Parameterized Model Checking by enhancing the SPIN checker

SHANSHAN LIU, ANKIT GOEL, ABHIK ROYCHOUDHURY
School of Computing
National University of Singapore

Parameterized systems are characterized by the presence of a large (or even unbounded) number of behaviorally similar processes, and they often appear in distributed controllers and protocols. Verification of parameterized systems involves reasoning about unboundedly many processes and hence cannot be accomplished directly by model checking. In this work, we develop an abstraction refinement based verification framework for parameterized systems. We enhance the well-known SPIN model checker with process count abstractions to develop a time and memory efficient Linear-time Temporal Logic (LTL) model checker for parameterized systems. We also develop methods for automated detection of spurious counter-examples, and their elimination via abstraction refinement. The usability / scalability of our checker is demonstrated via the modeling and automated verification of several real-life parameterized control systems and protocols.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Model Checking*

General Terms: Verification

Additional Key Words and Phrases: SPIN Model Checker, Abstraction Refinement

1. INTRODUCTION

Many distributed control systems contain a large number of concurrently running, behaviorally similar processes interacting among themselves, as well as with other processes. Examples of such control systems are common in automotive, avionics and other application domains. As an example consider a centralized controller disseminating traffic information to various flights in a flight control system. Or, consider a centralized registry enabling the communication among various devices (such as CD player, TV, GPS) in a modern car. The behaviorally similar processes in these systems are generally of the same process-type (e.g. the process-type of flights in a flight-control system). Furthermore, the number of processes in a process type is often unbounded when the system is designed, and fixed later when the system is deployed. For example, at design time we do not know how many incoming/outgoing flights a flight control system may need to deal with when it is deployed. It is necessary to verify crucial properties of the control system with unbounded number of flights, such that the verification results hold for any number of flights.

Authors' e-mails: {liushans,ankit,abhik}@comp.nus.edu.sg Address of the authors: School of Computing, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417.

Concurrent systems with unboundedly many behaviorally similar processes of the same process type are known as *parameterized systems*. Parameterized systems typically consist of finitely many process types – the behavior of each process type being summarized by a finite-state transition system. The number of processes for (some of) the process types is unbounded — denoted as ω. Verification of parameterized systems is beyond the realm of model checking since we need to reason about unboundedly many processes. Note that it is unnatural to fix the number of processes to an arbitrary constant, just for making their verification amenable to model checking. This is because it is in general difficult (not decidable) to compute a cutoff number on the number of processes such that the restricted system will exhibit all behaviors of the system with unbounded number of processes.

Most existing works on parameterized system verification have focused on developing (more) automated checking procedures. However, researchers have ignored the issues in developing a usable and efficient checker which can be used for debugging real-life parameterized control software, namely: (a) being supported by a powerful modeling language to describe non-trivial control systems and protocols, (b) time/space efficiency in checking, (c) spurious counter-example detection (the spurious behaviors arising from abstraction), along with abstraction-refinement, and (d) analysis of non-spurious counter-examples to determine a finite-state system exhibiting the same trace. In this paper, we address these issues.

We develop an abstraction-refinement based automated proof method for parameterized systems. Our abstractions (and their refinements) deal with the number of processes to keep track of, for process types in the system with unbounded number of processes. Thus, for every process type with unbounded number of processes — a *cutoff number* is assumed in the initial abstraction and then gradually refined by repeated application of abstract - modelcheck - refine steps. Since our abstractions in general lead to over-approximations of behavior, model checking of the abstracted system may lead to spurious counter-examples. We develop automated methods to (i) check whether a given counter-example trace is spurious, and (ii) refine our abstraction (by increasing cutoff numbers) to eliminate a given spurious counter-example. For non-spurious traces (traces which point to real errors), we develop heuristics to determine a system with a small, finite number of processes which exhibit the same counter-example trace. *Our method can be used for verifying Linear-time Temporal Logic (LTL) properties of parameterized systems, subject to few restrictions; these restrictions appear in Section 8.1.*

Similar to works on abstraction refinement based sequential software verification [Chaki et al. 2003; Beyer et al. 2007; Ball and Rajamani 2002], our proof method follows an *abstract - verify - refine* loop which is iterated. While those approaches deal with abstraction of program variables from unbounded data domains, our abstractions deal with unbounded number of processes. Our spurious counter-example check and refinement step is fully automated, thereby making our proof method *fully automated*. Since parameterized system verification is undecidable [Apt and Kozen 1986], our verification procedure may not terminate in general. However, this is not unexpected given the undecidability of the problem — existing works on abstraction refinement based software verification (such as [Chaki et al. 2003; Beyer et al. 2007; Ball and Rajamani 2002]) also cannot guarantee termination of the abstract-verify-refine loop. In our experiments on several real-life control systems and protocols, our verification procedure not only terminates, but also is highly efficient in terms of time and memory.

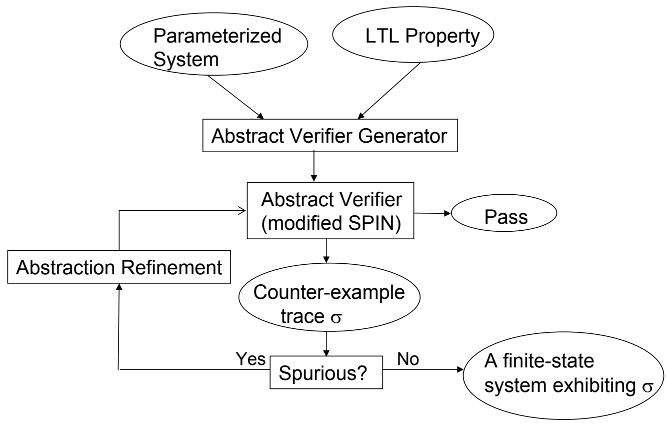

Fig. 1. Verification framework

In terms of implementation, we modify the internals of the well-known SPIN model checker [Holzmann 2003; 1997] to integrate our proof method. Any designer familiar with SPIN, and its input language PROMELA, can use our method as a black-box to verify parameterized systems. *We take advantage of powerful optimizations inside* SPIN *to develop an efficient checker for parameterized systems.*

Contributions. The contributions of this paper are as follows.

—We develop a fully automated methodology for parameterized system verification via abstraction refinement. Given a LTL property φ, our proof method gradually discovers the cutoff numbers needed to prove φ for each process type in the system with unbounded number of processes.

—As far as system modeling is concerned, we tie up with the rich PROMELA modeling language used in SPIN. Our goal here is pragmatic — since PROMELA/SPIN are widely used for system modeling/verification, this immediately increases the potential usability of our method. Indeed, we implement our proof method inside the verification engine of SPIN. Thus, any user familiar with PROMELA/SPIN can straightaway use our proof method for parameterized system verification. Moreover, all of the state-space optimizations already built inside SPIN enhance the efficiency of our checker.

—Last but not the least, experiments with our parameterized system checker implemented inside SPIN demonstrate its scalability in terms of time and memory. We have experimented our checker on large-scale parameterized control systems from the automotive/avionics domain as well as on well-known bus protocols (such as Futurebus+). We have also made our tool available for usage in research / teaching. The tool and its description can be obtained from `http://www.comp.nus.edu.sg/~abhik/SPIN++/`

In summary, we develop an abstraction refinement based automated proof method for parameterized systems, and integrate it with the mature SPIN model checking tool. This also enables rich modeling of parameterized systems using PROMELA, and experiments on large-scale parameterized control systems.

2. OVERVIEW

An outline of our verification framework appears in Figure 1. The verification procedure involves deriving an abstract verifier (based on model checker SPIN) corresponding to a given system model and property to be verified. Parameterized verification of the system proceeds by executing the abstract verifier thus generated. At the end of a verification run, either the verifier outputs "pass"– indicating no property violation in the system model, or a counter-example exhibiting the property violation. Since, in general our abstraction is an over-approximation of concrete behaviors, a counter-example obtained from the abstract verifier can be spurious. Thus, a spuriousness check is performed on the counter-example obtained. If a counter-example is not spurious, a system with finite number of processes exhibiting the same counterexample is generated. Otherwise, we refine our abstraction to prevent the spurious counter-example from occurring in the subsequent verification runs. Since parameterized system verification is undecidable [Apt and Kozen 1986], the abstraction-refinement loop shown in Figure 1 is not guaranteed to terminate. Hence, user may specify a bound on the number of refinement steps undertaken.

We now illustrate the various steps in our verification framework with the help of a small example. Consider a system model consisting of a single process type p_1 with no local variables. The transition system corresponding to process type p_1 is shown in Figure 2, where for $i \in [0,3]$, l_i represents a control location, and α_0 – α_2 represent the actions executed by a p_1-process. Thus, for example, a process of type p_1 can move from location l_0 to l_1 by executing action α_0. Assume now, that we want to verify certain properties for this system model for any number of p_1 processes. Let us consider an unbounded number of p_1 processes which are initially in the state l_0.

In our abstract verification we only maintain the count of processes in various local states, and not their individual states or identities. If the process count is unbounded in some state, it is represented as ω during abstract verification. Further, a user-provided p_1 specific cutoff parameter (called cut_{p_1}) is used, such that ω represents *greater-than or equal-to* cut_{p_1} p_1-processes. Then, (a) if a p_1 process moves in to a state with cut_{p_1} number of processes, the process count of that state becomes ω, *and* (b) if a p_1 process moves out of a state with currently ω number of processes, there remains either ω or cut_{p_1} number of processes in the source state.

We now consider verification of the given system against the following LTL property: $\neg(\alpha_0 \wedge X\alpha_1 \wedge XX\alpha_2)$. It specifies that the action sequence $\sigma = \alpha_0\alpha_1\alpha_2$ can never occur in a system execution. Initially, let $cut_{p_1} = 1$. This means in the abstract verification, the count of processes in any state l_i is either 0 (denoting no processes in l_i) or ω (denoting one or more processes in l_i). Abstract verification returns a counter-example trace, which is σ itself. The number of processes in different states during abstract execution of σ are shown in the following.

Control state	Number of processes ($cut_{p_1} = 1$)			
	Initially	After α_0	After α_1	After α_2
l_0	ω	$\omega, 0$	$\omega, 0$	$\omega, 0$
l_1	0	ω	$\omega, 0$	$\omega, 0$
l_2	0	0	ω	ω
l_3	0	0	0	ω

However, it is easy to see that the counter-example trace σ cannot be exhibited in any

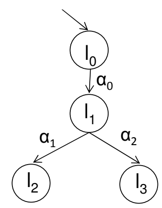

Fig. 2. Example transition system for a process type p_1.

concrete system. At least two occurrences of α_0 are required for α_1 and α_2 to be executed subsequently. Hence, σ is spurious. Trace σ can be exhibited in the abstract system because after a single occurrence of α_0, the process count in l_1 becomes ω (since $cut_{p_1} = 1$). Consequently, both α_1 and α_2 can be executed from l_1. In order to prevent this spurious counter-example, we refine our abstraction by increasing the cutoff number cut_{p1} to 2. This means in the abstract verification, the count of processes in any state l_i is now either 0 (denoting no processes in l_i) or 1 (denoting exactly one process in l_i) or ω (denoting more than one process in l_i). Now, after a single occurrence of α_0 the process count at l_1 will become 1 (and not ω), which is not sufficient to execute both α_1 and α_2. As a result, σ can no longer occur in abstract execution.

As another example, consider the LTL property $\neg(\alpha_0 \wedge X\alpha_1)$, which specifies that the action sequence $\sigma' = \alpha_0\alpha_1$ can never occur. Similar to the previous example, here also the abstract verification run returns a counter-example, which is σ'. However, unlike above, σ' is not spurious and can be exhibited in a concrete system. Further, we can easily see that σ' can be exhibited in a concrete system with only a single process of type p_1. Later, (in Section 6.2) we describe a heuristic procedure for deriving such a smaller concrete system corresponding to a non-spurious counter-example for debugging purposes.

3. MODELING

We use (a fragment of) PROMELA, the input language of SPIN, for modeling the system to be verified. This enables a user already familiar with SPIN, and hence PROMELA, to readily use our parameterized verification framework.

We fix a finite set of process types $\mathcal{P}$ with p, q ranging over $\mathcal{P}$. Each process type in $\mathcal{P}$ corresponds to a process declaration via `proctype` in PROMELA. Various processes, which are the instances of process types in $\mathcal{P}$, are described by means of finite-state labeled transition systems. We also fix a finite alphabet of *actions* Σ with α, β ranging over Σ. Actions in Σ represent basic PROMELA statements, such that an action in Σ may correspond to a send/receive event, an assignment, an assertion, or creation of an instance of a process type. Various processes can communicate via *synchronous* message exchange or through shared variables. With each action $\alpha \in \Sigma$, we associate– (a) a pre-condition Pre_α specifying a boolean condition to be satisfied by a system state for executing α, and (b) a post-condition $Post_\alpha$ capturing the system state update upon execution of α.

We note here that PROMELA allows inter-process communication via shared variables, synchronous message passing as well as asynchronous message passing. In our modeling, we restrict ourselves to systems which do not have asynchronous message passing.

In order to model the internal states and computations performed by processes, we fix a set of local variables Var_p for each process type $p \in \mathcal{P}$, and a set of global variables

Var_G. Assume that $Var_p \cap Var_q = \emptyset$ whenever $p, q \in \mathcal{P}$ and $p \neq q$. Also, let $Var_{\mathcal{P}} = \bigcup_{p\in\mathcal{P}} Var_p$ and $Var_{\mathcal{P}} \cap Var_G = \emptyset$. Each variable $x \in Var_{\mathcal{P}} \cup Var_G$ takes values from a *finite* domain $\mathcal{D}_x$. Thus, a *variable valuation* refers to a mapping of each variable to a value in its finite domain.

DEFINITION 1 SYSTEM MODEL. *A* **system model** *is a structure*

$$\mathcal{S} = (Var_G, v^g_{in}, TS_{\mathcal{P}})$$

consisting of (i) a set of global variables Var_G, (ii) their initial valuation v^g_{in} and (iii) a p-indexed family of transition systems

$$TS_{\mathcal{P}} = \{TS_p = (L_p, \rightarrow_p, l^p_{in}, Var_p, v^p_{in})\}_{p\in\mathcal{P}}$$

such that for each $p \in \mathcal{P}$,

—*L_p is a finite set of p's control states,*

—*$\rightarrow_p \subseteq L_p \times \Sigma \times L_p$ is a transition relation for p,*

—*$l^p_{in} \in L_p$ is the initial control state of p, and*

—*Var_p is the set of local variables in p, and v^p_{in} is their initial valuation.*

Consider the example in Figure 2, which consists of a single process type p_1 with no local variables. The actions appearing in this specification are $\Sigma = \{\alpha_0, \alpha_1, \alpha_2\}$, and the transition system of p_1 is represented as:

$$TS_{p_1} = (\{l_0, l_1, l_2, l_3\}, \rightarrow_{p_1}, l_0, \emptyset)$$

where $\rightarrow_{p_1} = \{(l_0, \alpha_0, l_1), (l_1, \alpha_1, l_2), (l_1, \alpha_2, l_3)\}$.

Let *OBJ*$_p$ denote a finite non-empty set of processes populating process type p. We require that *OBJ*$_p \cap$ *OBJ*$_q = \emptyset$ whenever $p \neq q$. We set *OBJ* $= \bigcup_{p\in\mathcal{P}}$ *OBJ*$_p$ and let o, o' range over *OBJ*. Further, each variable x in the system has a finite domain D_x. Hence, we let Val_G be a mapping for each global variable to its finite domain, and Val_p be a mapping for each variable in process type p to its finite domain. We denote $Val_{\mathcal{P}} = \bigcup_{p\in\mathcal{P}} Val_p$. For each process type p, let $S_p \subseteq L_p \times Val_p$ represent the execution states of p, where L_p is the set of p's control states describing TS_p and Val_p is the set of valuations of variables in Var_p. The initial p-state is given by $s^p_{in} = (l^p_{in}, v^p_{in})$, where $l^p_{in} \in L_p$ is the initial p-control state and v^p_{in} is an initial valuation of variables in p. We set $S_{\mathcal{P}} = \bigcup_{p\in\mathcal{P}} S_p$ and let s, s' range over $S_{\mathcal{P}}$. Again, consider the example shown in Figure 2. The initial state corresponding to process type $p1$ is $s^{p1}_{in} = (l_0, \epsilon)$, where the variable valuation part is empty (represented as ϵ) since $p1$ has no local variables.

In order to define the operational semantics of a system model, we define the notion of a *configuration* capturing the global system state during execution. Since we are defining a system configuration where the system consists of concrete processes, we call it a "concrete configuration". This is to distinguish this notion from the state space abstraction and the abstract configurations we will introduce later.

DEFINITION 2 CONCRETE CONFIGURATION. *Let $\mathcal{S} = (Var_G, v^g_{in}, TS_{\mathcal{P}})$ be a given system model. A* ***concrete configuration*** *of $\mathcal{S}$ is a pair of mappings (v_g, M), where v_g is a valuation of global variables Var_G, and mapping $M : S_{\mathcal{P}} \rightarrow 2^{OBJ}$ is defined such that:*

—*$M(s) \subseteq$ OBJ$_p$ for every p and every s in S_p,*

—*$M(s) \cap M(s') = \emptyset$ whenever $s \neq s'$, and*

—$\bigcup\{M(s) \mid s \in S_p\} =$ *OBJ*$_p$ *for every* p.

Let CFG denote the set of all concrete configurations.

Given a system model $\mathcal{S} = (Var_G, v^g_{in}, TS_{\mathcal{P}})$ and an initial set of processes *OBJ*$^{in}_p$ for each process type p, the *initial* configuration of $\mathcal{S}$ is defined as $C_{in} = (v^g_{in}, M^{in})$, where– (a) v^g_{in} is an initial valuation of global variables, and (b) for every $p \in \mathcal{P}$ and every $s \in S_p$, $M^{in}(s) =$ *OBJ*$^{in}_p$ if $s = s^p_{in}$, otherwise $M^{in}(s) = \emptyset$. For the example shown in Figure 2, suppose two instances (say, o_1, o_2) of process type $p1$ are created initially. Since, $p1$ has no local variables, all possible execution states of $p1$ are determined by its local control states, *i.e.* $S_{p1} = \{s_0, s_1, s_2, s_3\} = S_{\mathcal{P}}$, where $s_i = (l_i, \epsilon)$, with ϵ representing an empty variable valuation. Also, since there are no global variables in this example, the global variable valuations in this example will also be empty (or, ϵ). Then, the initial configuration in this case is given by: (ϵ, M^{in}), where $M^{in}(s_0) = \{o_1, o_2\}$ and $M^{in}(s_1) = M^{in}(s_2) = M^{in}(s_3) = \emptyset$.

During execution, system moves from one concrete configuration to another by participating in an action from Σ. If a process o of type p moves from state $s_1 \in S_p$ at concrete configuration $C = (v, M)$ to state $s_2 \in S_p$ by executing an action $\alpha \in \Sigma$, the processes at the resulting configuration $C' = (v', M')$ are determined as follows. Let $I : S_{\mathcal{P}} \to 2^{OBJ}$ be an intermediate mapping s.t.

—If $s_1 \neq s_2$, then
$I(s_1) = M(s_1) - \{o\}$.
$I(s_2) = M(s_2) \cup \{o\}$.
$I(s) = M(s)$ for $s \in S_{\mathcal{P}} \backslash \{s_1, s_2\}$.
—Otherwise, $\forall s \in S_{\mathcal{P}}, I(s) = M(s)$.

The relationship between the resulting mapping M' at configuration C' and the intermediate mapping I is as follows – (i) If α does not create new process, then $M' = I$, (ii) Otherwise, suppose by executing α, a new process o_q of type q is created and starts its execution from an execution state $s_q \in S_q$. Then, we have

—$M'(s_q) = I(s_q) \cup \{o_q\}$;
—$M'(s) = I(s)$, for all $s \neq s_q$.

We use relation $\mathbf{update_c}(s_1, M, \alpha, s_2, M')$ to denote that the mapping M' can be derived from M due to migration of a process from state s_1 to s_2 by executing α. The transition relation for the concrete execution $\hookrightarrow \subseteq$ *CFG* $\times\, \Sigma \,\times$ *CFG* is defined as follows.

DEFINITION 3 CONCRETE TRANSITION RELATION $\hookrightarrow$. *Let* $C = (v_g, M)$, $C' = (v'_g, M')$ $\in$ *CFG be concrete configurations of a system model* $\mathcal{S} = (Var_G, v^g_{in}, TS_{\mathcal{P}})$, *and* $\alpha \in \Sigma$ *be an action. Then* $(C, \alpha, C') \in \hookrightarrow$ *iff* $\exists p \in \mathcal{P}, \exists s = (l, v), s' = (l', v') \in S_p$, *s.t.*

(1) $(l, \alpha, l') \in \to_p$ *is a transition in* TS_p.

(2) $|M(s)| \geq 1$, *i.e. there is at least one process at state* s.

(3) v *and* v_g *satisfy the pre-condition* Pre_α.

(4) $v'(v'_g)$ *is the effect of post-condition* $Post_\alpha$ *on* $v(v_g)$. *Here* v'_g *represents an update of global variables* Var_G, *which can later be read/updated by other processes, thus allowing for* shared variable communication.

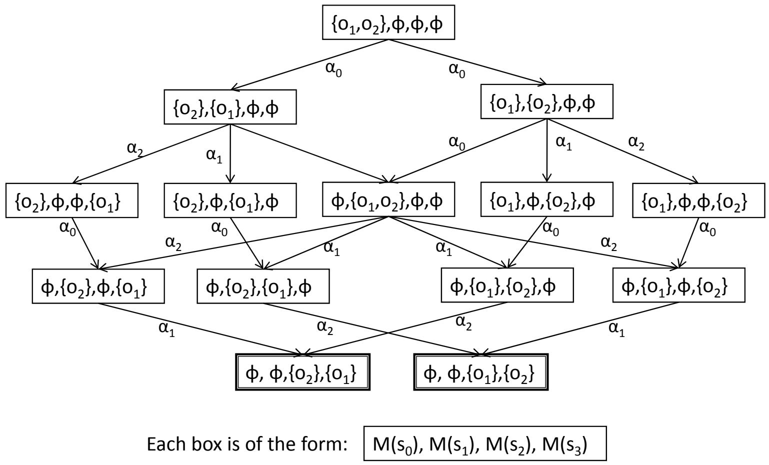

Fig. 3. Sample Concrete Transition Relation

(5) *If last action executed was a* send[1] *event, then α must be the corresponding receive event. If the matching receive cannot be executed, then the last executed send event is rolled back, and some other enabled action is executed in its place.*

(6) *The relation $update_c(s, M, \alpha, s', M')$ holds as described above.*

For the example shown earlier in Figure 2, we present its state exploration graph depicting all reachable concrete configurations in Figure 3. Since, no global variables are used in this example, we omit their valuation from a state representation. In each global state, the processes presented in various execution states of process type $p1$ are shown, which is the only process type appearing in this example. Also, for $i = 0, 1, 2, 3$, $s_i = (l_i, \epsilon)$, where l_i is the local control state of $p1$ and ϵ represents an empty local variable valuation. Initially, two processes of type $p1$ are created, which are represented as o_1 and o_2 residing at state $s_0 = (l_0, \epsilon)$. Then, either o_1 or o_2 can be chosen to execute α_0, following by the execution of α_0 from the other process, or the execution of either α_1 or α_2 from the same process, resulting in different paths in the state exploration graph ending in two configurations with o_1 residing at state s_2 and o_2 residing at state s_3, or vice versa.

4. STATE SPACE ABSTRACTION

4.1 Core Abstraction

For efficient verification of parameterized systems, we employ an abstract state space representation, where the core idea is to group together processes in a process type which are in *similar* states. However, the grouping of processes is not fixed statically, but changes dynamically with the state space construction. Two processes of type p are *similar* if and only if they are in the same state $s = (l, v) \in S_p$, where l is the control state in TS_p (the transition system of p) and v is a valuation of p's variables. Based on this, the key idea in our abstraction is that, if two processes are in the same execution state– there is *no need* to distinguish between them via their process ids. Hence, our abstraction systematically

[1]Recall that, we only consider synchronous message communication.

exploits this observation by only maintaining the count of processes in each execution state in $\bigcup_{p\in\mathcal{P}} S_p$.

Along with the state space abstraction as described above, we allow a process-type to have an *unbounded* number of processes in our abstract execution semantics. If a process type p initially has unbounded number of processes, or if p has unbounded number of processes due to dynamic process creation during execution – the user provides an input parameter $cut_p \in \mathbb{N}$. By default cut_p is set to 1. Then, for any number of processes equal to or greater than cut_p, we represent it as ω.

For a process type p with initially fixed number of processes, and no dynamic process creation – the process counts never become ω and the number of processes is fixed. Hence, the cutoff number is not an issue! We can simply assume the cutoff number to be a number greater than the number of p-processes by default.

Based on our abstract state representation, we now define the notion of an abstract configuration.

DEFINITION 4 ABSTRACT CONFIGURATION. *Let $\mathcal{S} = (Var_G, v_{in}^g, TS_{\mathcal{P}})$ be a given system model and for each process type $p \in \mathcal{P}$, N_p^a denote the number of p-processes during execution. An* **abstract configuration** *is defined as a pair of mappings (v_g^a, M_a), where $v_g^a \in Val_G$ is a valuation of global variables and $M_a : S_{\mathcal{P}} \to \mathbb{N} \cup \{\omega\}$ s.t. $\forall p \in \mathcal{P}, \sum_{s\in S_p} M_a(s) = N_p^a$.*

Let CFG$_{abs}$ denote the set of all abstract configurations.

Let $\mathcal{S} = (Var_G, v_{in}^g, TS_{\mathcal{P}})$ be a given system model with N_p number of processes of type $p \in \mathcal{P}$. Then, the *initial* abstract configuration of $\mathcal{S}$ is defined as $C_a^{in} = (v_g^{in}, M_a^{in})$, where– (a) v_g^{in} is the initial valuation of global variables, and (b) for every $p \in \mathcal{P}$ and every $s \in S_p$, $M_a^{in}(s) = N_p$ if $s = s_{in}^p$, otherwise $M_a^{in}(s) = 0$.

During execution, system moves from one abstract configuration to another by executing an action from Σ. If a process of type p moves from state $s_1 \in S_p$ at configuration $C_a = (v, M_a)$ to state $s_2 \in S_p$ by executing an action $\alpha \in \Sigma$, the process counts at a resulting configuration $C_a' = (v', M_a')$ are determined as follows. Let $I_a : S_{\mathcal{P}} \to \mathbb{N} \cup \{\omega\}$ be an intermediate mapping, s.t. –

—If $s_1 \neq s_2$, then

$$\begin{aligned} I_a(s_1) &= \begin{cases} M_a(s_1) - 1, & \text{if } M_a(s_1) < cut_p \\ cut_p - 1 \textbf{ or } \omega, & \text{if } otherwise. \end{cases} \\ I_a(s_2) &= \begin{cases} M(s_2) + 1, & \text{if } M(s_2) < cut_p - 1 \\ \omega, & \text{otherwise .} \end{cases} \\ I_a(s) &= M(s), \text{ for } s \in S_{\mathcal{P}} \backslash \{s_1, s_2\} \end{aligned} \tag{1}$$

—Otherwise, $\forall s \in S_{\mathcal{P}} \cdot I_a(s) = M_a(s)$.

If α does not create new process, then $M_a' = I_a$. Otherwise, suppose by executing α, a process of type q is created and starts its execution from state $s_q \in S_q$. Then we set

—for state s_q,

$$M_a'(s_q) = \begin{cases} I_a(s_q) + 1, & \text{if } I_a(s_q) < cut_q - 1 \\ \omega, & \text{otherwise.} \end{cases} \tag{2}$$

—for $s \in S_{\mathcal{P}} \backslash \{s_q\}$, $M_a'(s) = I_a(s)$.

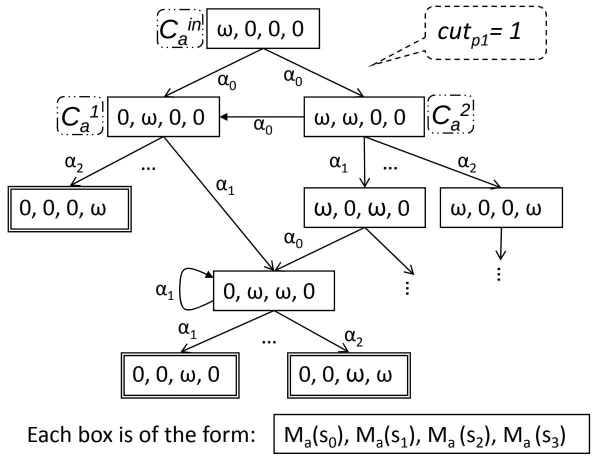

Fig. 4. Partial (abstract) state exploration graph for p_1 ($N_{p_1} = \omega$).

We use the relation $\mathbf{update_a}(s_1, M_a, \alpha, s_2, M'_a)$ to denote that mapping M'_a can be derived from M_a due to migration of a process from state s_1 to s_2 by executing action α.

Note that, when there are ω processes in the source state s_1 at configuration $C_a = (v, M_a)$ (i.e. $M_a(s_1) = \omega$) and the destination state s_2 is different from s_1, then two possible configurations may result from C_a as described above (see Eqs. (1)). If $C'_a = (v', M'_a)$ represents the resulting abstract configuration, then process count in state s_1 at configuration C'_a (i.e. $M'_a(s_1)$) is either– (i) ω, assuming there were greater than cut_p processes in s_1 at configuration C_a, or (ii) $cut_p - 1$, assuming there were exactly cut_p processes in s_1 at configuration C_a. Given the above notion of abstract configurations *CFG*$_{abs}$, we define an abstract transition relation $\hookrightarrow_a \subseteq$ *CFG*$_{abs} \times \Sigma \times$ *CFG*$_{abs}$ as follows.

DEFINITION 5 ABSTRACT TRANSITION RELATION $\hookrightarrow_a$. *Let* $\mathcal{S} = (Var_G, v^g_{in}, TS_{\mathcal{P}})$ *be a system model,* $C_a = (v_g, M)$, $C'_a = (v'_g, M'_a) \in$ *CFG*$_{abs}$ *be its abstract configurations, and* $\alpha \in \Sigma$ *be an action. Then* $(C_a, \alpha, C'_a) \in \hookrightarrow_a$ *if and only if* $\exists p \in \mathcal{P}, \exists\, s = (l, v), s' = (l', v') \in S_p$, *s.t.*

(1) $(l, \alpha, l') \in \rightarrow_p$ *is a transition in* TS_p,

(2) $M_a(s) \geq 1$, *i.e. there is at least one process in* s,

(3) v *and* v_g *satisfy the pre-condition* Pre_α.

(4) $v'(v'_g)$ *is the effect of post-condition* $Post_\alpha$ *on* $v(v_g)$. *Here* v'_g *represents an update of global variables* Var_G, *which can later be read/updated by other processes, thus allowing for* shared variable communication.

(5) *If last action executed was a* send[2] *event, then* α *must be the corresponding receive event. If the matching receive cannot be executed, then the last executed send event is rolled back, and some other enabled action is executed in its place.*

(6) *The relation* $update_a(s, M_a, \alpha, s', M'_a)$ *holds.*

[2] Recall that, we only consider synchronous message communication.

For illustration, we again consider the example in Figure 2. Assume that, an unbounded number of processes (ω) of type $p1$ are created initially, and the default cutoff number $cut_{p1} = 1$ is used. For $i \in [0, 3], s_i = (l_i, \epsilon)$, where l_i is a control location in TS_{p1}, and since $p1$ has no local variables, their valuation is represented as ϵ. Further, assuming that there are no global variables, we omit global variable valuation from the abstract configurations. Hence, we represent the abstract configurations for this system by a mapping M_a, such that $M_a(s_i), i \in [0, 3]$ represents the number of p_1 processes in state s_i. Its partial abstract state exploration graph is shown in Figure 4. Initially, action α_0 is executed by a process in state s_0 from the initial configuration (C_a^{in}), resulting in two different configurations C_1 and C_2. The configuration C_a^1 (towards left) corresponds to the case where ω represents exactly one process in state s_0 at C_a^{in}, while configuration C_a^2 (towards right) corresponds to the case where ω represents two or more processes in state s_0 at C_a^{in}. Further, all actions α_0, α_1 and α_2 are enabled at configuration C_a^2, while only α_1 and α_2 are enabled at configuration C_a^1. The paths following these abstract configurations are explored in a similar manner.

4.2 Soundness of Abstraction

We now show the soundness of proof search over the abstract state space. Before proceeding to the proof, we first define a relation $\simeq \subseteq$ *CFG* $\times$ *CFG*$_{abs}$ as follows.

DEFINITION 6. *For all $C_c = (v_c^g, M_c) \in$ CFG and $C_a = (v_a^g, M_a) \in$ CFG$_{abs}$, $C_c \simeq C_a$ iff $v_c^g = v_a^g$, and $\forall p \in \mathcal{P}, \forall s \in S_p$, $M_a(s) \geq |M_c(s)|$.*

We now prove that our abstract execution semantics is an over-approximation of the concrete execution semantics.

THEOREM 1. *Let σ be a possibly infinite sequence of actions that can be exhibited in the concrete execution of a system model $\mathcal{S}$ with initially $N_p^c \in \mathbb{N}$ processes of type p. Then, σ can be exhibited in the abstract execution of $\mathcal{S}$ with initially N_p^a processes of type p, where either $N_p^a = \omega$ or $N_p^a \in \mathbb{N}$ s.t. $N_p^a \geq N_p^c$.*

PROOF. In order to prove this theorem, we consider the following property. Recall that $\hookrightarrow$ and $\hookrightarrow_a$ denote the concrete and abstract transition relations respectively.

Property 1:
$\forall (C_c, \alpha, C_c') \in \hookrightarrow, \forall C_a \in$ *CFG*$_{abs}$, $C_c \simeq C_a \Rightarrow \exists (C_a, \alpha, C_a') \in \hookrightarrow_a$, s.t. $C_c' \simeq C_a'$.

We prove the above property as follows. Let $C_c = (v_c^g, M_c)$, $C_c' = (v_c^{g'}, M_c') \in$ *CFG*, and $C_a = (v_a^g, M_a) \in$ *CFG*$_{abs}$, s.t. $(C_c, \alpha, C_c') \in \hookrightarrow$ and $C_c \simeq C_a$. Suppose by executing α in concrete execution, a process o of type p moves from state s_α to state s_α'. Hence, $update_c(s_\alpha, M_c, \alpha, s_\alpha', M_c')$ from Def. 3 holds.

Since $C_c \simeq C_a$, by Def. 6, $v_c^g = v_a^g$; moreover, $M_a(s_\alpha) \geq |M_c(s_\alpha)|$ and $M_a(s_\alpha') \geq |M_c(s_\alpha')|$. Thus, action α can be executed by choosing a process from state s_α in the abstract execution. Let $C_a' = (v_a^{g'}, M_a')$ be the resulting abstract configuration. Without loss of generality, we assume that $s_\alpha \neq s_\alpha'$.

If $M_a(s_\alpha) \in \mathbb{N}$, by the definition of $update_c$ and $udpate_a$ (ref. Sections 3 and 4), we have intermediate mappings $I_c'(s_\alpha) = I_c(s_\alpha) - \{o\}$ and $I_a'(s_\alpha) = I_a(s_\alpha) - 1$. Now, we consider the following cases based on action α.

(1) If α does not create new process and $s_\alpha \neq s'_\alpha$, then $M'_a(s_\alpha) = I'_a(s_\alpha)$ and $M'_c(s_\alpha) = I'_c(s_\alpha)$. Since $M_a(s_\alpha) \geq |M_c(s_\alpha)|$, we have $M'_a(s_\alpha) \geq |M'_c(s_\alpha)|$.
(2) If, by executing α, a new process o_q of type q is created and starts its execution at state s_q, then we have $M'_c(s_q) = M_c(s_q) \cup \{o_q\}$ and $M'_a(s_q) = M_a(s_q) + 1$. Since $M_a(s_\alpha) \geq |M_c(s_\alpha)|$, we have $M'_a(s_\alpha) \geq |M'_c(s_\alpha)|$.

If $M_a(s_\alpha) = \omega$, then by the definition of $update_a$, we always allow the possibility that $M'_a(s_\alpha) = \omega$. By the similar argument as in the case of $M_a(s_\alpha) \in \mathbb{N}$, we have $M'_a(s_\alpha) \geq |M'_c(s_\alpha)|$. Similar argument applies to $M'_a(s'_\alpha)$. Finally since the effect of action α is the same on v^g_c and v^g_a, we have $v^{g'}_c = v^{g'}_a$. Therefore, $C'_c \simeq C'_a$.

Property 1 establishes that $\simeq$ is a simulation relation. To complete the proof of the main theorem, we only need to show that the initial configurations in the concrete and abstract execution semantics are related by $\simeq$. This is indeed the case, and this concludes the proof. □

4.3 When is the abstraction exact?

We have now established that our abstract execution semantics is an *over approximation* — any execution trace exhibited in the concrete execution semantics is also exhibited in the abstract execution semantics. Further, our abstract execution semantics is *exact* (*i.e.*, any sequence of actions allowed by the abstract execution semantics is also allowed by the concrete execution semantics) iff the following conditions hold–

C1. In abstract execution, the process counts in a process type p are always represented using a natural number (i.e. they never become ω) and are updated following the usual arithmetic rules. Note that cut_p does not play any role in this case.

C2. For each process type p, the initial number of processes in abstract execution (N^a_p) is equal to the initial number of processes in the concrete execution (N^c_p), i.e. $N^a_p = N^c_p$.

DEFINITION 7. *Let CFG_{abs} (CFG) be the set of abstract (concrete) configurations of system model $\mathcal{S}$. Then for all $C_a = (v^g_a, M_a) \in CFG_{abs}$ and $C_c = (v^g_c, M_c) \in CFG$, $C_a \simeq_a C_c$ iff $v^g_a = v^g_c$ and $\forall s \in S_\mathcal{P}, (M_a(s) = |M_c(s)|)$.*

THEOREM 2. *Let $\sigma = \alpha_0\alpha_1 \ldots$ be a possibly infinite action sequence exhibited in the abstract execution of $\mathcal{S}$ satisfying* **C1***, then σ can be exhibited in the concrete execution of $\mathcal{S}$ satisfying* **C2***, and for $i \geq 0$, $C^i_a \simeq_a C^i_c$, where C^i_a (C^i_c) is the abstract (concrete) configuration before the abstract (concrete) execution of α_i.*

PROOF. In order to prove this theorem, we consider the following property. The $\hookrightarrow$ and $\hookrightarrow_a$ denote the concrete and abstract transition relations respectively.

Property 2:
$\forall (C_a, \alpha, C'_a) \in \hookrightarrow_a, \forall C_c \in \mathit{CFG}, C_a \simeq_a C_c \Rightarrow \exists (C_c, \alpha, C'_c) \in \hookrightarrow$, s.t. $C'_a \simeq_a C'_c$.

We prove the above property as follows. Let $C_a = (v^g_a, M_a), C'_a = (v^{g'}_a, M'_a) \in \mathit{CFG}_{abs}$, and $C_c = (v^g_c, M_c) \in \mathit{CFG}$, s.t.$(C_a, \alpha, C'_a) \in \hookrightarrow_a$ and $C_a \simeq_a C_c$. Suppose by executing action α in the abstract execution, a process of type p moves from state s_α to s'_α, where $s_\alpha, s'_\alpha \in S_p$.

Since $C_a \simeq_a C_c$, we have $v^g_a = v^g_c$ and $\forall s \in S_\mathcal{P}, (M_a(s) = |M_c(s)|)$, which implies that $\forall s \in S_\mathcal{P}, M_a(s) \in \mathbb{N}$. Hence, by the definition of $update_a$, we have intermediate mappings $I'_a(s_\alpha) = M_a(s_\alpha) - 1$ and $I'_a(s'_\alpha) = M_a(s'_\alpha) + 1$. Moreover α can be executed by a process o_p of type p at state s_α in the concrete execution, which results in the intermediate mappings – $I_c(s_\alpha) = M_c(s_\alpha) - \{o_p\}$ and $I_c(s'_\alpha) = M_c(s'_\alpha) + \{o_p\}$. Therefore, we have $I_a(s_\alpha) = |I'_c(s_\alpha)|$ and $I_a(s'_\alpha) = |I'_c(s'_\alpha)|$. Now, based on the characteristic of action α, we consider the following cases:

(1) If α does not create new process, then for abstract execution, we have $M'_a(s_\alpha) = I_a(s_\alpha)$, $M'_a(s'_\alpha) = I_a(s'_\alpha)$; and for concrete execution, we have $M'_c(s_\alpha) = I_c(s_\alpha)$, $M'_c(s'_\alpha) = I_c(s'_\alpha)$. Therefore, $M'_a(s_\alpha) = |M'_c(s_\alpha)|$, $M'_a(s'_\alpha) = |M'_c(s'_\alpha)|$.

(2) If, by executing α, a new process of type q is created and starts its execution at local state s_q. Moreover, in concrete execution, this new process is identified as o_q. Then, in abstract execution, we have $M'_a(s_q) = I_a(s_q) + 1$, and for all other $s \in \mathcal{S}^c_{\mathcal{P}}$, $M'_a(s) = I_a(s)$. In concrete execution, we have $M'_c(s_q) = I_c(s_q) \cup \{o_q\}$, and for all other $s \in \mathcal{S}_{\mathcal{P}}$, $M'_c(s) = I_c(s)$. Therefore, we have $M'_a(s_\alpha) = |M'_c(s_\alpha)|$, $M'_a(s'_\alpha) = |M'_c(s'_\alpha)|$.

Finally, the effect of α is the same on v^g_c and v^g_a. Therefore, we have $C'_a \simeq_a C'_c$.

Property 2 establishes that $\simeq_a$ is a simulation relation. It is easy to see that $C^0_a \simeq_a C^0_c$, i.e. the initial configurations in the abstract and concrete execution semantics are related by $\simeq_a$. This concludes the proof. □

5. EXTENDING ABSTRACTION WITH COUNT VARIABLES

In the previous section we discussed our state space abstraction which involved abstracting away process ids. From the real life case studies that we have modeled for our experiments, we observe that this counter abstraction alone is not sufficient for modeling most of these examples. These examples generally involve a process (e.g. a *controller*) that needs to communicate with, and maintain a count of processes of another type (e.g. several *clients*). Then, if we intend to verify a system which has an unbounded number of processes, say of type p, we cannot use a variable with a finite domain to keep a count of p-processes.

5.1 Extended Abstraction Scheme

In order to keep track of the number of processes of type p with an unbounded number of processes, we introduce *process-count* variables having the domain $\mathbb{N} \cup \{\omega\}$. We denote the set of all process-count variables as Var^ω. For a process-count variable, we only allow assignment operation that initializes it with a constant value or ω, as well as operations that increment or decrement its value by 1, and obey the following execution semantics[3]. For a process-count variable $v \in Var^\omega$ used for counting processes of type p:

$$v{+}{+} \;=\; \begin{cases} v+1, & v < cut_p - 1 \\ \omega, & \text{otherwise.} \end{cases}$$

$$v{-}{-} \;=\; \begin{cases} v-1, & v < cut_p \\ cut_p - 1 \textbf{ or } \omega, & \text{otherwise.} \end{cases}$$

Moreover, v can be involved in a boolean expression: $B \equiv v$ **Relop** c, where **Relop** is a relational operator and $c \in \mathbb{N}$. Here, we only consider the case where **Relop** is $\leq$. If

[3]The abstract semantics can be similarly extended to support increment/decrement of a process count variable by any constant number c. The case $c = 1$, worked out here, is most common.

$v \in [0, cut_p)$, B evaluates to *true* if $v \leq c$, and *false* otherwise. If $v = \omega$ and $c < cut_p$, then B is *false*. Otherwise, if $v = \omega$ and $c \geq cut_p$, then we non-deterministically allow B to be either *true* or *false*. Various other relational operators are considered in a similar manner.

Thus, if process type p has ω processes in abstract execution, the value of v lies in the domain $[0, cut_p) \cup \{\omega\}$, where ω indicates the value of v to be cut_p or greater. Further, when v is decremented by one (i.e. $v--$), if the original value of v is ω, then the resulting value of v is non-deterministically chosen to be either $cut_p - 1$ or ω. The former (latter) choice corresponds to the possibility that value of v was equal-to (greater-than) cut_p.

In PROMELA, a process-count variable used for counting processes of type p, is declared using the following syntax– '*abs_p_X*', where p is the process-type and X is any valid string allowed in a variable name in PROMELA. This specific format allows the verifier to identify and update these variables as per the rules described above.

Since the domain of variable $x \in Var^{\omega}$ includes the unbounded value represented as ω, to distinguish it from the concrete domain of x, i.e. $\mathcal{D}_x$ ($= \mathbb{N}$), we represent the abstract domain as $\mathcal{D}^a_x$ (*i.e.* $\mathbb{N} \cup \{\omega\}$). Note that, for all other variables $y \in (Var_{\mathcal{P}} \cup Var_G) \setminus Var^{\omega}$, $\mathcal{D}^a_y = \mathcal{D}_y$. Then, we use Val^a_p to represent the abstract valuations of variables in Var_p, which is a mapping for each variable to its abstract domain. Let $Val^a_{\mathcal{P}} = \bigcup_{p \in \mathcal{P}} Val^a_p$. The abstract valuations of global variables is represented similarly as Val^a_G. Accordingly, the abstract states of a process type p are represented as $S^a_p \subseteq L_p \times Val^a_p$, where L_p if the set of local states in the transition system TS_p as before. The abstract initial p-state is given by $s^a_{p,in} = (l^p_{in}, v^a_{p,in})$, where $v^a_{p,in}$ is an initial valuation of variables in p over abstract variable domain. Further, we set $S^a_{\mathcal{P}} = \bigcup_{p \in \mathcal{P}} S^a_p$.

Since the domain of variables in Var^{ω} differs in the abstract execution as compared to the concrete execution, we are to establish a relation between valuation of variables in the concrete and abstract execution as following. Let $\mathcal{R}$ be a relation between valuation of variables in concrete and abstract domain: $\mathcal{R} \subseteq (Val_G \times Val_{\mathcal{P}}) \times (Val^a_G \times Val^a_{\mathcal{P}})$. For all $g \in Val_G, f \in Val_{\mathcal{P}}, g_a \in Val^a_G, f_a \in Val^a_{\mathcal{P}} : (g, f)\mathcal{R}(g_a, f_a)$ iff

(1) $\forall v \in Var_G \setminus Var^{\omega}, g_a(v) = g(v)$.
(2) $\forall v \in Var_{\mathcal{P}} \setminus Var^{\omega}, f_a(v) = f(v)$.
(3) $\forall v \in Var_G \cap Var^{\omega}, g_a(v) = g(v)$, if $g(v) \in [0, cut_p)$; and $g_a(v) = \omega$, otherwise.
(4) $\forall v \in Var_{\mathcal{P}} \cap Var^{\omega}, f_a(v) = f(v)$, if $f(v) \in [0, cut_p)$; and $f_a(v) = \omega$, otherwise.

5.2 Soundness with Process-count Variables

We now refine the relation $\simeq \subset$ *CFG* $\times$ *CFG*$_{abs}$ with respect to $\mathcal{R}$ between concrete and abstract configurations as follows.

DEFINITION 8. *For all* $C_c = (v^g_c, M_c) \in$ *CFG and* $C_a = (v^g_a, M_a) \in$ *CFG*$_{abs}$, $C_c \simeq C_a$ *iff* $\forall p \in \mathcal{P}$, $\forall s_c = (l, v_c) \in S_p, \exists s_a = (l, v_a) \in S^a_p, s.t. (v^g_c, v_c)\mathcal{R}(v^g_a, v_a) \wedge M_a(s_a) \geq |M_c(s_c)|$.

We now prove that our abstract execution semantics is an over-approximation of the concrete execution semantics.

THEOREM 3. *Let* σ *be a possibly infinite sequence of actions that can be exhibited in the concrete execution of a system model* $\mathcal{S}$ *with initially* $N^c_p \in \mathbb{N}$ *processes of type* p.

Then, σ can be exhibited in the abstract execution of $\mathcal{S}$ with initially N_p^a processes of type p, where either $N_p^a = \omega$ or $N_p^a \in \mathbb{N}$ s.t. $N_p^a \geq N_p^c$.

PROOF. In order to prove this theorem, we consider the following property. Recall that $\hookrightarrow$ and $\hookrightarrow_a$ denote the concrete and abstract transition relations respectively.

Property 1:
$\forall (C_c, \alpha, C_c') \in \hookrightarrow, \forall C_a \in \textit{CFG}_{abs}, C_c \simeq C_a \Rightarrow \exists (C_a, \alpha, C_a') \in \hookrightarrow_a$, s.t. $C_c' \simeq C_a'$.

We prove the above property as follows. Let $C_c = (v_c^g, M_c)$, $C_c' = (v_c^{g'}, M_c') \in \textit{CFG}$, and $C_a = (v_a^g, M_a) \in \textit{CFG}_{abs}$, s.t. $(C_c, \alpha, C_c') \in \hookrightarrow$ and $C_c \simeq C_a$. Suppose by executing α, a process o of type p is moved from state $s_c = (l, v_c)$ to state $s_c' = (l', v_c')$. Hence, $update_c(s_c, M_c, \alpha, s_c', M_c')$ in Def. 3 holds.

Since $C_c \simeq C_a$, by Def. 8, $\exists s_a = (l, v_a) \in S_{\mathcal{P}}^a, s.t.(g, f)\mathcal{R}(g_a, f_a) \wedge M_a(s_a) \geq |M_c(s_c)|$, where $g \in Val_G, f \in Val_{\mathcal{P}}, f_a \in Val_G^a, f_a \in Val_{\mathcal{P}}^a$. Since α is executable from s_c, to show that α is also executable from s_a in the abstract execution, we first need to show that v_a^g and v_a satisfy Pre_α. Since $(v_c^g, v_c)\mathcal{R}(v_a^g, v_a)$, let x be a variable in the system, we consider the following cases:

(1) If $x \in Var_G \backslash Var^\omega$, then $g_a(x) = g(x)$, and hence $g_a(x)$ satisfies Pre_α. Similar argument applies to $x \in Var_{\mathcal{P}} \backslash Var^\omega$.

(2) If $x \in Var_G \cap Var^\omega$, then $g_a(x) \geq g(x)$. Now, consider boolean expression $B \equiv x$**Relop**c, where **Relop** is any relational operator and c is a constant. Here, we take the case where **Relop** is $\leq$ as an example. Various other relational operators can be considered in a similar fashion. Since $g(x) \leq c$ evaluates to *true*, we consider the evaluation of $g_a(x) \leq c$ as follows.
—If $g(x) \in [0, cut_p)$, then $g_a(x) = g(x)$. Hence, $g_a(x) \leq c$ evaluates to *true*.
—If $g(x) \geq cut_p$, then $g_a(x) = \omega$. Let $g(x) = n_0 \in \mathbb{N}$. Since ω represents a value greater than or equal to cut_p, $g_a(x)$ is possible to evaluate to n_0 ($n_0 \geq cut_p$). Therefore, we always allow the possibility that $g_a(x) \leq c$ evaluates to *true*.
Hence, whenever $g(x)$ satisfies Pre_α, $g_a(x)$ satisfies Pre_α. Similar argument applies to $x \in Var_{\mathcal{P}} \backslash Var^\omega$.

Let $C_a' = (v_a^{g'}, M_a') \in \textit{CFG}_{abs}$ be the resulting abstract configuration, such that a process moves to state $s_a' = (l', v_a') \in S_p^a$ by executing α in abstract execution. Consider the global and local variables. For a variable x, let v_c and v_c' be the valuation of x before and after execution of α in concrete execution, respectively. Their corresponding valuations in the abstract execution are v_a and v_a'. If $x \in Var_G \backslash Var^\omega$, then since $v_c = v_a$ and the effect of α on x in concrete and abstract executions are identical. Hence, we have $v_c' = v_a'$. Similar result is obtained for $Var_{\mathcal{P}} \backslash Var^\omega$. If $x \in Var^\omega$, since $v_a \geq v_c$, by the operational semantics for abstract-count variables, we have $v_a' \geq v_c'$. Hence, we can easily see that $(v_c^{g'}, v_c')\mathcal{R}(v_a^{g'}, v_a')$.

Further, similar to the proof of Theorem 1 in Section 4, by the semantics of $update_c$ and $udpate_a$, we have $M_a'(s_a) \geq |M_c'(s_c)|$ and $M_a'(s_a') \geq |M_c'(s_c')|$. Therefore, $C_c' \simeq C_a'$.

Property 1 establishes that $\simeq$ is a simulation relation. To complete the proof of the main theorem, we only need to show that the initial configurations in the concrete and abstract execution semantics are related by $\simeq$. This is indeed the case, and this concludes the proof. $\square$

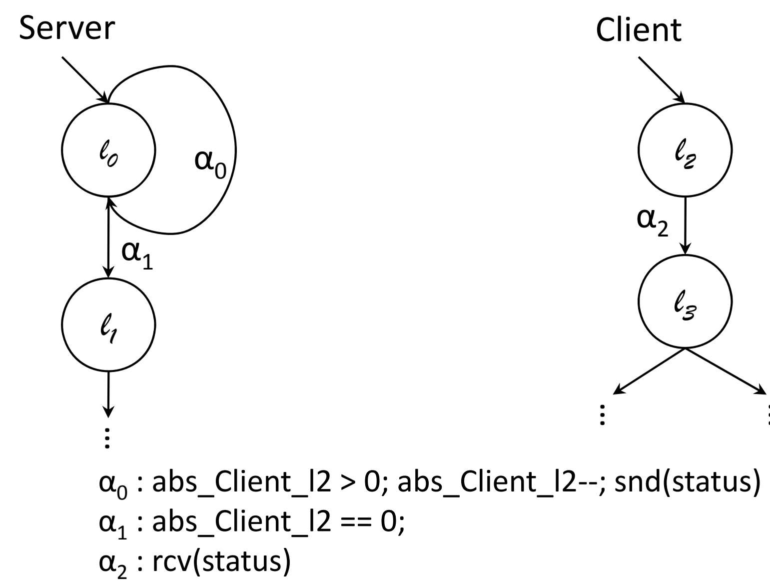

Fig. 5. Example of using process-count variable

Note that under condition **C1** and **C2** (refer Page 12), $S_p^a = S_p$ for all process type p and $D_x^a = D_x$ for all variable x. Hence, our abstraction is again *exact*. Theorem 2 and its proof in Section 4 also apply in this case.

5.3 Elimination of Spuriousness Caused by Process-Count Variables

In our modeling, process-count variables can be used as a global shared-variable or local variable that keeps track of the number of processes in a particular control state of a process type. The use of data abstraction, in particular, process-count variables, introduces extra spurious behaviors in the system. This is mainly due to the fact that a process-count variable and the actual number of processes that it keeps track of are both updated in a non-deterministic manner. A server-client example in Figure 5 demonstrates the issue. In this example, a Server informs all the connected Clients of its status, and the number of connected clients is maintained via Server's local variable abs_Client_l2. After one execution of send and receive actions, four global states can be generated (Figure 6). The global states in dotted box are spurious, since in these states, the process-count variable (abs_Client_l2) is unbounded while the actual number of processes in the corresponding state (l_2) is a concrete number less than cut_{Client}, or vise-versa.

To eliminate the spuriousness caused by using process-count variables, we require that a process-count variable to be associated with a control state whose number of processes it keeps track of. In PROMELA, to count the number of processes at local state l of a process type p, we follow the naming convention abs_p_l for the associated process-count variable. Moreover, an internal variable (name it as int_p_l) is used to maintain the execution choice of abs_p_l (i.e., $\omega - 1 = \omega$ or $\omega - 1 = cut_p$). During state space exploration, when control reaches an execution state $s = (l, v_p)$ of process type p and an action $\alpha \in \Sigma$ is enabled, a check is performed to make sure that $M_a(s)$ matches the value of abs_p_l (note that, the only possible reason for the two not matching is because the initial assignment

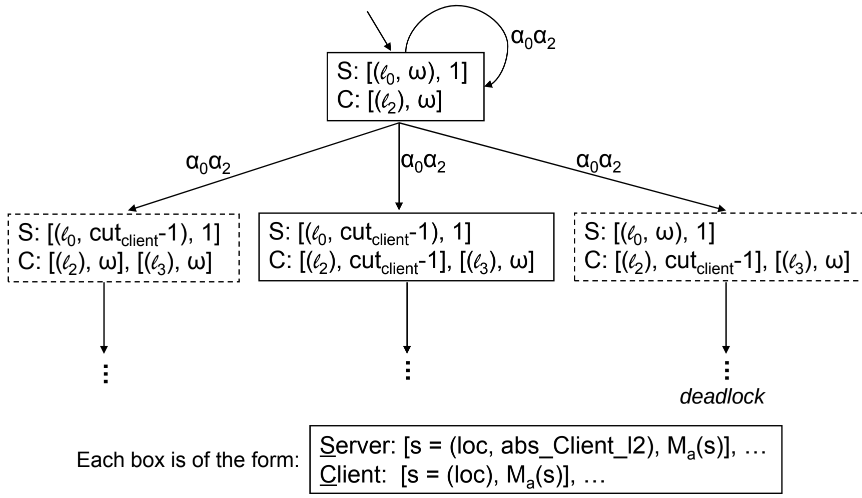

Fig. 6. Partial state exploration in the presence of process-count variable

of abs_p_l does not equal to $M_a(s)$, and hence should be reported as an error); moreover, after the execution of α, $M_a(s)$ is updated following our abstract semantics but with the same execution choice as the update of abs_p_l. Consider the Server-Client example in Figure 5. The control state associated with process-count variable abs_Client_l2 is l_2 of process type $Client$. When abs_Client_l2 is updated as a post-condition of action α_0, internal variable int_Client_l2 is updated; and when α_2 is enabled from an execution state $s = (l_2, v_{Client})$, $M_a(s)$ is updated based on the value of int_Client_l2, which ensures that the spurious global states (the dotted boxes in Figure 6) are never reached.

To summarize, the use of process-count variables in a PROMELA model requires the following steps:

(1) Identify and label the control states that associate with any process-count variables.

(2) Declare process-count variable (either local or global) using naming convention abs_p_l, such that it counts the number of processes at control state (with label) l of process type p.

6. VERIFICATION

We now elaborate our verification procedure outlined earlier in Figure 1. It proceeds on the abstract state representation discussed in the preceding.

6.1 Model Checking

We use *linear-time temporal logic* (LTL) [Manna and Pnueli 1991] for specifying the properties to be verified. This decision is influenced by our use of model checker SPIN [Holzmann 2003] for implementing our verification framework. SPIN uses LTL as a property specification language. Properties in LTL are specified using atomic-propositions, boolean-operators ($\neg$, $\vee$, $\wedge$), and *temporal-operators* (G, F, X, U, R).

As described earlier in Section 5, our system model may also contain process-count variables (denoted as Var^{ω}), such that a variable $v \in Var^{\omega}$ is used for counting processes of a given type, say p, with its domain ranging over $[0, cut_p) \cup \{\omega\}$. Then, for LTL property specification, we restrict the boolean expressions involving a global process-count variable v to be of the form v $\mathbf{Relop}$ c, such that $c \in [0, cut_p)$ and $\mathbf{Relop}$ is any relational operator. This restriction ensures deterministic evaluation of boolean expressions involving the process-count variables.

Since we use SPIN as our underlying implementation framework, we are able to take advantage of its model checking capabilities. SPIN performs on-the-fly (explicit) state-space construction, while trying to find a counterexample trace violating the property being verified. If such a trace cannot be found, it means that the property holds true in the given system model. Otherwise, a counterexample trace indicating property violation is reported by SPIN. Our abstract execution semantics allow us to verify a family of concrete systems as follows.

Suppose a LTL property φ is satisfied in our abstract verification with N_p^a processes of type p, where $N_p^a \in \mathbb{N} \cup \{\omega\}$. Then, from Theorem 1, φ is also satisfied by all concrete systems having $N_p^c \leq N_p^a$ processes of type p, where $N_p^c \in \mathbb{N}$.

6.2 Spurious counter-example detection

Our abstract execution semantics is an over-approximation in terms of allowed execution traces. Thus, a counter-example trace obtained from model checking over the abstract state space may be *spurious*, i.e. it cannot be exhibited in *any* concrete system with a finite number of processes (less-than or equal-to the number of processes in the abstract execution for each process type). In the following, we present an approach for detecting spurious counter-examples in the absence of process-count variable, and discuss abstraction-refinement for eliminating them in the next section. The result also holds in the presence of process-count variable with spuriousness elmination introduced in section 5.3.

We now introduce some definitions on finite traces. *Note that our spurious counter-example detection and abstraction-refinement work for finite as well as infinite counter-example traces*. The notions we introduce now, will work on finite prefixes of counter-example traces obtained from model checking.

Let $\sigma = \alpha_1 \ldots \alpha_n \in \Sigma^*$ be a finite execution trace s.t., action α_i is executed by a process moving from execution state $s_i \in S_{\mathcal{P}}$ to state $s'_i \in S_{\mathcal{P}}$. We set $src(\alpha_i) = s_i$ and $dst(\alpha_i) = s'_i$. For a state $s \in S_{\mathcal{P}}$, we define:

$$\begin{aligned} in(s, \sigma) &= |\{i \mid dst(\alpha_i) = s, i \in [1, n]\}| \\ out(s, \sigma) &= |\{i \mid src(\alpha_i) = s, i \in [1, n]\}| \end{aligned}$$

Here, $in(s, \sigma)$ ($out(s, \sigma)$) gives the number of processes moving in to (out of) state s during the execution of σ. We use $new(s, \sigma)$ ($del(s, \sigma)$) to represent the number of processes that are created (deleted) during execution of σ such that, they start (terminate) their execution in state s. For convenience, we also define the following terms: $enter(s, \sigma) = in(s, \sigma) + new(s, \sigma)$, and $leave(s, \sigma) = out(s, \sigma) + del(s, \sigma)$. Finally, we define predicate $valid(s, \sigma)$ as:

$$init(s) + enter(s, \sigma) - leave(s, \sigma) > 0 \tag{3}$$

Further, for a given finite trace σ and a process type p we define the quantity $n_{p,\sigma}$ as follows. We first determine $leave(s_{in}^p, \sigma)$, the number of p processes that move out from the initial p-state $s^p_{in} \in S_p$ during the execution of σ. Then, we define

$$n_{p,\sigma} = min(N_p, leave(s^p_{in}, \sigma)) \tag{4}$$

where $N_p \in \mathbb{N} \cup \{\omega\}$ is the initial number of processes of type p in the abstract verification run, i.e. $N_p = init(s^p_{in})$. Note that $n_{p,\sigma} \in \mathbb{N}$. Let $Pre(\sigma)$ denote the set of all prefixes of σ (excluding σ). We now consider two cases, based on whether a counter-example is finite or infinite.

Case-A: σ is finite. Let $\sigma = \alpha_0 \ldots \alpha_n$ be a finite counter-example trace obtained from an abstract verification run s.t., action α_i is executed by a process moving from state $s_i \in S_{\mathcal{P}}$ to state $s'_i \in S_{\mathcal{P}}$, i.e. $src(\alpha_i) = s_i$ and $dst(\alpha_i) = s'_i$. We show that σ *is non-spurious* $\Leftrightarrow \forall \gamma \in Pre(\sigma)$*,* $valid(s_{|\gamma|}, \gamma)$ *is true in abstract execution.*

PROOF. **A.1 $\Leftarrow$:** Assume that $\forall \gamma \in Pre(\sigma)$, $valid(s_{|\gamma|}, \gamma)$ is *true* in the abstract execution. We now show σ to be non-spurious, by showing that σ can be exhibited in the concrete execution of a system where each process type p initially has $n_{p,\sigma}$ processes. The proof proceeds by induction on the length of σ.
<u>*Base case*</u>: It holds trivially for $|\sigma| = 0$.
<u>*Induction hypothesis*</u>: Trace $\sigma_1 = \alpha_0 \ldots \alpha_{k-1}$ can be exhibited in a concrete execution with initially $n_{p,\sigma}$ number of p-processes. Further, for all $0 \leq i < k$, action α_i has the same source state $src(\alpha_i)$ and destination state $dst(\alpha_i)$ in both concrete and abstract execution.
<u>*Inductive step*</u>: We now consider execution of $\sigma_1 \cdot \alpha_k$. Since, σ_1 is a prefix of σ and $\forall \gamma \in Pre(\sigma)$, $valid(s_{|\gamma|}, \gamma)$ is true in the abstract execution, $valid(s_k, \sigma_1)$ also holds in the abstract execution. Here, $k = |\sigma_1|$ and s_k is the state from which a process executes α_k in the abstract execution. Thus, $init(s_k) + enter(s_k, \sigma_1) - leave(s_k, \sigma_1) > 0$ in the abstract execution (see Eq. (3)). Note that, both $leave(s_k, \sigma_1)$ and $enter(s_k, \sigma_1)$ depend on the source and destination states of processes executing various actions in σ_1. Since σ_1 is also exhibited in concrete execution (from induction hypothesis), the value of these quantities in concrete execution will be the same as in the abstract execution. We now consider following two cases for execution of α_k in the concrete execution.

1. If $s_k \in S_p$ is not the initial p-state ($s_k \neq s^p_{in}$), then $init(s_k) = 0$ in both abstract and concrete executions. Since, $valid(s_k, \sigma_1)$ is *true* in the abstract execution, we get $enter(s_k, \sigma_1) > leave(s_k, \sigma_1)$, which will also hold in the concrete execution. Hence, there is at least one process in state s_k in concrete execution (after σ_1) which can be chosen to execute α_k.

2. If $s_k = s^p_{in}$ is the initial p-state, then in the concrete execution there will be initially $n_{p,\sigma} = min(N_p, leave(s_k, \sigma))$ processes in state s_k. Since $valid(s_k, \sigma_1)$ holds *true* in the abstract execution, we get $init(s_k)(= N_p) + enter(s_k, \sigma_1) > leave(s_k, \sigma_1)$. Recall that $n_{p,\sigma} = min(N_p, leave(s_k, \sigma))$, as we are considering the case $s_k = s^p_{in}$. If $n_{p,\sigma} = N_p$ in concrete execution, since values of $enter(s_k, \sigma_1)$ and $leave(s_k, \sigma_1)$ in concrete execution are same as in the abstract execution, $valid(s_k, \sigma_1)$ is also *true* in the concrete execution – a process can then be chosen from s_k to execute α_k in concrete execution. Otherwise, $n_{p,\sigma} = leave(s_k, \sigma)$. Since in abstract execution α_k is executed by a process in state s_k (after occurrence of σ_1), we have $leave(s_k, \sigma) \geq leave(s_k, \sigma_1) + 1$. Thus, in the concrete execution $valid(s_k, \sigma_1)$ also holds since, $init(s_k)(= leave(s_k, \sigma)) + enter(s_k, \sigma_1) > leave(s_k, \sigma_1)$, and a process from state s_k can be chosen to execute α_k.

A.2 $\Rightarrow$: We show this by contradiction. Assume that $\sigma = \alpha_0 \ldots \alpha_n$ is non-spurious. Then σ can be exhibited in a concrete execution with $N_p^c \leq N_p$ ($N_p^c \in \mathbb{N}$) number of processes of type p, s.t. $src(\alpha_i) = s_i$ and $dst(\alpha_i) = s'_i$. Now assume that there exists a $\gamma = \alpha_0 \ldots \alpha_{k-1} \in Pre(\sigma)$ such that for $j \in [0, k-1]$ $src(\alpha_j) = s_j$, $dst(\alpha_j) = s'_j$, and $valid(s_k, \gamma)$ ($k = |\gamma|$) is *false*. This implies $init(s_k) + enter(s_k, \gamma) - leave(s_k, \gamma) \leq 0$ in the abstract execution. If $s_k = s_{in}^p$ is the initial state of a type p, then initially there are N_p processes in state s_k in the abstract execution (i.e. $init(s_k) = N_p$ in abstract execution). In concrete execution $init(s_k)$ will be equal to $N_p^c \leq N_p$. Otherwise, if s_k is not an initial state of any process type, then initially there are zero processes in s_k in both abstract and concrete executions (i.e. $init(s_k) = 0$).

Therefore, in either case, the value of $init(s_k)$ in concrete execution is less than or equal to that in abstract execution. Further, since both $leave(s_k, \gamma)$ and $enter(s_k, \gamma)$ depend on the source and destination states of processes executing various actions in γ, their value in concrete execution will be same as in the abstract execution. Hence, $init(s_k) + enter(s_k, \gamma) - leave(s_k, \gamma) \leq 0$ in the concrete execution, and there can be no process in state s_k after the occurrence of γ that can be chosen to execute α_k, which is a contradiction. $\square$

Case-B: σ is infinite. In this case, σ is of the form $\sigma_{pr}(\sigma_{sx})^\omega$. Here σ_{pr} and σ_{sx} are finite action sequences s.t., $(\sigma_{sx})^\omega$ represents an unbounded repetition of σ_{sx}, and the abstract configurations before and after each iteration of σ_{sx} are same in abstract execution. Let $S_{\sigma_{sx}} \subseteq S_{\mathcal{P}}$ denote the execution states from/to which processes move during an iteration of σ_{sx}. Then, we show that: *σ is non-spurious $\Leftrightarrow$ (i) $\sigma_{pr}\sigma_{sx}$ is non-spurious, and (ii) $\forall s \in S_{\sigma_{sx}} \cdot enter(s, \sigma_{sx}) = leave(s, \sigma_{sx})$.*

PROOF. **B.1 $\Leftarrow$:** Suppose conditions (i) and (ii) hold above. Let $\sigma' = \sigma_{pr}\sigma_{sx}$. From condition-(i) we get that σ' can be exhibited in a concrete system. In fact, by reusing the arguments from Case-A.1 described earlier, we get that σ' can be exhibited in a concrete system with initially $n_{p,\sigma'} \in \mathbb{N}$ processes for each process type p.

Further, condition-(ii) ensures that the number of processes residing in state $s \in S_{\sigma_{sx}}$ before and after each iteration of σ_{sx} are same. Hence, σ_{sx} can be repeated infinitely often in the concrete execution. This means $\sigma = \sigma_{pr}(\sigma_{sx})^\omega$ is also exhibited in a concrete execution with $n_{p,\sigma'}$ processes for each process type p

B.2 $\Rightarrow$: Assume that σ is non-spurious. Then $\sigma_{pr}\sigma_{sx}$ is also non-spurious (i.e. condition (i) holds), and it can be exhibited in a concrete execution. We use contradiction to show that condition (ii) also holds. Assume that there exists a state $s' \in S_{\sigma_{sx}}$ such that $enter(s', \sigma_{sx}) \neq leave(s', \sigma_{sx})$. Consider the following two cases: (a) $enter(s', \sigma_{sx}) < leave(s', \sigma_{sx})$, and (b) $enter(s', \sigma_{sx}) > leave(s', \sigma_{sx})$.

Case (a) above implies that after each iteration of suffix σ_{sx}, the number of processes in state s' will be strictly less than what it was before the occurrence of σ_{sx}. Hence, after a finite number of iterations of σ_{sx} in concrete execution, the number of processes in s' will become 0. Therefore, σ_{sx} cannot iterate infinitely often.

Case (b) above implies that after each iteration of suffix σ_{sx}, the number of processes in state s' will be strictly greater than what it was before the occurrence of σ_{sx}. Hence, the number of processes in s' will grow unboundedly as σ_{sx} is repeated infinitely often. If s' is a state of process type p, the number of processes in s' is no greater than the total number of processes of type p. However, in a concrete execution, the total number of processes of each process type is bounded. Thus, σ cannot occur in a concrete execution (i.e. it is spurious), which contradicts our assumption. $\square$

6.3 Abstraction Refinement

We now discuss an abstraction-refinement approach for eliminating spurious counter-examples.

Finite counter-example. Let $\sigma = \alpha_0 \ldots \alpha_n$ be a finite spurious counter-example such that, action α_i is executed by a process moving from state $s_i \in S_{\mathcal{P}}$ to state $s'_i \in S_{\mathcal{P}}$ in the abstract verification, i.e. $src(\alpha_i) = s_i$ and $dst(\alpha_i) = s'_i$. Recall that, $Pre(\sigma)$ is the set of all execution prefixes of σ, excluding σ itself. Since, σ is spurious, there exists a prefix $\gamma \in Pre(\sigma)$ such that $valid(s_{|\gamma|}, \gamma)$ is *false* (see Section 6.2, Case-A). We determine the smallest prefix $\sigma_m = \alpha_0 \ldots \alpha_{k-1}$ such that, in abstract execution: (i) $\neg valid(s_k, \sigma_m)$, where $k = |\sigma_m|$, and (ii) $\forall \gamma \in Pre(\sigma_m) \cdot valid(s_{|\gamma|}, \gamma)$. Since, $valid(s_k, \sigma_m)$ $(k = |\sigma_m|)$ is false, this implies $init(s_k) + enter(s_k, \sigma_m) - leave(s_k, \sigma_m) \leq 0$. Hence, after the occurrence of σ_m in abstract execution there can be no processes in state s_k. However, α_k is executed by a process from s_k after the occurrence of σ_m in abstract execution – this is only possible, if process count in s_k becomes unbounded (i.e. ω) during execution of σ_m. Assuming s_k is a state of process type p, its process count can become ω only if number of processes in s_k becomes cut_p (the cutoff number of p) during execution of σ_m. In order to prevent the process count in s_k from becoming ω in abstract execution, we determine the maximum number of processes that can reside in s_k during execution of σ_m, i.e. $max(s_k, \sigma_m) = max\{init(s_k) + enter(s_k, \gamma) - leave(s_k, \gamma) | \gamma \in Pre(\sigma_m)\}$. Consequently, we set $cut_p = max(s_k, \sigma_m) + 1$ to prevent the occurrence of spurious trace σ in subsequent abstract verification runs.

For illustration, consider the system model with a single process type p_1 as shown in Figure 2 and the LTL property $\neg(\alpha_0 \wedge X\alpha_1 \wedge XX\alpha_2)$, specifying that the trace $\alpha_0\alpha_1\alpha_2$ never occurs. As shown earlier in Section 2, with $cut_{p1} = 1$, σ can be exhibited in the abstract execution s.t., $src(\alpha_0) = (l_0, \epsilon)$, $dst(\alpha_0) = (l_1, \epsilon)$, $src(\alpha_1) = src(\alpha_2) = (l_1, \epsilon)$, $dst(\alpha_1) = (l_2, \epsilon)$ and $dst(\alpha_2) = (l_3, \epsilon)$. Our check finds σ to be spurious, as there exists $\sigma_m = \alpha_0.\alpha_1$ s.t. $valid(s_{|\sigma_m|}, \sigma_m)$ is *false*, where $s_{|\sigma_m|} = s_2 = src(\alpha_2) = (l_1, \epsilon)$. This is because, during execution of σ_m, we have $init(s_2) = 0$ and $enter(s_2, \sigma_m) = leave(s_2, \sigma_m) = 1$. Considering all prefixes of σ_m, we can easily find that $max(s_2, \sigma_m) = 1$. Then, if cut_{p_1} is set to 2, the execution sequence σ can no longer be exhibited in the abstract execution.

Infinite counter-example. Here $\sigma = \sigma_{pr}(\sigma_{sx})^\omega$. From Case-B in Section 6.2, σ is spurious if either–
(i) $\sigma_{pr}\sigma_{sx}$ is spurious. This case is similar to that of the finite spurious counter-example discussed in the preceding.
(ii) $\sigma_{pr}\sigma_{sx}$ is not spurious, but there exists a state s belonging to some process type p, from which a p-process executes one of the actions appearing in σ_{sx} s.t. $enter(s, \sigma_{sx}) \neq leave(s, \sigma_{sx})$ in abstract execution.

For case (ii), since suffix σ_{sx} is repeated infinitely often in the abstract execution of σ, the abstract configuration, and hence, the process counts in state s are the same during repeated execution of σ_{sx} in σ. As $enter(s, \sigma_{sx}) \neq leave(s, \sigma_{sx})$, this is only possible if the count of processes in s is approximated to ω sometime during the repeated execution of σ_{sx}. We consider two sub-cases.

(ii-a) The process count in s is ω at the beginning/end of every execution of σ_{sx} (in the abstract execution of σ).

(ii-b) The process count in s is a natural number n_0 at the beginning/end of every execution of σ_{sx} (in the abstract execution of σ). However, it grows to ω during execution of σ_{sx} (and shrinks back to n_0 before next execution of σ_{sx}).

For case (ii-a), the process count in s is ω at the beginning of the first execution of σ_{sx} while executing $\sigma = \sigma_{pr}(\sigma_{sx})^{\omega}$, that is, at the end of σ_{pr} itself. Our abstraction refinement sets the cutoff number of process type p to the maximum value of $1 + init(s) + enter(s,\gamma) - leave(s,\gamma)$, where $\gamma \in \{\sigma_{pr}\} \cup Pre(\sigma_{pr})$. This bounds the maximum number of processes in state s after execution of σ_{pr}.

For case (ii-b), our abstraction refinement similarly prevents the process count from becoming ω in s during the execution of σ_{sx} in σ. Here we set the cutoff number of process type p to $1 + max\{n_0 + enter(s,\gamma) - leave(s,\gamma) | \gamma \in Pre(\sigma_{sx})\}$.

6.4 Deriving a finite-state system for a non-spurious counter-example

Finally we discuss the derivation of a system with finite number of processes that can exhibit a given non-spurious counter-example trace σ. Thus, trace σ is obtained from our abstract verification with unbounded number of processes, and then shown to be non-spurious (as per our spuriousness check). We consider two cases.

(i) σ is *finite*. Then from Section 6.2, Case-A.1, we know that σ can be exhibited in a finite state system with initially $n_{p,\sigma} \in \mathbb{N}$ number of processes of type p. From Eqn. (4) page 19, when the number of processes of type p is unbounded, $n_{p,\sigma}$ is equal to $leave(s_p^{in}, \sigma)$ — the number of processes exiting the initial state of p while executing σ.

(ii) σ is *infinite* and hence is of the form $\sigma_{pr} \cdot (\sigma_{sx})^{\omega}$. Let $\sigma' = \sigma_{pr}\sigma_{sx}$. Since σ is non-spurious, from Section 6.2, Case-B.1, σ can be exhibited in a finite state system with initially $n_{p,\sigma'} \in \mathbb{N}$ number of processes of type p.

7. IMPLEMENTATION

SPIN [Holzmann 2003; 1997] is a popular open-source linear-time temporal logic (LTL) model checker for software verification. In this section, we describe our modifications involving process abstractions to SPIN, and for convenience call the modified version as SPIN++.

7.1 Abstract State Representation

In SPIN, the state of each active process in the system is maintained separately during verification. The information maintained corresponding to each process consists of its current control state and a valuation of its local variables. In addition, a unique process id is used to identify each process in the system.

We modify the default SPIN state representation by introducing abstraction over process identities in SPIN++, such that process instantiations of the same process-type (declared using keyword `proctype` in SPIN) are no longer distinguished based on their process ids. Moreover, we no longer maintain the state of each process separately. Instead, processes corresponding to the same process-type, say p, are grouped into **partitions** during execution. Each such partition is identified by a p-state in S_p, consisting of a control state

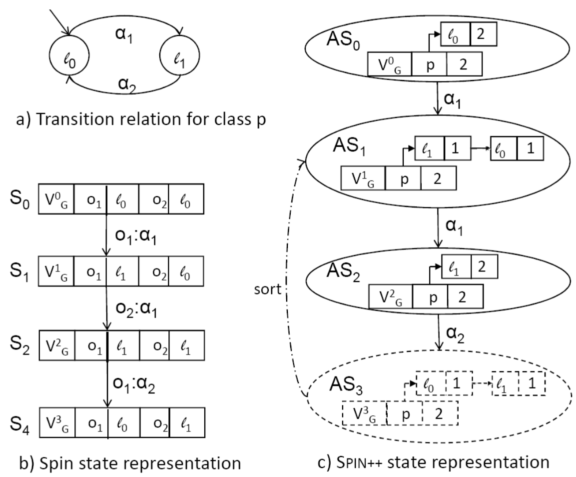

Fig. 7. State representation in SPIN and SPIN++

in TS_p and a valuation of p's local variables. Then, at runtime, corresponding to each process type we maintain a set of its partitions, and the number of processes currently residing in each partition. Note that, the partition set of a process type only contains partitions with at least one process in them; if the process count of a partition becomes zero after some action at run time, then this partition is removed from the partition set.

State Storage and Matching. In SPIN, each system state in the current exploration graph is stored for state matching during verification. In the SPIN state representation, the order of processes is fixed for all the states, and comparison of any two states is done byte by byte, with time complexity linear in the size of a system state. Consider the example shown in Figure 7(a), where type p has two actions α_1 and α_2, and no local variables (hence, p's execution state is characterized by its local control state). Two processes o_1 and o_2 of type p are created, and a sequence of actions $\sigma = \alpha_1\alpha_1\alpha_2$ occurs. This results in the generation of four global states as shown in Figure 7(b), with V_G^i ($i \in [0, 4]$) representing the valuations of global variables. We assume that α_1, α_2 do not modify global variables.

However, with process abstraction in SPIN++, although the order of active process types in each system state is fixed, the order of partitions within a process type p (representing p's execution states, i.e. a subset of S_p with non-zero processes) may vary. This occurs due to the addition or deletion of partitions, as processes move in or out[4] of partitions. Therefore, to be able to use the default SPIN state matching algorithm, we first sort the partitions corresponding to each process type before storing them in the state vector. Consider the same example in Figure 7(a). The abstract system states visited during the execution of σ are shown in Figure 7(c). Among the four abstract global states generated, two of them (namely, AS_1 and AS_3) differ only in the permutation of partitions of type p. This is due to the dynamic addition and deletion of partitions as illustrated. As shown, sorting the partitions in AS_3 results in state AS_1 and hence, AS_3 is not stored as a new state in the state space.

[4]A partition is deleted when its process count becomes zero.

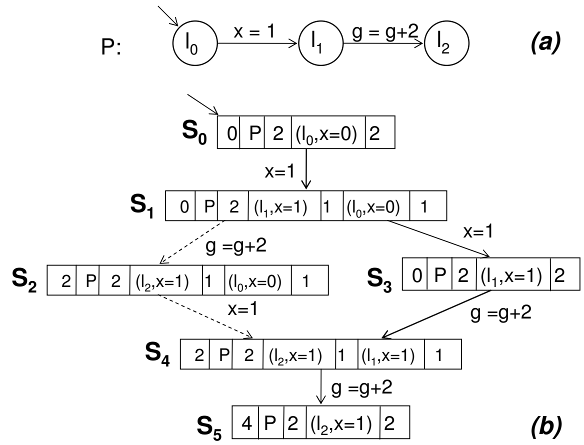

Fig. 8. Partial Order Reduction in SPIN++

7.2 Preservation of SPIN Optimizations

Optimization techniques in SPIN fall into two categories– (a) reducing the number of reachable system states that must be searched to verify properties (e.g. partial order reduction and statement merging), and (b) reducing the amount of memory needed to store each state (eg. collapse compression and bitstate hashing).

Partial Order Reduction. The partial order reduction and statement merging techniques are based on the knowledge of dependency relations among different transitions in a system model. In SPIN [Holzmann 2003], to avoid any run-time overheads, these dependency relations are computed off-line, before a model checking run is initiated. In our case, we only modify the state representation in SPIN without affecting the syntax or semantics of other operations. Hence, the dependency relations, and consequently the partial order reduction and statement merging are preserved with process abstractions in SPIN++. For illustration, consider a system consisting of a global variable g and a process type P with local variable x as shown in Figure 8(a). The state exploration graph for this example in SPIN++ with two instantiations of process type P is shown in Figure 8(b). Note that, for any two instantiations of P, transitions labeled $x = 1$ and $g = g + 2$ are mutually independent, and their different inter-leavings would lead to the same system state. For example, in Figure 8(b) the two paths –$S_1.S_2.S_4$ and $S_1.S_3.S_4$– between system states S_1 and S_4 are considered equivalent. Hence, with partial order reduction enabled in SPIN++, the dashed path ($S_1.S_2.S_4$) in Figure 8(b) is not explored.

Collapse compression. In addition, SPIN++ can also take advantage of state compression techniques such as collapse compression and bit-state hashing. Collapse compression addresses the state space explosion problem by dividing a system state into several "components". These "components" are then assigned a unique index number and stored separately. This technique tries to exploit the observation that most of the components of two distinct system states may be the same. When collapse compression is enabled in SPIN, the global data objects and each active process in a system state are identified as system components. For example, in Figure 7(c), the global state S_1 of original SPIN consists of three components: value of global variables, state of o_1 and state of o_2. In SPIN++, we consider the following as system "components" in a global system state — value of global variables, and the count of processes in each local state of each process type. For example, for the abstract global state AS_1 in SPIN++ as shown in Figure 7(c), the "components" would be – *global information*, and partitions $s_0 = (l_0, \epsilon)$, $s_1 = (l_0, \epsilon)$ of type P (ϵ denotes emptiness of local variables). In other words, each local state (with process count greater than 0) of a process type is considered as a system component in our state representation. This enables the designer to use SPIN's collapse compression optimization on our abstract state space.

Bitstate hashing. In SPIN, each system state is represented as a sequence of bits (i.e, a bitvector). With bitstate hashing enabled, a hash table containing single bit entries is used to store the visited states information. Further, a parameter k is used such that, k independent hash functions are applied to a system state, with each function pointing to an entry in the hash table. Then, if all the k entries in the hash table corresponding to some state are found to be 1, it indicates that the state has already been visited. Otherwise, the state has not been visited and any of the corresponding k bits that are 0 are set to 1. In SPIN the default value of k is 2 and can be set to other values using runtime options. Since several system states can map to the same hashtable entry, state space search with bitstate hashing may not be exhaustive. Of course, any counter-examples found can still be used for debugging. In our tool SPIN++, we also allow the designer the flexibility of using bitstate hashing. The only change is in how the bitvector representation of a system state is constructed. As mentioned earlier, a system state in the abstract state space consists of (a) the state of global variables and (b) process counts for all local states of all process types (for those local states where the process count is greater than 0). This state representation gets converted into a bitvector. The rest of the state space traversal — applying hash function(s) to the bitvector, looking up the hashtable, and storing 0/1 in a hash table entry depending on whether the state is visited – remains unchanged, allowing the designer to use bitstate hashing if he/she wants to.

8. EXPERIMENTS

In this section, we first describe our restrictions on PROMELA for system specification, and on LTL properties for property specification. We then discuss various experimental results involving the use of SPIN++ for verification. All our experiments were done on a Pentium-IV 3 GHz machine with 2 GB of main memory. We have made our parameterized checker available from `http://www.comp.nus.edu.sg/~abhik/SPIN++/`

8.1 Restrictions

Our proof method is applicable to verification of arbitrary LTL properties for any PROMELA model, subject to the following restrictions. Recall that PROMELA is the input language of the SPIN model checker [Holzmann 2003; 1997] which allows system modeling via concurrent processes communicating by shared variables and/or message passing.

Restrictions on PROMELA model. Below, we summarize our restrictions on model specifications described in PROMELA.

—Since we suppress the use of process ids in our abstraction, we disallow the use of special SPIN variables `_pid` and `_last`, which can refer to individual process ids. For the same reason, we avoid accessing or checking the value returned by a `run` statement (which

Table I. PROMELA modeling results

Example	# Global Vars	Process type	# Local Vars	# Local Control Loc.
CTAS	1	Client	0	26
		CM	2	77
		WCP	0	8
MOST	3	Env	2	6
		NS	0	5
		NM	5	37
Meta-lock	3	Handoff	0	4
		Shared Obj.	0	10
		Thread	1	7
Futurebus+	17	Cache	0	11

creates a process and returns the process id in PROMELA).

—Only channels of size 0 can be declared, i.e. communication via message passing is synchronous. In addition, we allow inter-process communication via shared variables.

Any PROMELA *model satisfying these two restrictions can be verified in our parameterized verification framework. Thus, the user can now model parameterized systems using a rich and popular modeling language like* PROMELA*, rather than having to construct FSMs for each process type. Note that dynamic process creation and annihilation is allowed in our system model.*

Restrictions on LTL property. Given a PROMELA model satisfying the above restrictions, we verify any LTL property with the following restrictions.

—Atomic propositions in the LTL property do not refer to process identifiers. For example we cannot have an atomic proposition of the form $pid == 1$ where pid is a local variable capturing process identifiers. This restriction stems from our count abstraction which does not keep track of process identifiers.

—Recall that our system model may also contain process-count variables (denoted as Var^ω), such that a variable $v \in Var^\omega$ is used for counting processes of a given type, say p, with its domain ranging over $[0, cut_p) \cup \{\omega\}$. Then, for LTL property specification, we restrict the boolean expressions involving a process-count variable v_p (which counts processes of type p) to be of the form v_p $\mathbf{Relop}$ c, such that $c \in [0, cut_p)$ and $\mathbf{Relop}$ is any relational operator. This restriction ensures deterministic evaluation of boolean expressions involving the process-count variables.

8.2 Examples Modeled

For experiments, we modeled the following four examples. In Table I, we summarize the key statistics of the PROMELA models for each of these examples.

The first example is a weather update controller, which is an important component of the *Center TRACON Automation System (CTAS)* automation tools developed by NASA for controlling air-traffic in large airports [CTA]. It consists of a central controller (CM), a weather-control panel (WCP), and several Client processes. Clients first get connected to the CM. Subsequently, all connected clients are updated with the latest weather information from WCP via CM.

The second example models part of the *Media Oriented Systems Transport (MOST)* protocol [MOS], which is a networking standard designed for interconnecting various multimedia components in automobiles. The main components consist of a network-manager (NM) and several network-slaves (NS). We model the network management part of this protocol which ensures secure communication between various applications in the MOST network.

The third example is the Java-metalock [Agesen et al. 1999] protocol, a distributed algorithm ensuring mutually exclusive access to a shared object (S) among arbitrary number of Java Threads. A hand-off process (H) handles the race between the releasing thread and several threads waiting to acquire S. If the object S is *not-busy* (i.e, no thread currently owns it), then a requesting thread is immediately granted access to it. Otherwise, S is *busy* and the owner thread releases access of S to one of the requesting threads via the hand-off process.

As the final example, we modeled the cache coherence part of the IEEE Futurebus+ Protocol [IEEE Computer Society 1992], where we restrict our model to contain only a single bus segment with one shared-memory module and multiple caches.

8.3 Reachability Analysis

The initial set of experiments involved doing a reachability analysis for the examples modeled using both SPIN and SPIN++. The main aim of these experiments was to– (i) compare the run-time and memory usage between SPIN and SPIN++, and (ii) experimentally evaluate the benefits of partial order reduction and collapse-compression optimizations in SPIN++. For each example we created several versions differing in number of processes.

The experimental results for state space exploration are shown graphically in Figure 9. As we can observe, SPIN++ clearly outperforms SPIN by a significant margin as the number of processes in the system increases. Moreover, with the increasing number of processes, while almost linear growth is observed for both run-time and memory usage for SPIN++, the growths are exponential in case of SPIN. The results with ω number of processes using SPIN++ are also shown (the last entry in these graphs).

In Figure 10, we show the reduction in the number of states explored due to partial order reduction (POR) for MOST and Java Meta-lock protocols in SPIN++. We are able to take significant advantage of POR using SPIN++ on these two protocols. The results for CTAS and Futurebus+ are omitted here, as they do not exhibit a significant improvement with POR enabled. For CTAS, there is almost no concurrency among Client processes, and they interact with the controller in a synchronous manner one by one; for Futurebus+ protocol, most transitions involve modification of shared global variables (used for communication) and hence, are not independent.

Finally, with collapse compression enabled in SPIN++, we could verify larger models which would otherwise run out of memory (see Table II). For example, *CTAS with 200 Clients* and *MOST with 700 Slaves* cannot be explored without using collapse compression. Both these instances ran out of memory as indicated by **O.M.** In case of Futurebus+ protocol, we observe less memory reduction as compared with the other two protocols. This is because, its model contains no local variables and a process state only consists of a control location. Hence, no significant reduction can be obtained using collapse compression.

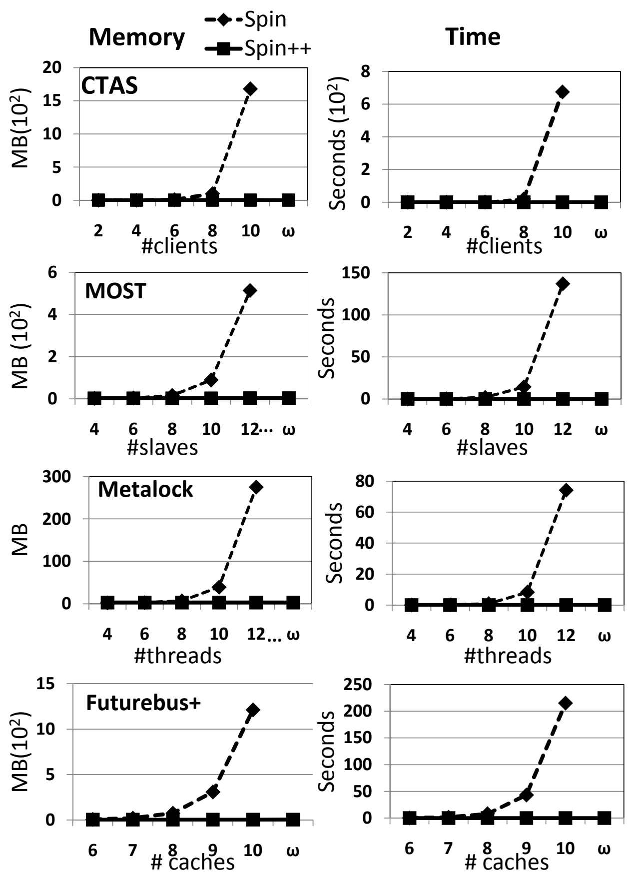

Fig. 9. State space exploration results.

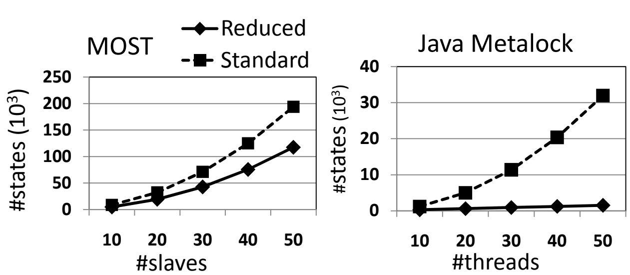

Fig. 10. Partial order reduction in SPIN++.

Table II. Collapse compression in SPIN++.

Example (Proc. Type)	# of Proc.	No Collapse Compr.		Collapse Compr.	
		Mem(MB)	Time(s)	Mem(MB)	Time(s)
CTAS (Clients)	100	489.62	39.41	236.60	64.95
	200	**O.M.**	–	1512.67	1037.00
MOST (Slaves)	350	699.04	40.04	298.87	57.36
	700	**O.M.**	–	1367.03	354.44
Metalock (Threads)	1.5×10^5	653.29	56.27	479.93	86.67
	3×10^5	1304.38	119.67	959.40	190.19
Futurebus+ (Caches)	50	26.88	2.07	26.47	3.24
	100	190.42	16.74	186.94	29.16

O.M. indicates Out of Memory.

8.4 Verification of LTL properties

We verified our examples against some interesting LTL properties using SPIN++. Here, we consider one property for each example and present the verification results. The verification results for our examples appear in Table III.

For CTAS, the weather-panel (WCP) is *disabled* each time there is an interaction initiated between Clients and the central-controller (CM), and is *enabled* once the interaction is over. Hence, for CTAS we specify a liveness property: *whenever WCP is disabled, it will eventually be enabled* (property **P1**, Tab. III).

In case of MOST, we verify the property — whenever network-manager receives a valid registration message from any slave, its registry gets updated (property **P2**, Tab. III).

For the Java meta-lock protocol, we verify the invariant property that at most one thread can own a shared object at any point of time (property **P3**, Tab. III).

For Futurebus+, we verify the property — if a cache holds an *exclusively-modified* copy of data, then no other caches can hold an *exclusively-unmodified* copy of the same data (property **P4**, Tab. III).

As we can observe from Table III, all examples satisfied the respective properties with initially a *concrete* number of processes for various process types. For these experiments, the choice of cutoff number is not an issue — since the number of processes is fixed initially and there is no unbounded process creation in these examples. In other words, the process counts never become ω, thus avoiding any spurious behaviors during abstract verification.

For experiments with an unbounded (ω) number of processes for some process type, spurious counter-examples may be reported by abstract verification. During abstract verification, the CTAS and MOST protocols satisfied their respective properties, with a cutoff number 1. For Futurebus+ protocol, a finite counter-example of length 34 was obtained, with cutoff number 1. However, our spuriousness check procedure (see Sec. 6.2) found this counter-example to be spurious. Using our abstraction-refinement approach (see Sec. 6.3), we obtained a new cutoff number of 2 for process-type Cache. Subsequently, verification with an unbounded number of caches in Futurebus+ protocol also succeeded. For the Java meta-lock protocol also, verification of mutual exclusion of shared object access by unbounded number of threads succeeded.

9. RELATED WORK

Verification of parameterized systems is undecidable [Apt and Kozen 1986]. There are two possible remedies to this problem: either we look for restricted subsets of parameterized

Table III. LTL property verification in SPIN++.

Example	# Proc.	Mem (MB)	Time	Result	Cutoff
	P1: $\mathbf{G}(disabled \Rightarrow \mathbf{F}(\neg disabled))$				
CTAS	10 Clients	4.97	0.23s	$\surd$	—
	20 Clients	14.09	1.23s	$\surd$	—
	ω Clients	3.31	0.06s	$\surd$	1
	P2: $\mathbf{G}(regValid \Rightarrow \mathbf{F}(regUpdtd))$				
MOST	10 Slaves	4.27	0.08s	$\surd$	—
	20 Slaves	7.96	0.28s	$\surd$	—
	ω Slaves	3.31	0.02s	$\surd$	1
	P3: $\mathbf{G}(abs\_Thread\_isOwner \leq 1)$				
Java Metalock	50 Threads	3.28	0.03s	$\surd$	—
	100 Threads	3.41	0.05s	$\surd$	—
	ω Threads	3.26	0.01s	$\surd$	2
	P4: $\mathbf{G}(abs\_Cache\_em > 0 \Rightarrow abs\_Cache\_eu == 0)$				
Futurebus+	10 Caches	3.36	0.03s	$\surd$	—
	20 Caches	4.81	0.17s	$\surd$	—
	ω Caches	3.62	0.09s	$\surd$	2

systems for which the verification problem becomes decidable, or we look for sound but not necessarily complete methods.

The first approach tries to identify a *restricted subset* of parameterized systems and temporal properties, such that if a property holds for a system with up to a certain number of processes, then it holds for every number of processes in the system. Moreover, the verification for the reduced system can be accomplished by model checking. Systems that are verified with this approach include systems with a single controller and arbitrary number of user processes [German and Sistla 1992], rings with arbitrary number of processes communicating by passing tokens [Emerson and Namjoshi 2003; Emerson and Kahlon 2004], systems formed by composing an arbitrary number of identical processes in parallel [Ip and Dill 1999], and systems formed by unbounded processes of several process types where the communication mechanism between the processes is restricted to conjunctive / disjunctive transition guards [Emerson and Kahlon 2000].

The sound but incomplete approaches include methods based on synthesis of invisible invariant (*e.g.*, [Fang et al. 2006]) which can be viewed as a combination of assertion synthesis techniques with abstraction for verification; methods based on network invariant (*e.g.*, [Lesens et al. 1997]) that relies on the effectiveness of a generated invariant and the invariant refinement techniques; regular model checking [Jonsson and Saksena 2007; Kesten et al. 1997] that requires acceleration techniques. Compositional proof methods have been studied in [Basu and Ramakrishnan 2003], while explicit induction based proof methods for parameterized families have been discussed in [Roychoudhury and Ramakrishnan 2001].

Parameterized verification of extended system models having data variables with unbounded domains have been studied (*e.g.*, see [Abdualla et al. 2007; Betin-Can et al. 2005]). In comparison, our focus has been to link up our work with a mature system modeling and verification tool (we have chosen SPIN for this purpose). We do not define a new modeling language, whatever features are supported in PROMELA can appear in our system models. This frees the user from learning a new modeling language or tool for parameterized system verification.

The works closest to ours are the methods based on *counter abstraction* (*e.g.*, [Delzanno 2000; Pong and Dubois 1995; Pnueli et al. 2002]). These works also employ process count abstraction. The verification of safety properties is discussed in [Delzanno 2000], the verification of liveness properties is addressed in [Pnueli et al. 2002]. Unlike previous works which perform one run of verification in the abstract state space, we use abstraction refinement to gradually discover cutoff numbers. To the best of our knowledge, ours is the first work to use abstraction refinement for parameterized system verification.

Murphi is an explicit-state model checker with extensions to support parameterized system verification [Ip and Dill 1999]. However, parameterized verification in Murphi is restricted to systems where only one process type has unbounded number of processes. SPIN is also an explicit-state model checker, whose enhancements have been studied for combatting state space explosion. Different techniques have been proposed to remedy state space explosion, including – compositional verification methods as studied in [Pasareanu and Giannakopoulou 2006], and Symmetric SPIN, a symmetry reduction package for PROMELA programs [Bosnacki et al. 2000]. These approaches are not applicable to parameterized verification of systems with *unbounded* number of processes.

Counter-example guided abstraction refinement has earlier been studied for verification of large finite-state or infinite-state systems [Clarke et al. 2003; Beyer et al. 2007; Chaki et al. 2003; Ball and Rajamani 2002]. A common thread among these works is that they abstract the domains of variables which appear in the program being verified. However, if the program being verified is infinite-state because of having infinitely many processes — it is not clear how to employ abstraction refinement methods. In this paper, we have developed an abstraction refinement method on the previously studied counter-abstraction [Delzanno 2000; Pnueli et al. 2002], and used it for parameterized system verification.

10. DISCUSSION

In this paper, we have presented a pragmatic approach for verifying parameterized systems — concurrent systems with large / unbounded number of behaviorally similar processes. Our checker modifies the verification engine of the popular model checker SPIN by introducing process count abstractions. Since we do not modify PROMELA, the input language of SPIN, there is no learning curve involved for users. We can detect spurious counter-examples (introduced by our abstraction) and eliminate them via abstraction-refinement. Thus abstraction refinement seeks to discover the cutoff number of processes to be kept track of, for each process type with unboundedly many processes. For a non-spurious counter-example trace, we can construct a finite-state system which exhibits the same trace. We have made our parameterized checker available from `http://www.comp.nus.edu.sg/~abhik/SPIN++/`

In future, we plan to augment our parameterized verifier with data-abstraction refinement methods, which have been implemented in sequential software verifiers such as BLAST [Beyer et al. 2007], MAGIC [Chaki et al. 2003] and SLAM [Ball and Rajamani 2002]. This will allow us to verify systems with unbounded number of similar processes, as well as data variables with unbounded domains.

REFERENCES

Center-tracon automation system (CTAS) for air traffic control. `http://ctas.arc.nasa.gov/`.

Media oriented system transport (MOST). `http://www.mostcooperation.com/`.

ABDUALLA, P., DELZANNO, G., AND REZINE, A. 2007. Parameterized Verification of Infinite-state Processes with Global Conditions. In *CAV*.

AGESEN, O., DETLEFS, D., GARTHWAITE, A., KNIPPEL, R., RAMAKRISHNA, Y., AND WHITE, D. 1999. An efficient meta-lock for implementing ubiquitous synchronization. In *OOPSLA*.

APT, K. AND KOZEN, D. 1986. Limits for automatic verification of finite-state systems. *Information Processing Letters 15*.

BALL, T. AND RAJAMANI, S. 2002. The SLAM Project: Debugging System Software via Static Analysis. In *POPL*.

BASU, S. AND RAMAKRISHNAN, C. 2003. Compositional analysis for verification of parameterized systems. In *TACAS*.

BETIN-CAN, A., BULTAN, T., LINDVALL, M., LUX, B., AND TOPP, S. 2005. Application of design for verification with concurrency controllers to air traffic control software. In *ASE*.

BEYER, D., HENZINGER, T., JHALA, R., AND MAJUMDAR, R. 2007. The Software Model Checker BLAST: Applications to Software Engineering. *Int. Journal on Software Tools for Technology Transfer*.

BOSNACKI, D., DAMS, D., AND HOLENDERSKI, L. 2000. Symmetric spin. In *SPIN*. 1–19.

CHAKI, S., CLARKE, E., GROCE, A., JHA, S., AND VEITH, H. 2003. Modular verification of software components in c. In *ICSE*.

CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *JACM 50,* 5.

DELZANNO, G. 2000. Automatic Verification of Parameterized Cache Coherence Protocols. In *CAV*.

EMERSON, E. AND KAHLON, V. 2000. Reducing model checking of the many to the few. In *CADE*.

EMERSON, E. AND KAHLON, V. 2004. Parameterized model checking of ring-based message passing systems. In *CSL*.

EMERSON, E. AND NAMJOSHI, K. 2003. On Reasoning about rings. *Intl. Jnl. on Foundations of Computer Science (IJFCS) 14,* 4.

FANG, Y., MCMILLAN, K., PNUELI, A., AND ZUCK, L. D. 2006. Liveness by Invisible Invariants. In *FORTE*.

GERMAN, S. AND SISTLA, A. 1992. Reasoning about systems with many processes. *Journal of the ACM 39*.

HOLZMANN, G. 2003. *The SPIN model checker: Primer and Reference Manual*. Addison Wesley.

HOLZMANN, G. J. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering 23,* 5 (May), 279–295.

IEEE COMPUTER SOCIETY. 1992. *IEEE Standard for Futurebus+, Logical Protocol Specification*.

IP, C. AND DILL, D. 1999. Verifying Systems with Replicated Components in Murφ. *Formal Methods in System Design 14,* 3.

JONSSON, B. AND SAKSENA, M. 2007. Systematic Acceleration in Regular Model Checking. In *CAV*.

KESTEN, Y., MALER, O., MARCUS, M., PNUELI, A., AND SHAHAR, E. 1997. Symbolic model checking with rich assertional languages. In *CAV*.

LESENS, D., HALBWACHS, N., AND RAYMOND, P. 1997. Automatic Verification of Parameterized Linear Networks of Processes. In *POPL*.

MANNA, Z. AND PNUELI, A. 1991. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag.

PASAREANU, C. AND GIANNAKOPOULOU, D. 2006. Towards a compositional SPIN. In *SPIN*.

PNUELI, A., XU, J., AND ZUCK, L. 2002. Liveness with (0,1,infinity)-counter abstraction. In *CAV*.

PONG, F. AND DUBOIS, M. 1995. A new approach for the verification of cache coherence protocols. *IEEE Transacations on Parallel and Distributed Systems 6,* 8.

ROYCHOUDHURY, A. AND RAMAKRISHNAN, I. 2001. Automated inductive verification of parameterized protocols. In *CAV*.