\alpha : \overline{k} \rightarrow M$, a path $d = n_0e_0n_1e_1\ldots$ of $\mathcal{D}$ is a *trail* of π whenever for some extended concretization $\{\alpha_i\}$, the following holds: $s_i \vDash \alpha_i(\mu(n_i))$ for all $i \geq 0$. A run π is a *computation* of $\mathcal{D}$ if there exists a trail of π that is fair and accepting. $\mathcal{L}^{[M]}(\mathcal{D})$ denotes the set of computations

of $\mathcal{D}$ for parameter instance M (i.e., sequences of states of $\mathcal{S}[M]$ accepted by $\mathcal{D}$). In the next subsection we will list a collection of verification conditions extracted from the diagram, and we will show that proving the validity of these verification conditions implies that all computations of $\mathcal{S}[M]$ are in $\mathcal{L}^{[M]}(\mathcal{D})$.

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

B. Verification Conditions

A PVD shows that $\mathcal{S}[M] \vDash \varphi(\overline{k})$ via the inclusions $\mathcal{L}(\mathcal{S}[M]) \subseteq \mathcal{L}^{[M]}(\mathcal{D})$ and $\mathcal{L}_p^{[M]}(\mathcal{D}) \subseteq \mathcal{L}^{[M]}(\varphi(k))$. Theorem 1 below shows that to prove $\mathcal{L}(\mathcal{S}[M]) \subseteq \mathcal{L}^{[M]}(\mathcal{D})$ it is enough to prove the verification conditions presented in Fig. 3.

The main difficulty is to define a *finite* number of verification conditions that guarantee the previous language inclusion. A key notion is that of a formula vocabulary, the set of free variables of type tid appearing in a formula. Formally:

$$
\begin{array}{lcl}
Voc(c) & = & \begin{cases}\{c\} & \text{if } c \in C^{\mathsf{tid}} \\ \emptyset & \text{otherwise}\end{cases} \\
Voc(pc(k)) & = & \{k\} \\
Voc(v) & = & \begin{cases}\{v\} & \text{if } v \in V^{\mathsf{tid}}_{global} \\ \emptyset & \text{otherwise}\end{cases} \\
Voc(v(k)) & = & \begin{cases}\{v(k),k\} & \text{if } V^{\mathsf{tid}}_{local} \\ \{k\} & \text{otherwise}\end{cases} \\
Voc(\varphi_1 \bowtie \varphi_2) & = & Voc(\varphi_1) \cup Voc(\varphi_2) \\
Voc(\triangleright\varphi) & = & Voc(\varphi)
\end{array}
$$

where $\bowtie$ represents any binary operators like $\wedge$, $\vee$, $\rightarrow$, $\mathcal{U}$ or $\mathcal{W}$, and $\triangleright$ denotes any unary operator such as $\neg$, $\bigcirc$, $\Box$, $\Diamond$, etc. We use $Voc(x_1,\ldots,x_n)$ to denote $\bigcup_{i=1}^{n} Voc(x_i)$. Note that the vocabulary represents the set of variables of type tid whose modification can potentially alter the truth value of a given formula. We let the vocabulary of a node be: $NVoc(n) = \{b_i \mid n \in \mathcal{B}_i\} \cup \overline{k}$. Given a node n, let $next(n) = \{n' \in N \mid \text{for some } n \to_e n' \in E\}$.

Given a parametrized transition system $\mathcal{S}[M]$, a parametrized temporal formula $\varphi(\overline{k})$ and a parametrized verification diagram $\mathcal{D}$, Fig. 3 presents the verification conditions generated from the diagram. Given an edge $e \in E$, with $in(e), out(e) \in \mathcal{B}_i$, $\beta(e)$ is the formula $(b_i' = b_i)$ if $e \in$ *within* and the formula *true* otherwise. Also, $\tau(i)$ is the formula obtained from the transition relation τ by replacing all occurrences of local variables $v[i]$ by parameters $v(i)$, and all occurrences of $v[i]'$ by $v'(i)$.

Condition (Init), called initiation, says that at least one initial node in N_0 satisfies the initial condition of $\mathcal{S}$. Condition (SelfConsec), called self-consecution, establishes that any τ-successor of a state satisfying $\mu(n)$ satisfies the label of some successor node of n. In other words, the diagram can always move when taking any transition by any thread mentioned in the property. Condition (OtherConsec), others-consecution, is analogous, but considers transitions taken by an arbitrary thread not considered in the vocabulary of $\varphi(\overline{k})$ nor as argument of any of the boxes in the diagram. This condition is the key to guaranteed that only a finite number of verification conditions is necessary, because this condition encompasses all other threads not mentioned in the formula (or in boxes). Conditions (SelfAcc) and (OtherAcc), called self-acceptance and others-acceptance resp., guarantee the acceptance condition of the diagram though the verification of ranking functions. Intuitively speaking, these VCs use information extracted from the data in the system to infer that certain sequences of states must be terminating. For example, this is the manner in which one checks that at most a finite number of threads can out-run a given thread when entering the critical section. These conditions verify that the ranking function δ_i is (strictly) decreasing in B_i edges, and non-increasing in edges $E - (G_i \cup B_i)$. We use P_i to denote edges in $E - (G_i \cup B_i)$, called *permitted edges*. If the verification conditions for δ are valid, infinite trails either traverse G_i edges infinitely often, or traverse edges in P_i and B_i infinitely often. However, in this second case bad edges cannot be seen infinitely often because (1) the domain is well-founded, (2) permitted edges are non-increasing, and (3) bad edges are decreasing. Condition (En) establishes that any transition labeling an edge coming out from a node must be enabled at every state modeled by the node. Condition (Succ) establishes that if a transition labeling an edge is taken at the incoming node, then all edge labels cover the possible actions of the transitions. The combination of (En) and (Succ) guarantee that a label τ is always enabled at the given nodes and that the only way to exit the nodes taking τ is through the label edges. This relates fairness in any concrete system with fairness in the diagram.

Finally, condition (Prop) guarantees the correctness of the propositional models of the diagram. The propositional label f allows to use a single query to a finite state model-checker to show that propositional models of the diagram are included in traces of the property, in (ModelCheck).

For a parametrized system $\mathcal{S}[M]$, a formula $\varphi(\overline{k})$ and a PVD $\mathcal{D}$, if all verification conditions described above hold

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

Initiation:

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

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

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

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

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

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

$$\big(\, \mu(n) \wedge \tau(i) \wedge \mu'(m) \wedge \beta(e) \,\big) \rightarrow \delta(n) \succ \delta(m) \quad \text{if } e \in B$$
$$\big(\, \mu(n) \wedge \tau(i) \wedge \mu'(m) \wedge \beta(e) \,\big) \rightarrow \delta(n) \succeq \delta(m) \quad \text{if } e \in E \setminus (G \cup B)$$

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

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

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

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

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

Satisfaction:

(Prop) $\quad \mu(n) \rightarrow f(n)$ for all $n \in N$

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

Figure 3: Verification conditions for parametrized verification diagrams.

we say that $\mathcal{D}$ is $(\mathcal{S}, \varphi)$-valid.

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

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

Proof: We start by assuming that there is a $(\mathcal{S}, \varphi)-valid$ PVD $\mathcal{D}$, and show that $\mathcal{S} \vDash \varphi$. This requires showing that for an arbitrary M and concretization $\alpha : \overline{k} \rightarrow [M]$, $\mathcal{S}[M] \vDash \alpha(\varphi(\overline{k}))$. We will use repeatedly the following result from [21]: let $\psi(\overline{k})$ be a parametrized (non-temporal) formula and α a concretization. Then, if $\psi(\overline{k})$ is valid, so is $\alpha(\psi(\overline{k}))$.

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

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

First, we show by induction that there is a path $\pi : n_0e_0n_1e_1n_2\ldots$ of σ in the diagram, and a sequence of thread identifiers $j_0j_1\ldots$ such that $\alpha(j_k) = i_k$ and $s_i \vDash \alpha(\mu(n_i))$. It is enough to prove that there is a trail of nodes n_k of the diagram and a extended concretization α_k such that (1) $s_k \vDash \alpha_k(\mu(n_k))$ and (2) $\tau_k^{j_k}$ can be taken to traverse edge e_k (that is, $\neg(\mu(n_k) \wedge \tau_k^{j_k} \wedge \beta(e_k) \rightarrow \mu'(n_{k+1}))$ is not valid). We build the trace by induction:

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

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

    is valid, so the following is also valid

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

    or equivalently

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

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

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

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

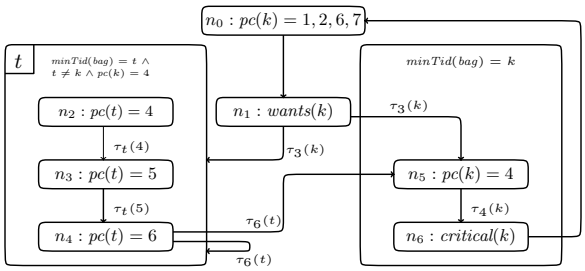

Figure 4: Parametrized verification diagram for MUTEXC $\vDash \Box(wants(k) \to \Diamond critical(k))$.

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

■

V. PROGRESS OF A MUTUAL EXCLUSION ALGORITHM

We illustrate PVDs by showing a diagram for a response property of the simple mutual exclusion algorithm MUTEXC shown in Fig. 1.

We use $wants(k)$ for $(pc(k) = 3)$, and $critical(k)$ for $pc(k) = 5, 6$. We show now how to verify a simple liveness property: every thread that wants to enter the critical section eventually does, formally expressed in LTL using the following response property [11]:

$$\psi(k) \mathrel{\hat{=}} \Box(wants(k) \to \Diamond critical(k))$$

To verify this example, we use the theory of finite sets of pairs of integers with ordered comprehension and minimum value. In this theory, the function $lower(s, n)$ receives a set of pairs s and an integer n, and returns the subset of pairs whose first component is strictly lower than n. Additionally, this theory also provides a function that returns the lowest value in a set of pairs, for each of the components. We show now a parameterized verification diagram that represents the desired proof. The diagram is depicted in Fig. 4 and it is formally defined by:

$$\begin{aligned}
N &\mathrel{\hat{=}} \{n_i \mid 0 \le i \le 6\} \\
N_0 &\mathrel{\hat{=}} \{n_0\} \\
E &\mathrel{\hat{=}} \{n_0 \to n_1, n_5 \to n_6, n_6 \to n_0\} \quad \cup \\
&\quad \{n_j \to n_i \mid j = 1, 4 \text{ and } i = 2, 3, 4, 5\} \cup \\
&\quad \{n_2 \to n_3, n_3 \to n_4\} \\
\textit{within} &\mathrel{\hat{=}} \{n_2 \to n_3, n_3 \to n_4\} \\
\mathcal{B} &\mathrel{\hat{=}} \{(\mathcal{B}_1, t)\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F} \mathrel{\hat{=}}\ & \langle\langle \{n_4 \to n_i \mid i = 2, 3, 4, 5\}, \{n_6 \to n_0\}, \\
& \lambda n \to lower(bag, ticket(k))\rangle\rangle
\end{aligned}$$

The value of the ranking function is the subset of tickets lower than the ticket of k. This set decreases (with respect to $\subseteq$) every time the leader thread (whoever that is, captured by the box variable) exits the critical section. The map f labels node n_1 into *wants*(k) and node n_6 with *critical*(k) and all the other nodes to T. Each node in the diagram contains self-loop edges for all transitions which are not labeling any other outgoing edge from such node. For example, for node n_4 there exists an (implicit) edge $n_4 \rightarrow n_4$ for all transitions other than $\tau_6(t)$. In the diagram, function *minTid*(s) returns the thread identifier (second component) in the pair considered as the minimun in set s following the order provided by the first component of the pairs.

VI. Conclusion

This paper has introduced parametrized verification diagrams, an extension of verification diagrams that allow to prove temporal properties of concurrent systems with an unbounded number of processes.

PVDs enable to encode in a single proof an evidence that all instances of the parametrized system satisfy a given temporal specification. This evidence can be automatically checked solving a finite-state model checking problem, and proving a finite number of verification conditions, generated automatically from the program and the diagram. Decision procedures for the underlying theories of the data-types in the program allow to handle this VCs automatically as well.

Ongoing work includes an implementation of PVDs in the theorem prover for parametrized systems Leap—under development at the IMDEA Software Institute [1]—and their use in verifying various concurrent protocols and datatypes.

Future work includes studying the completeness for PVDs, and relaxations of the symmetry requirement for which parametrized diagrams can also be used.

References

[1] A. Browne, Z. Manna, and H. B. Sipma, "Generalized temporal verification diagrams," in *FSTTCS'95*, ser. LNCS, vol. 1206. Springer, 1995, pp. 484–498.

[2] H. B. Sipma, "Diagram-based verification of discrete, real-time and hybrid systems," Ph.D. dissertation, Stanford University, 1999.

[3] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan-Kaufmann, 2008.

[4] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *LICS'02*. IEEE Computer Society Press, 2002, pp. 55–74.

[5] G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani, "A logic of reachable patterns in linked data-structures," in *FOSSACS'06*, 2006, pp. 94–110.

[1]The current version of Leap can be downloaded from *http://software.imdea.org/leap*.

[6] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu, "A logic-based framework for reasoning about composite data structures," in *CONCUR'09*, ser. LNCS, vol. 5710. Springer, 2009, pp. 178–195.

[7] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro, "Proving correctness of highly-concurrent linearisable objects," in *PPOPP'06*, 2006, pp. 129–136.

[8] A. Hobor, A. W. Appel, and F. Z. Nardelli, "Oracle semantics for concurrent separation logic," in *ESOP'08*, ser. LNCS, vol. 4960. Springer, 2008, pp. 353–367.

[9] P. W. O'Hearn, "Resources, concurrency and local reasoning," in *CONCUR'04*, ser. LNCS, vol. 3170. Springer, 2004, pp. 49–67.

[10] S. Brookes, "A semantics for concurrent separation logic," in *CONCUR'04*, vol. 3170. Springer, 2004, pp. 16–34.

[11] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems*. Springer, 1995.

[12] A. Sánchez and C. Sánchez, "Decision procedures for the temporal verification of concurrent lists," in *ICFEM'10*, ser. LNCS, vol. 6447. Springer, 2010, pp. 74–89.

[13] ——, "A theory of skiplists with applications to the verification of concurrent datatypes," in *NFM'11*, ser. LNCS, vol. 6617. Springer, 2011, pp. 343–358.

[14] Z. Manna and H. Sipma, "Verification of parameterized systems by dynamic induction on diagrams," in *CAV'99*, ser. LNCS, vol. 1633. Springer, 1999.

[15] N. Bjørner, A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H. Sipma, and T. E. Uribe, "Verifying temporal properties of reactive systems: A STeP tutorial," *FMSD*, vol. 16, no. 3, pp. 227–270, 2000.

[16] E. M. Clarke, M. Talupur, and H. Veith, "Proving Ptolemy right: The environment abstraction framework for model checking concurrent systems," in *TACAS'08*, ser. LNCS, vol. 4963. Springer, 2008, pp. 33–47.

[17] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv, "Thread quantification for concurrent shape analysis," in *CAV'08*, ser. LNCS, vol. 5123. Springer, 2008, pp. 399–413.

[18] E. A. Emerson and V. Kahlon, "Model checking large-scale and parameterized resource allocation systems," in *TACAS*, ser. LNCS, vol. 2280. Springer, 2002, pp. 251–265.

[19] L. Groves, "Verifying Michael and Scott's lock-free queue algorithm using trace reduction," in *CATS*, ser. CRPIT, vol. 77. Australian Computer Society, 2008, pp. 133–142.

[20] A. R. Bradley, Z. Manna, and H. B. Sipma., "What's decidable about arrays?" in *Proc. of VMCAI'06*, ser. LNCS, vol. 3855. Springer, 2006, pp. 427–442.

[21] A. Sánchez and C. Sánchez, "Parametrized invariance for infinite state processes," *CoRR*, vol. abs/1312.4043, 2013.