# Model Checking Parameterized Process Classes

**Liu Shanshan**

**(School of Computing, National University of Singapore)**

**A THESIS SUBMITTED**

**FOR THE DEGREE OF MASTER OF SCIENCE**

**DEPARTMENT OF COMPUTER SCIENCE**

**NATIONAL UNIVERSITY OF SINGAPORE**

**July 2009**

# ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my supervisor, A/P. Abhik Roychoudhury, for the continuous support of my study and research, for his valuable guidance, motivation and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

Besides my supervisor, I would like to thank Prof. P.S. Thiagarajan for his encouragement and valuable discussion on my research. My sincere thanks also goes to A/P. Dong Jing Song, for his insightful comments on the project.

I thank my fellow labmates and friends: Ankit Goel, Ju Lei, Liang Yun, Nguyen Dang Kathy, Ioana Cutcutache, Sim Joon Edward, Sun Zhenxing, VIVY Suhendra, Qi Dawei, Wang Chundong, for their help in various ways of my research and personal life and for the fun we have had together. In particular, I am grateful to my senior Ankit Goel for the discussions, valuable comments and the sleepless nights we were working together.

Last but not the least, I would like to thank my family for their love, encouragement and understanding.

# TABLE OF CONTENTS

# SUMMARY

Systems consisting of large (or even unbounded) number of behaviorally similar processes communicating with each other are known as *parameterized systems*. Such systems are common in distributed computing and real-life software systems. Verifying properties for such systems involves reasoning about unboundedly many processes and hence cannot be accomplished directly by model-checking.

In this thesis, we present an abstraction refinement based verification framework for parameterized systems. We enhance the well-known SPIN model checker with process count abstraction to develop a time/memory efficient Linear-time Temporal Logic (LTL) verifier for parameterized systems. We also developed methods to automatically detect spurious counter-examples and refine the abstraction. The usability / scalability of our checker is demonstrated via the modeling and automated verification of several real-life parameterized control systems and protocols.

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

## 1.1 Background and Motivation

Systems that utilize cache-coherence or telecommunication protocols are common in distributed computing. Such systems usually consist of a large number of processes of the same type communicating with each other, or with processes of the other types. The number of processes is usually unknown prior to system deployment, which poses difficulty in verification and validation of system correctness.

Among the existing approaches, model checking has been advertised as a promising automated technique for ensuring correctness of complex distributed software systems. However, modeling of systems with (potentially) unbounded number of processes results in infinite-state system, which is not suitable for explicit model-checking. Moreover, fixing the number of processes to a constant (i.e., cutoff number) requires reasoning about that the restricted system can actually exhibit all valid behaviors of the actual system with unbounded number of processes.

Systems with unbounded number of processes are known as *parameterized systems*. A parameterized system usually consists of finite number of process types, each of which may contain unbounded number of instances. Verifying that a parameterized system satisfies certain specification entails proving that no matter how many number of processes participating in the system, the specification is satisfied.

In general, automated verification of parameterized system is undecidable [4]. Existing techniques for parameterized verification usually focus on developing their own verification procedure, which is not only hard for third-party evaluation, but also difficult to scale to an efficient and effective checker for real-life systems. The problems need to be addressed are whether : (1) the verification method is supported by a powerful modeling language to describe non-trivial control systems and protocols; (2) the checking is time/memory efficient; (3) it supports any spurious counter-example detection techniques and its refinement; (4) the analysis of non-spurious traces leads to any finite system that exhibits the same trace.

## 1.2 Objective and Scope

Our research aims to address the above mentioned problems by developing a usable and efficient abstraction-refinement based automated verification framework for concurrent parameterized systems. Our abstraction deals 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. Moreover, for non-spurious counter-examples, we also developed heuristics to determine a small,

finite system that exhibits the same trace.

In terms of implementation, we modify the internals of the well-known SPIN model checker [19] to integrate our proof method. Hence, any designer familiar with SPIN and its input modeling language PROMELA may easily adapt our method to verify parameterized systems. Moreover, we take advantage of the powerful optimization (such as partial order reduction, bitstate hashing, etc.) inside SPIN to develop a time and memory efficient model checker for parameterized systems.

## 1.3 Thesis Organization

The rest of this thesis is organized as following: we survey related verification techniques in Chapter 2, followed by an overview of our abstraction framework in Chapter 3. We then proceed to discuss the concrete and abstract system modeling in Chapter 4 and 5. The verification procedure, including our spuriousness detection and abstraction refinement techniques, are presented in Chapter 6. The details of how we modify the internals of SPIN are presented in Chapter 7, while experiments on several real-life software systems are discussed in Chapter 8. Finally, we conclude this thesis in Chapter 9.

# CHAPTER 2

# BACKGROUND ON SYSTEM VERIFICATION

In this Chapter, we review related state-of-art research on parameterized system verification, abstraction refinement and the SPIN model-checker.

## 2.1 Parameterized System Verification

Verification of parameterized systems is undecidable [4]. There are two possible remedies to this problem: either we look for restricted subsets of parameterized 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 [17], rings with arbitrary number of processes communicating by passing tokens [15, 14], systems formed by composing an arbitrary number of identical processes in parallel [21], and systems formed by unbounded processes of several process types where the communication mechanism between the processes is restricted to conjunctive / disjunctive transition guards [13].

The sound but incomplete approaches include methods based on synthesis of invisible invariant (*e.g.*, [16]) which can be viewed as a combination of assertion synthesis techniques with abstraction for verification; methods based on network invariant (*e.g.*, [25]) that relies on the effectiveness of a generated invariant and the invariant refinement techniques; regular model checking [22, 23] that requires acceleration techniques. Compositional proof methods have been studied in [6], while explicit induction based proof methods for parameterized families have been discussed in [29].

Among the above mentioned works, we emphasize on the *counter abstraction* (*e.g.*, [12, 28, 27]), which is closest to our current research work. These works also employ process count abstraction. The verification of safety properties is discussed in [12], the verification of liveness properties is addressed in [27].

Parameterized verification of extended system models having data variables with unbounded domains have been studied in [24, 10, 7, 30]. These approaches combine counter abstraction with data abstraction or linear-arithmetic constraints.

## 2.2 Counter-Example Guided Abstraction Refinement

Counter-example guided abstraction refinement has earlier been studied for verification of large finite-state or infinite-state systems [11, 8, 9, 5]. The common approach in these works is to abstract the variable domains based on the control flow predicates in the program, so that the state space are partitioned into different abstract states. When a counter-example is generated, it will be checked whether it corresponds to a concrete counter-example in the original system. If so, a program error is found;

otherwise, the predicates will be refined and the verification will start, either from the very beginning or from point where the previous predicate becomes not-so-precise (*Lazy Abstraction* [8]). This technique is mostly used in analysis of sequential programs, and systems with finite number of processes. Hence, if a system being verified is infinite-state due to unbounded number of processes, it is not clear how to employ the above mentioned abstraction refinement methods.

## 2.3 SPIN: The Model Checker

SPIN is a generic verification system that supports the design and verification of asynchronous process systems. SPIN verification models are focused on proving the correctness of *process interactions*, and they attempt to abstract as much as possible from internal sequential computations. Process interactions can be specified in SPIN with rendezvous primitives, with asynchronous message passing through buffered channels, through access to shared variables, or with any combination of these. As a formal methods tool, SPIN aims to provide [19]:

1. an intuitive, program-like notation for specifying design choices unambiguously, without implementation detail;

2. a powerful, concise notation for expressing general correctness requirements;

3. a methodology for establishing the logical consistency of the design choices from 1 and the matching correctness requirements from 2.

SPIN accepts design specification written in the verification language PROMELA(a Process Meta Language), and it accepts correctness claims specified in the syntax

of standard Linear Temporal Logic (LTL). Models specified in PROMELA are always required to be bounded and have only countably many distinct behaviors. All verification systems have physical limitations that are set by problem size, machine memory size, and the maximum runtime that the user is willing, or able, to endure. These constraints are an often neglected issue in formal verification. SPIN addresses this issue by offering some complexity management techniques.

The basic structure of the SPIN model checker is illustrated in Figure 2.1. The typical mode of working is to start with the specification of a high level model of a concurrent system, or distributed algorithm, typically using SPIN's graphical front-end XSPIN. After fixing syntax errors, one can choose to perform interactive simulation, or generate an optimized on-the-fly verification program from the high-level specification. This verification program is compiled, with various compile-time options for the types of reduction algorithms to be used, and executed. If any counter examples to the correctness claims are detected, the error trail can be fed back into the simulator and inspected in detail to establish, and remove, its cause.

In SPIN, the description of a concurrent system in PROMELA consists of one or more user-defined process templates, or *proctype* definitions, and at least one process instantiation. The templates define the behavior of different types of processes. Any running process can instantiate further asynchronous processes, using the process templates.

SPIN translates each process template into a finite automata. The global behavior of the concurrent system is obtained by computing an asynchronous interleaving product of automata, one automaton per asynchronous process behavior. The resulting

global system behavior is itself again represented by an automaton. This interleaving product is often referred to as the *state space* of the system, and, because it can easily be represented as a graph, it is also commonly referred to as the global *reachability graph*.

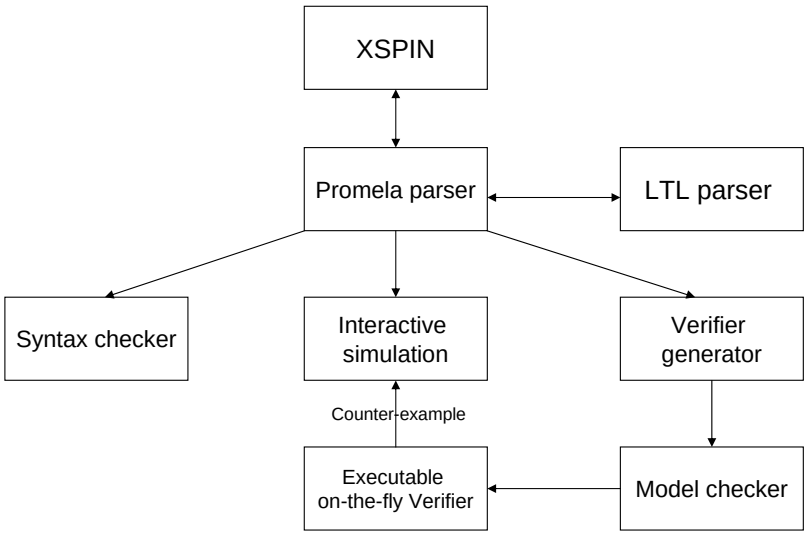


**Figure 2.1:** The structure of SPIN simulation and verification [19]

In SPIN, a correctness claim is to formalize erroneous system behaviors, i.e, behaviors that are *undesirable*. The verification process then either proves that such behaviors are impossible or it provides detailed examples of behaviors that match. To perform verification, SPIN takes a correctness claim that is specified as a temporal logic formula, converts that formula into a Büchi automaton, and computes the *synchronous* product of this claim and the automaton representing the global state space. The result is again a Büchi automaton. If the language accepted by this automation is empty, this means that the original claim is not satisfied for the given

system. If the language is nonempty, it contains precisely those behaviors that satisfy the original temporal logic formula.

Like other model checking techniques, SPIN's verification procedure is based on some graph traversal methods. Two kinds of traversal methods available in SPIN are depth-first search and breadth-first search. In the worst case, the global reachability graph has the size of the Cartesian product of all component systems. Although, in practice, the size of the global reachability never approaches the worst case size, the reachable portion of the Cartesian product can also easily become prohibitively expensive to construct exhaustively. A number of complexity management techniques have been developed to combat this problem. These techniques include partial order reduction, state compression and Bit-State hashing.

# CHAPTER 3

# VERIFICATION FRAMEWORK OVERVIEW

We have developed an abstraction-refinement based automated proof method for parameterized systems. The systems of interest consist of finitely many process types, each of which (may) have unbounded number of processes executing the same program. Our abstraction and its refinement, which greatly assembles the *abstract-verify-refine* loop, deal with the number of processes in the system.

An outline of our verification framework appears in Figure 3.1. 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

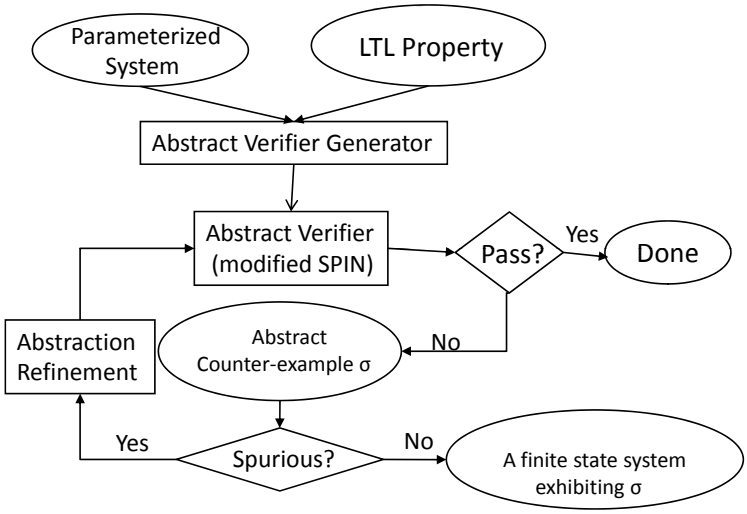


**Figure 3.1:** Verification Framework

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 [4], the abstraction-refinement loop shown in Figure 3.1 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 3.2, where for $i \in [0, 3]$, $l_i$ represents a control location, and $\alpha_0 - \alpha_2$ 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 $\alpha_0$. 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 $\omega$ during abstract verification. Further, a user-provided $p_1$ specific cutoff parameter (called $cut_{p_1}$) is used, such that $\omega$ 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 $\omega$, *and* (b) if a $p_1$ process moves out of a state with currently $\omega$ number of processes, there remains either $\omega$ 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 $\omega$ (denoting one or more processes in $l_i$). Abstract verification returns a counter-example trace, which is $\sigma$ itself. The number of processes in different states during abstract execution of $\sigma$ are shown in the following.

| Control state | Number of processes ($cut_{p_1} = 1$) | | | |
|---|---|---|---|---|
| | Initially | After $\alpha_0$ | After $\alpha_1$ | After $\alpha_2$ |
| $l_0$ | $\omega$ | $\omega, 0$ | $\omega, 0$ | $\omega, 0$ |
| $l_1$ | 0 | $\omega$ | $\omega, 0$ | $\omega, 0$ |
| $l_2$ | 0 | 0 | $\omega$ | $\omega$ |
| $l_3$ | 0 | 0 | 0 | $\omega$ |

However, it is easy to see that the counter-example trace $\sigma$ cannot be exhibited in any concrete system. At least two occurrences of $\alpha_0$ are required for $\alpha_1$ and $\alpha_2$ to be executed subsequently. Hence, $\sigma$ is spurious. Trace $\sigma$ can be exhibited in the abstract system because after a single occurrence of $\alpha_0$, the process count in $l_1$ becomes $\omega$ (since $cut_{p_1} = 1$). Consequently, both $\alpha_1$ and $\alpha_2$ 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

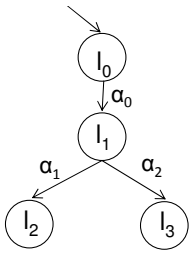


**Figure 3.2:** Example transition system for a process type $p_1$.

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 $\omega$ (denoting more than one process in $l_i$). Now, after a single occurrence of $\alpha_0$ the process count at $l_1$ will become 1 (and not $\omega$), which is not sufficient to execute both $\alpha_1$ and $\alpha_2$. As a result, $\sigma$ 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 $\sigma'$. However, unlike above, $\sigma'$ is not spurious and can be exhibited in a concrete system. Further, we can easily see that $\sigma'$ 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.

# CHAPTER 4

# 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* $\Sigma$ with $\alpha$, $\beta$ ranging over $\Sigma$. Actions in $\Sigma$ represent basic PROMELA statements, such that an action in $\Sigma$ 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 $Pre_\alpha$ specifying a boolean condition to be satisfied by a system state for executing $\alpha$, and (b) a post-condition $Post_\alpha$ capturing the system state update upon execution of $\alpha$.

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 4.1** (System Model)**.** *A* **system model** *is a structure*

$$\mathcal{S} = (Var_G, v_{in}^g, TS_{\mathcal{P}})$$

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

$$TS_{\mathcal{P}} = \{TS_p = (L_p, \rightarrow_p, l_{in}^p, Var_p, v_{in}^p)\}_{p \in \mathcal{P}}$$

*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,*
- *$l_{in}^p \in L_p$ is the initial control state of p, and*
- *$Var_p$ is the set of local variables in p, and $v_{in}^p$ is their initial valuation.*

Consider the example in Figure 3.2, 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 3.2. The initial state corresponding to process type $p1$ is $s^{p1}_{in} = (l_0, \epsilon)$, where the variable valuation part is empty (represented as $\epsilon$) 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 4.2** (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 3.2, 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 $\epsilon$ 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, $\epsilon$). Then, the initial configuration in this case is given by: $(\epsilon, M^{in})$, 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 $\Sigma$. 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 $I : S_{\mathcal{P}} \rightarrow 2^{OBJ}$ 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 $\alpha$ does not create new process, then $M' = I$, (ii) Otherwise, suppose by executing $\alpha$, 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 $\alpha$. The transition relation for the concrete execution $\hookrightarrow \subseteq CFG \times \Sigma \times CFG$ is defined as follows.

**Definition 4.3** (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. *$(l, \alpha, l') \in \rightarrow_p$ is a transition in $TS_p$.*

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

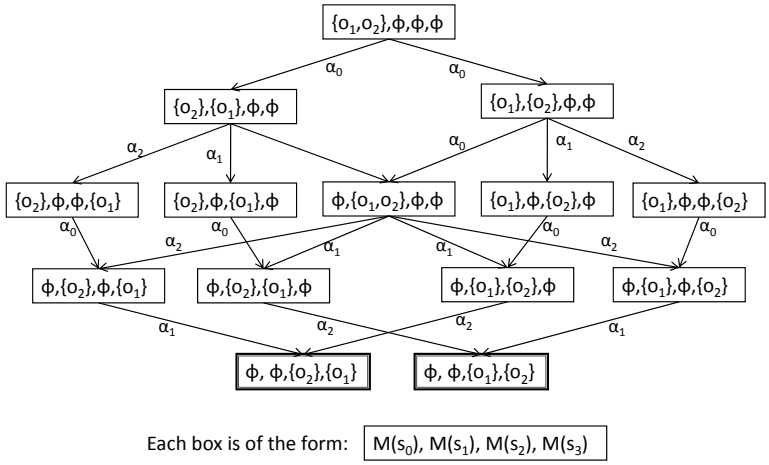


**Figure 4.1:** Sample Concrete Transition Relation

3. *$v$ and $v_g$ satisfy the pre-condition $Pre_\alpha$.*

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[1] *event, then $\alpha$ 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 3.2, we present its state exploration graph depicting all reachable concrete configurations in Figure 4.1. 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,

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

for $i = 0, 1, 2, 3$, $s_i = (l_i, \epsilon)$, where $l_i$ is the local control state of $p1$ and $\epsilon$ 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 $\alpha_0$, following by the execution of $\alpha_0$ from the other process, or the execution of either $\alpha_1$ or $\alpha_2$ 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.

# CHAPTER 5

# STATE SPACE ABSTRACTION

## 5.1 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 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 $\omega$.

For a process type $p$ with initially fixed number of processes, and no dynamic

process creation – the process counts never become $\omega$ 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 5.1** (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 $\Sigma$. 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

$$\begin{aligned} I_a(s_1) &= \begin{cases} M_a(s_1) - 1, & \text{if } M_a(s_1) < cut_p \\ cut_p - 1 \textbf{ or } \omega, & \text{if } otherwise. \end{cases} \\ I_a(s_2) &= \begin{cases} M(s_2) + 1, & \text{if } M(s_2) < cut_p - 1 \\ \omega, & \text{otherwise .} \end{cases} \\ I_a(s) &= M(s), \text{ for } s \in S_{\mathcal{P}} \backslash \{s_1, s_2\} \end{aligned} \tag{5.1}$$

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

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

- for state $s_q$,

$$M'_a(s_q) = \begin{cases} I_a(s_q) + 1, & \text{if } I_a(s_q) < cut_q - 1 \\ \omega, & \text{otherwise.} \end{cases} \tag{5.2}$$

- for $s \in S_{\mathcal{P}} \backslash \{s_q\}$, $M'_a(s) = I_a(s)$.

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 $\alpha$.

Note that, when there are $\omega$ processes in the source state $s_1$ at configuration $C^a = (v, M^a, D^a)$ (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. (5.1)). If $C^{a'} = (v', M^{a'}, D^{a'})$ 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) $\omega$, 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 $C^a$. Similar arguments apply to $D^a(m_1)$ and $D^a(m_2)$ when there are $\omega$ number of channels with contents $m_1$. 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 5.2** (Abstract Transition Relation $\hookrightarrow_a$)**.** *Let $\mathcal{S} = (Var_G, v_{in}^g, TS_{\mathcal{P}})$ be a system model, $C_a = (v_g, M_a)$, $C'_a = (v'_g, M'_a) \in CFG_{abs}$ 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 $Pre_\alpha$.*
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 $\alpha$*
6. *If last action executed was a* send[1] *event, then $\alpha$ 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.*

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

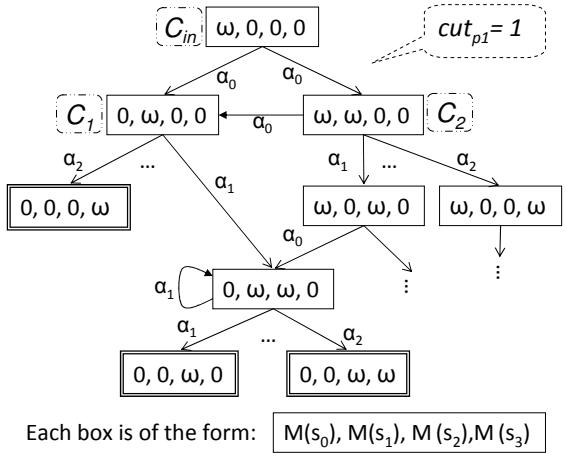


**Figure 5.1:** Partial (abstract) state exploration graph for $p_1$ ($N_{p_1} = \omega$).

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

For illustration, we again consider the example in Figure 3.2. Assume that, an unbounded number of processes ($\omega$) 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 $\epsilon$. 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 5.1. Initially, action $\alpha_0$ 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 $\omega$ represents exactly one process in state $s_0$ at $C_a^{in}$, while configuration $C_a^2$ (towards right) corresponds to the case where $\omega$ represents two or more processes in state $s_0$ at $C_a^{in}$. Further, all actions $\alpha_0$, $\alpha_1$ and $\alpha_2$ are enabled at configuration $C_a^2$, while only $\alpha_1$ and $\alpha_2$ are

enabled at configuration $C_a^1$. The paths following these abstract configurations are explored in a similar manner.

## 5.2 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 5.3.** *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 $\sigma$ 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, $\sigma$ 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

$\alpha$ in concrete execution, a process $o$ of type $p$ moves from state $s_\alpha$ to state $s'_\alpha$. Hence, $update_c(s_\alpha, M_c, \alpha, s'_\alpha, M'_c)$ from Def. 4.3 holds.

Since $C_c \simeq C_a$, by Def. 5.3, $v_c^g = v_a^g$; moreover, $M_a(s_\alpha) \geq |M_c(s_\alpha)|$ and $M_a(s'_\alpha) \geq |M_c(s'_\alpha)|$. Thus, action $\alpha$ can be executed by choosing a process from state $s_\alpha$ 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 4 and 5), 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 $\alpha$.

1. If $\alpha$ 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 $\alpha$, 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 $\alpha$ is the same on $v_c^g$ and $v_a^g$, we have $v_c^{g'} = v_a^{g'}$. 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. $\square$

## 5.3 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 $\omega$) 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 ($N_p^a$) is equal to the initial number of processes in the concrete execution ($N_p^c$), i.e. $N_p^a = N_p^c$.

**Definition 5.4.** *Let $CFG_{abs}$ (CFG) be the set of abstract (concrete) configurations of system model $\mathcal{S}$. Then for all $C_a = (v_a^g, M_a) \in CFG_{abs}$ and $C_c = (v_c^g, M_c) \in CFG$, $C_a \simeq_a C_c$ iff $v_a^g = v_c^g$ 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 $\sigma$ can be exhibited in the concrete execution of $\mathcal{S}$ satisfying* **C2**, *and for $i \geq 0$, $C_a^i \simeq_a C_c^i$, where $C_a^i$ ($C_c^i$) is the abstract (concrete) configuration before the abstract (concrete) execution of $\alpha_i$.*

*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:**

$\forall(C_a, \alpha, C'_a) \in \hookrightarrow_a, \forall C_c \in CFG,\ C_a \simeq_a C_c \Rightarrow \exists(C_c, \alpha, C'_c) \in \hookrightarrow$, s.t. $C'_a \simeq_a C'_c$.

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

Since $C_a \simeq_a C_c$, we have $v_a^g = v_c^g$ 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 $\alpha$ can be executed by a process $o_p$ of type $p$ at state $s_\alpha$ 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 $\alpha$, we consider the following cases:

1. If $\alpha$ 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 $\alpha$, 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 $\alpha$ 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. □

## 5.4 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.

### 5.4.1 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 $Var^\omega$. For a process-count variable, we only allow assignment operation that initializes it with a constant value

or $\omega$, as well as operations that increment or decrement its value by 1, and obey the following execution semantics[2]. 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 $v \in [0, cut_p)$, $B$ 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 $\omega$ processes in abstract execution, the value of $v$ lies in the domain $[0, cut_p) \cup \{\omega\}$, where $\omega$ 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 $\omega$, then the resulting value of $v$ is non-deterministically chosen to be either $cut_p - 1$ or $\omega$. 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

[2]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.

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 $\omega$, to distinguish it from the concrete domain of $x$, i.e. $\mathcal{D}_x$ $(= \mathbb{N})$, we represent the abstract domain as $\mathcal{D}_x^a$ (*i.e.* $\mathbb{N} \cup \{\omega\}$). Note that, for all other variables $y \in (Var_{\mathcal{P}} \cup Var_G) \backslash Var^{\omega}$, $\mathcal{D}_y^a = \mathcal{D}_y$. Then, we use $Val_p^a$ to represent the abstract valuations of variables in $Var_p$, which is a mapping for each variable to its abstract domain. Let $Val_{\mathcal{P}}^a = \bigcup_{p \in \mathcal{P}} Val_p^a$. The abstract valuations of global variables is represented similarly as $Val_G^a$. Accordingly, the abstract states of a process type $p$ are represented as $S_p^a \subseteq L_p \times Val_p^a$, 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 $s_{p,in}^a = (l_{in}^p, v_{p,in}^a)$, where $v_{p,in}^a$ is an initial valuation of variables in $p$ over abstract variable domain. Further, we set $S_{\mathcal{P}}^a = \bigcup_{p \in \mathcal{P}} S_p^a$.

Since the domain of variables in $Var^{\omega}$ 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: $\mathcal{R} \subseteq (Val_G \times Val_{\mathcal{P}}) \times (Val_G^a \times Val_{\mathcal{P}}^a)$. For all $g \in Val_G, f \in Val_{\mathcal{P}}, g_a \in Val_G^a, f_a \in Val_{\mathcal{P}}^a : (g, f)\mathcal{R}(g_a, f_a)$ iff

1. $\forall v \in Var_G \backslash Var^{\omega}, g_a(v) = g(v)$.

2. $\forall v \in Var_{\mathcal{P}} \backslash Var^{\omega}, f_a(v) = f(v)$.

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.

### 5.4.2 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 5.5.** *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* $\forall p \in \mathcal{P}$, $\forall s_c = (l, v_c) \in S_p, \exists s_a = (l, v_a) \in S_p^a$, $s.t. (v_c^g, v_c) \mathcal{R} (v_a^g, v_a) \wedge M_a(s_a) \geq |M_c(s_c)|$.

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

**Theorem 3.** *Let* $\sigma$ *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,* $\sigma$ *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 $\alpha$, 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. 4.3 holds.

Since $C_c \simeq C_a$, by Def. 5.5, $\exists s_a = (l, v_a) \in S_{\mathcal{P}}^a$, $s.t. (g, f) \mathcal{R} (g_a, f_a) \wedge M_a(s_a) \geq |M_c(s_c)|$, where $g \in Val_G, f \in Val_{\mathcal{P}}, f_a \in Val_G^a, f_a \in Val_{\mathcal{P}}^a$. Since $\alpha$ is executable from $s_c$, to show that $\alpha$ is also executable from $s_a$ in the abstract execution, we first need to show that $v_a^g$ and $v_a$ satisfy $Pre_\alpha$. 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 $Pre_\alpha$. 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 $B \equiv x\mathbf{Relop}c$, 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 $\omega$ 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 $Pre_\alpha$, $g_a(x)$ satisfies $Pre_\alpha$. Similar argument applies to $x \in Var_\mathcal{P} \backslash Var^\omega$.

Let $C'_a = (v_a^{g'}, M'_a) \in CFG_{abs}$ be the resulting abstract configuration, such that a process moves to state $s'_a = (l', v'_a) \in S_p^a$ by executing $\alpha$ 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 $\alpha$ 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 $\alpha$ 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 5, 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. □

Note that under condition **C1** and **C2** (refer Page 28), $S_p^a = S_p$ 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 5 also apply in this case.

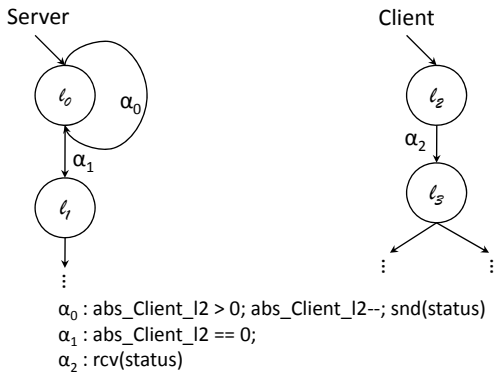


**Figure 5.2:** Example of using process-count variable

### 5.4.3 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 5.2 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 5.3). 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

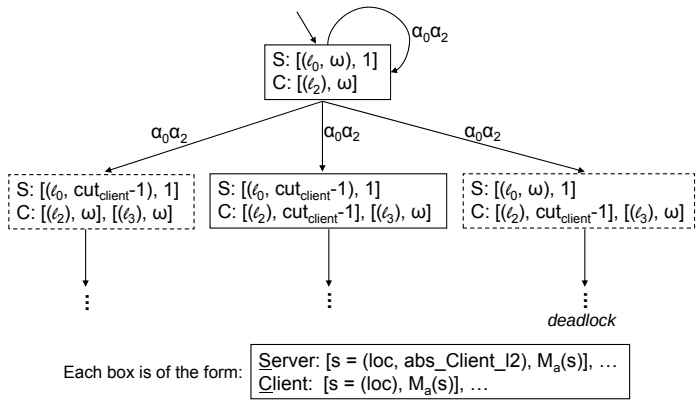


**Figure 5.3:** Partial state exploration in the presence of process-count variable

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 of $abs\_p\_l$ does not equal to $M_a(s)$, and hence should be reported as an error); moreover, after the execution of $\alpha$, $M_a(s)$ 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 5.2. 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 $\alpha_0$, internal variable $int\_Client\_l2$ is updated; and when $\alpha_2$ 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 5.3) 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 $p$.

# CHAPTER 6

# VERIFICATION

We now elaborate our verification procedure outlined earlier in Figure 3.1. It proceeds on the abstract state representation discussed in the preceding.

## 6.1 Model Checking

We use *linear-time temporal logic* (LTL) [26] for specifying the properties to be verified. This decision is influenced by our use of model checker SPIN [19] 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 5.4, our system model may also contain process-count variables (denoted as $Var^{\omega}$), 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 $v$ **Relop** $c$, such that $c \in [0, cut_p)$ and **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 $\varphi$ 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, $\varphi$ 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 elimination introduced in section 5.4.3.

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 $\alpha_i$ 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 $\sigma$. We use $new(s,\sigma)$ ($del(s,\sigma)$) to represent the number of processes that are created (deleted) during execution of $\sigma$ 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:

$$init(s) + enter(s,\sigma) - leave(s,\sigma) > 0 \tag{6.1}$$

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

$$n_{p,\sigma} = min(N_p, leave(s^p_{in},\sigma)) \tag{6.2}$$

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 $\sigma$ (excluding $\sigma$). We now consider two cases, based on whether a counter-example is finite or infinite.

**Case-A: $\sigma$ is finite.** Let $\sigma = \alpha_0 \dots \alpha_n$ be a finite counter-example trace obtained from an abstract verification run s.t., action $\alpha_i$ 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 $\sigma$ *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 $\sigma$ to be non-spurious, by showing that $\sigma$ 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 $\sigma$.

Base case: It holds trivially for $|\sigma| = 0$.

Induction hypothesis: Trace $\sigma_1 = \alpha_0 \dots \alpha_{k-1}$ can be exhibited in a concrete execution with initially $n_{p,\sigma}$ number of $p$-processes. Further, for all $0 \leq i < k$, action $\alpha_i$ has the same source state $src(\alpha_i)$ and destination state $dst(\alpha_i)$ in both concrete and abstract execution.

Inductive step: We now consider execution of $\sigma_1 \cdot \alpha_k$. Since, $\sigma_1$ is a prefix of $\sigma$ 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 $\alpha_k$ 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. (6.1)). 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 $\sigma_1$. Since $\sigma_1$ 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 $\alpha_k$ 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 $\sigma_1$) which can be chosen to execute $\alpha_k$.

**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 $\alpha_k$ in concrete execution. Otherwise, $n_{p,\sigma} = leave(s_k, \sigma)$. Since in abstract execution $\alpha_k$ is executed by a process in state $s_k$ (after occurrence of $\sigma_1$), 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 $\alpha_k$.

**A.2 $\Rightarrow$:** We show this by contradiction. Assume that $\sigma = \alpha_0 \ldots \alpha_n$ is non-spurious. Then $\sigma$ can be exhibited in a concrete execution with $N^c_p \leq N_p$ ($N^c_p \in \mathbb{N}$) 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 $\gamma$, 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 $\gamma$ that can be chosen to execute $\alpha_k$, which is a contradiction. □

**Case-B: $\sigma$ is infinite.** In this case, $\sigma$ is of the form $\sigma_{pr}(\sigma_{sx})^\omega$. Here $\sigma_{pr}$ and $\sigma_{sx}$ are finite action sequences s.t., $(\sigma_{sx})^\omega$ represents an unbounded repetition of $\sigma_{sx}$, and the abstract configurations before and after each iteration of $\sigma_{sx}$ 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 $\sigma_{sx}$. Then, we show that: *$\sigma$ 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 $\sigma'$ can be exhibited in a concrete system. In fact, by reusing the arguments from Case-A.1 described earlier, we get that $\sigma'$ 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 $\sigma_{sx}$ are same. Hence, $\sigma_{sx}$ 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 $\sigma$ 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 $\sigma_{sx}$, the number of processes in state $s'$ will be strictly less than what it was before the occurrence of $\sigma_{sx}$. Hence, after a finite number of iterations of $\sigma_{sx}$ in concrete execution, the number of processes in $s'$ will become 0. Therefore, $\sigma_{sx}$ cannot iterate infinitely often.

Case (b) above implies that after each iteration of suffix $\sigma_{sx}$, the number of processes in state $s'$ will be strictly greater than what it was before the occurrence of $\sigma_{sx}$. Hence, the number of processes in $s'$ will grow unboundedly as $\sigma_{sx}$ 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, $\sigma$ cannot occur

in a concrete execution (i.e. it is spurious), which contradicts our assumption. □

## 6.3 Abstraction Refinement

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

**Finite counter-example.** Let $\sigma = \alpha_0 \dots \alpha_n$ be a finite spurious counter-example such that, action $\alpha_i$ 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 $\sigma$, excluding $\sigma$ itself. Since, $\sigma$ 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 $\sigma_m = \alpha_0 \dots \alpha_{k-1}$ 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 $\sigma_m$ in abstract execution there can be no processes in state $s_k$. However, $\alpha_k$ is executed by a process from $s_k$ after the occurrence of $\sigma_m$ in abstract execution – this is only possible, if process count in $s_k$ becomes unbounded (i.e. $\omega$) during execution of $\sigma_m$. Assuming $s_k$ is a state of process type $p$, its process count can become $\omega$ only if number of processes in $s_k$ becomes $cut_p$ (the cutoff number of $p$) during execution of $\sigma_m$. In order to prevent the process count in $s_k$ from becoming $\omega$ in abstract execution, we determine the maximum number of processes that can reside in $s_k$ during execution of $\sigma_m$, 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 $\sigma$ in subsequent abstract verification runs.

For illustration, consider the system model with a single process type $p_1$ as shown in Figure 3.2 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 3, with $cut_{p1} = 1$, $\sigma$ 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 $\sigma$ 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 $\sigma_m$, we have $init(s_2) = 0$ and $enter(s_2, \sigma_m) = leave(s_2, \sigma_m) = 1$. Considering all prefixes of $\sigma_m$, we can easily find that $max(s_2, \sigma_m) = 1$. Then, if $cut_{p_1}$ is set to 2, the execution sequence $\sigma$ 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, $\sigma$ 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 $\sigma_{sx}$ s.t. $enter(s, \sigma_{sx}) \neq leave(s, \sigma_{sx})$ in abstract execution.

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

repeated execution of $\sigma_{sx}$. We consider two sub-cases.

**(ii-a)** The process count in $s$ is $\omega$ at the beginning/end of every execution of $\sigma_{sx}$ (in the abstract execution of $\sigma$).

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

For case (ii-a), the process count in $s$ is $\omega$ at the beginning of the first execution of $\sigma_{sx}$ while executing $\sigma = \sigma_{pr}(\sigma_{sx})^{\omega}$, that is, at the end of $\sigma_{pr}$ 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 $\sigma_{pr}$.

For case (ii-b), our abstraction refinement similarly prevents the process count from becoming $\omega$ in $s$ during the execution of $\sigma_{sx}$ in $\sigma$. 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 $\sigma$. Thus, trace $\sigma$ 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)** $\sigma$ is *finite*. Then from Section 6.2, Case-A.1, we know that $\sigma$ can be exhibited in a finite state system with initially $n_{p,\sigma} \in \mathbb{N}$ number of processes of type $p$. From

Eqn. (6.2) page 41, 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 $\sigma$.

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

# CHAPTER 7

# IMPLEMENTATION

SPIN [19, 18] 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 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

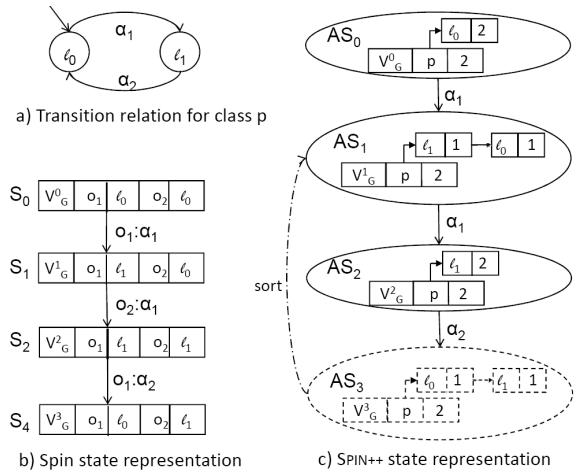


**Figure 7.1:** State representation in SPIN and SPIN++

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 7.1(a), where type $p$ has two actions $\alpha_1$ and $\alpha_2$, 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 7.1(b), with $V_G^i$ ($i \in [0, 4]$) representing the valuations of global variables. We assume that $\alpha_1, \alpha_2$ 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[1] 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 7.1(a). The abstract system states visited during the execution of $\sigma$ are shown in Figure 7.1(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.

## 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 [19], to avoid any run-time overheads, these

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

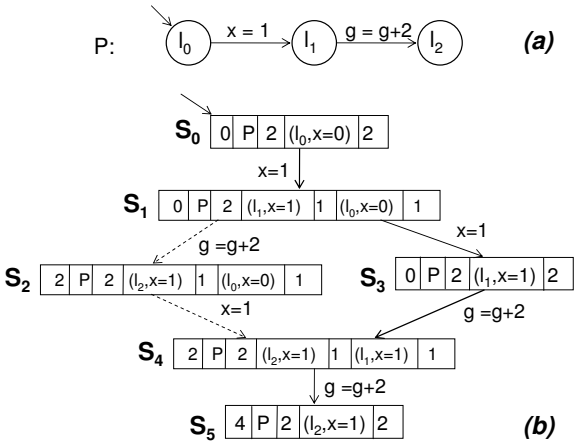


**Figure 7.2:** Partial Order Reduction in SPIN++

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 7.2(a). The state exploration graph for this example in SPIN++ with two instantiations of process type $P$ is shown in Figure 7.2(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 7.2(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 7.2(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 7.1(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 7.1(c), the "components" would be – *global information*, and partitions $s_0 = (l_0, \epsilon)$, $s_1 = (l_0, \epsilon)$ of type $P$ ($\epsilon$ 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.

# CHAPTER 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.

## 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 [19, 18] 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 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 $Var^{\omega}$), 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 $v_p$ **Relop** $c$, such that $c \in [0, cut_p)$ and **Relop** is any relational operator. This restriction ensures deterministic evaluation of boolean expressions involving the process-count variables.

**Table 8.1:** 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 |

## 8.2 Examples Modeled

For experiments, we modeled the following four examples. In Table 8.1, 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 [1]. 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 [2], 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 [3] 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 [20], 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 the number of processes.

The experimental results for state space exploration are shown graphically in Figure 8.1. As we can observe, SPIN++ clearly outperforms SPIN by a significant margin as the number of processes in the system increases. Moreover, with the

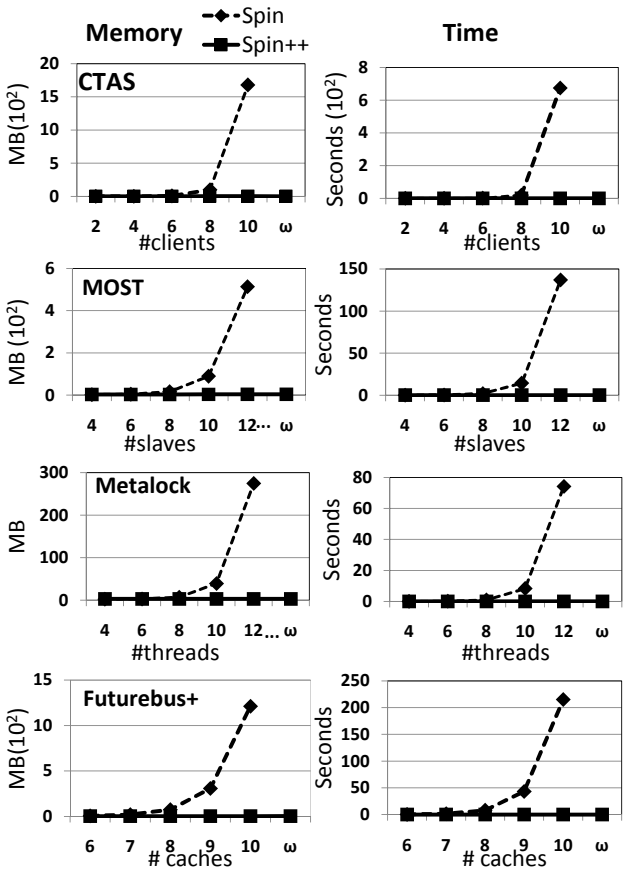


**Figure 8.1:** State space exploration results.

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 $\omega$ number of processes using SPIN++ are also shown (the last entry in these graphs).

In Figure 8.2, 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

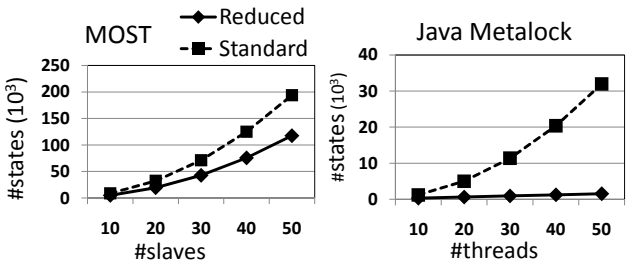


**Figure 8.2:** Partial order reduction in SPIN++.

**Table 8.2:** Collapse compression in SPIN++.

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

**O.M.** indicates Out of Memory.

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 8.2). 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.

**Table 8.3:** LTL property verification in Spin++.

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

## 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 8.3.

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. 8.3).

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. 8.3).

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. 8.3).

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. 8.3).

As we can observe from Table 8.3, 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 $\omega$, thus avoiding any spurious behaviors during abstract verification.

For experiments with an unbounded ($\omega$) 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.

# CHAPTER 9

# DISCUSSION AND CONCLUSION

In this thesis, we have presented our research on a usable and efficient verification framework for concurrent parameterized systems. Our verification method is based on the principle of abstraction refinement. Our abstraction only keeps the number of processes for each process type, but abstracts away the individual process identities. The abstraction was shown to be sound and exact. If a counter-example is found in model checking and is shown to be spurious, then the abstraction is refined by increasing the cutoff number for the process type consisting of unbounded number of processes. Moreover, if a counter-example is non-spurious, then heuristics is used to construct a small, finite system that is sufficient to exhibit the same counter-example trace.

We modified SPIN to integrate our verification techniques. The modified SPIN (called SPIN++) has been successfully used to verify several real-life software systems.

In terms of future works, there are many directions that we can pursue. First of all, our current verification framework can be extended to handle unbounded data domain. Many parameterized verification techniques are suitable for analyzing protocols consisting of large, finite-state systems. However, there are systems that cannot be modeled as finite-state systems, mainly due to unbounded data domains. Our use of process-count variables (in section 5.4) is a way of handling variables with

unbounded data domain, provided that the variable refers to process counts of the other process types. However, for protocols with global or local variables that grow unbounded regardless of the number of processes in the system, we may consider integrating predicate abstraction into our current verification framework. The idea is that, instead of keeping track of the number of processes at each local state, we keep track of the number of processes at each local state that satisfy or not-satisfy the given predicates.

Secondly, many distributed protocols, as well as real-life software systems, consist of processes that are not only distinguished by their identities, but also their connections/communication with other processes within the system (eg. leader election in a ring structure, where each process needs to have neighboring information). Hence, we may consider applying our count abstraction to the number of communication/association between processes.

Additionally, automatic counter-example explanation techniques may be integrated into our verification for easier debugging.

# REFERENCES


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

[2] "Media oriented system transport (MOST)," `http://www.mostcooperation.com/`.

[3] Agesen, O., Detlefs, D., Garthwaite, A., Knippel, R., Ramakrishna, Y., and White, D., "An efficient meta-lock for implementing ubiquitous synchronization," in *OOPSLA*, 1999.

[4] Apt, K. and Kozen, D., "Limits for automatic verification of finite-state systems," *Information Processing Letters*, vol. 15, 1986.

[5] Ball, T. and Rajamani, S., "The SLAM Project: Debugging System Software via Static Analysis," in *POPL*, 2002.

[6] Basu, S. and Ramakrishnan, C., "Compositional analysis for verification of parameterized systems," in *TACAS*, 2003.

[7] Betin-can, A. and Bultan, T., "Verifiable concurrent programming using concurrency controllers," in *In Proc. of the 19th IEEE Int. Conf. on Automated Software Eng*, pp. 248–257, 2004.

[8] Beyer, D., Henzinger, T., Jhala, R., and Majumdar, R., "The Software Model Checker BLAST: Applications to Software Engineering," *Int. Journal on Software Tools for Technology Transfer*, 2007.

[9] Chaki, S., Clarke, E., Groce, A., Jha, S., and Veith, H., "Modular verification of software components in c," in *ICSE*, 2003.

[10] Clarke, E., Talupur, M., and Veith, H., "Environment abstraction for parameterized verification," in *In 7 th VMCAI, LNCS 3855*, pp. 126–141, Springer, 2006.

[11] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H., "Counterexample-guided abstraction refinement for symbolic model checking," *JACM*, vol. 50, no. 5, 2003.

[12] Delzanno, G., "Automatic verification of parameterized cache coherence protocols," in *CAV*, 2000.

[13] Emerson, E. and Kahlon, V., "Reducing model checking of the many to the few," in *CADE*, 2000.

[14] Emerson, E. and Kahlon, V., "Parameterized model checking of ring-based message passing systems," in *CSL*, 2004.

[15] Emerson, E. and Namjoshi, K., "On Reasoning about rings," *Intl. Jnl. on Foundations of Computer Science (IJFCS)*, vol. 14, no. 4, 2003.

[16] Fang, Y., McMillan, K., Pnueli, A., and Zuck, L. D., "Liveness by Invisible Invariants," in *FORTE*, 2006.

[17] German, S. and Sistla, A., "Reasoning about systems with many processes," *Journal of the ACM*, vol. 39, 1992.

[18] Holzmann, G. J., "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279–295, May 1997.

[19] Holzmann, G., *The SPIN model checker: Primer and Reference Manual*. Addison Wesley, 2003.

[20] IEEE Computer Society, *IEEE Standard for Futurebus+, Logical Protocol Specification*. 1992.

[21] Ip, C. and Dill, D., "Verifying Systems with Replicated Components in Mur$\varphi$," *Formal Methods in System Design*, vol. 14, no. 3, 1999.

[22] Jonsson, B. and Saksena, M., "Systematic Acceleration in Regular Model Checking," in *CAV*, 2007.

[23] Kesten, Y., Maler, O., Marcus, M., Pnueli, A., and Shahar, E., "Symbolic model checking with rich assertional languages," in *CAV*, 1997.

[24] Lahiri, S. K. and Bryant, A. E., "Indexed predicate discovery for unbounded system verification," in *In CAV04*, pp. 135–147, Springer, 2004.

[25] Lesens, D., Halbwachs, N., and Raymond, P., "Automatic Verification of Parameterized Linear Networks of Processes," in *POPL*, 1997.

[26] Manna, Z. and Pnueli, A., *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1991.

[27] Pnueli, A., Xu, J., and Zuck, L., "Liveness with (0,1,infinity)-counter abstraction," in *CAV*, 2002.

[28] Pong, F. and Dubois, M., "A new approach for the verification of cache coherence protocols," *IEEE Transacations on Parallel and Distributed Systems*, vol. 6, no. 8, 1995.

[29] Roychoudhury, A. and Ramakrishnan, I., "Automated inductive verification of parameterized protocols," in *CAV*, 2001.

[30] Yavuz-kahveci, T. and Bultan, T., "Specification, verification, and synthesis of concurrency control components," in *In Proceedings of the 2002 ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pp. 169–179, ACM Press, 2002.