Parameterized Verification of π-Calculus Systems*

Ping Yang[1], Samik Basu[2], and C.R. Ramakrishnan[1]

[1] Dept. of Computer Science, Stony Brook Univ., Stony Brook, NY, 11794, USA
[2] Dept. of Computer Science, Iowa State Univ., Ames, IA, 50014, USA
{pyang, cram}@cs.sunysb.edu, sbasu@cs.iastate.edu

Abstract. In this paper we present an automatic verification technique for parameterized systems where the subsystem behavior is modeled using the π-calculus. At its core, our technique treats each process instance in a system as a property transformer. Given a property φ that we want to verify of an N-process system, we use a partial model checker to infer the property φ' (stated as a formula in a sufficiently rich logic) that must hold of an $(N-1)$-process system. If the sequence of formulas $\varphi, \varphi', \ldots$ thus constructed converges, and the limit is satisfied by the deadlocked process, we can conclude that the N-process system satisfies φ. To this end, we develop a partial model checker for the π-calculus that uses an expressive value-passing logic as the property language. We also develop a number of optimizations to make the model checker efficient enough for routine use, and a light-weight widening operator to accelerate convergence. We demonstrate the effectiveness of our technique by using it to verify properties of a wide variety of parameterized systems that are beyond the reach of existing techniques.

1 Introduction

A parameterized system consists of a number of instances of a component, the number of such occurrences being the parameter to the system. Many safety-critical systems are naturally parameterized: e.g. resource arbitration protocols, communication protocols, etc. Traditional model checking techniques are limited to verifying properties of a given instance of a parameterized system (i.e. for a specific value of the parameter). Many novel techniques have been developed to verify such systems for all instances of their parameters [12, 15, 16, 10]. These techniques vary in the classes of systems they can handle and the degree of automation they provide. Automatic techniques typically restrict the communication topology (e.g. rings or trees) or, at least, demand that the communication patterns be fixed.

The Driving Problem. In many systems, e.g. mobile systems, the process interconnections can change dynamically. Existing techniques for verifying parameterized systems do not readily extend to such systems. In this paper, we present an automatic technique to address this problem.

The π-calculus [28] is a well-known process calculus where communication channels as well as values transmitted over them belong to the same domain of *names*; names can be dynamically created, communicated to other processes, and can be used as channels. Due to these features, it is widely used as the basis for modeling mobile

* This research was supported in part by NSF grants CCR-0205376, CCR-0311512, and CCR 0509340.

© Springer-Verlag Berlin Heidelberg 2006

$$p(x) \overset{\text{def}}{=} (\nu y)\overline{x}y.p(x)$$
$$q(x) \overset{\text{def}}{=} x(y).q(x)$$
$$sys(\mathbf{n}) \overset{\text{def}}{=} (\nu x)(p(x) \mid q^{\mathbf{n}}(x))$$

(a)

$$\varphi_0 \equiv X =_\nu \langle\tau\rangle tt \wedge [\tau]X$$
$$\varphi_1 \equiv X_1(x) =_\nu \nu y'((\langle xy'\rangle tt \vee \langle\tau\rangle tt) \wedge [xy']X_1(x) \wedge [\tau]X_1(x))$$
$$\varphi_2 \equiv X_2(x) =_\nu \nu y'([xy']X_2(x) \wedge [\overline{x}\{y\}]X_2(x) \wedge [\tau]X_2(x))$$
$$\varphi_3 \equiv X_3(x) =_\nu \nu y'([xy']X_3(x) \wedge [\overline{x}\{y\}]X_3(x) \wedge [\tau]X_3(x))$$

(b)

Fig. 1. A simple example of a parameterized system

systems. In a parameterized mobile system, we assume that each component is specified as a finite-control π-calculus process: i.e. specified without using the replication operator of the calculus, and not containing a parallel composition within the scope of a recursive definition. A simple example of a parameterized system based on the π-calculus is shown in Fig. 1(a). In the figure, the parameterized system is represented by process $sys(n)$, which consists of one instance of process $p(x)$ and n instances $q(x)$. The process $p(x)$ creates a new name y and outputs it via channel x, while the process $q(x)$ receives a name via x. The property to be verified, φ_0, is specified in the modal μ-calculus [24, 8] and written in equational form (Fig. 1(b)). The property is a greatest fixed point formula (specified by a $=_\nu$ equation) and states that *a τ action is possible after every τ action*. An example of parameterized verification problem is to determine whether $\forall n.\ sys(n) \models \varphi_0$.

Background. In [6], we developed a compositional model checker for the process algebra CCS [27] and for properties specified in the model μ-calculus [8]. We used the compositional checker for the verification of parameterized CCS processes. The central idea of our approach is to view processes as property transformers: given a μ-calculus formula φ and a system containing a CCS process P, we compute the property φ' that should hold in P's environment (say, Q) if φ holds in $P|Q$. The property transformer of a process P, denoted by $\Pi(P)$, is such that: $\forall Q.\ (P|Q \models \varphi) \Leftrightarrow (Q \models \Pi(P)(\varphi))$.

Consider a parameterized system P^n consisting of n instances of a process P. To verify whether φ holds in P^n for all n, we construct the sequence of properties $\varphi_0, \varphi_1, \ldots$ such that $\varphi_0 = \varphi$ and $\varphi_{i+1} = \Pi(P)(\varphi_i)$ for all $i \geq 0$. Let the sequence converge after k steps: i.e. $\varphi_{k+1} = \varphi_k$. By definition of Π, note that for $n \geq k$, $P^n \models \varphi$ if $P^{n-k} \models \varphi_k$. Let $\mathsf{0}$ denote the deadlocked process, the unit of the parallel composition operator. Specifically, P^n is equivalent to $P^n|\mathsf{0}$. It then follows that $\forall n \geq k, P^n \models \varphi$ if $\mathsf{0} \models \varphi_k$, i.e. the zero process has the property specified by limit of the sequence of formulas.

Our Solution. Following the approach of [6], we develop a compositional model checker for the π-calculus and use that as the basis for verifying parameterized mobile systems. Consider the example in Figure 1. In order to show that $sys(n) \models \varphi_0$ for arbitrary n, we begin by determining a property $\varphi_1 = \Pi(p(x))(\varphi_0)$. By the definition of Π, we know $q^n(x) \models \varphi_1$ whenever $sys(n) \models \varphi_0$.

In order to specify φ_1 correctly, the property language needs to be expressive enough to specify names and their scopes. We extend the modal μ-calculus to a logic called the $\mathsf{C}\mu$-calculus. In this logic, formula variables may be parameterized by names. Moreover, formulas may specify local names (denoted by νx) and may contain modalities with new actions such as the *free input* action xy (see Section 2).

In the above example, observe that $p(x)|Q$ (for any process Q) can do a τ-action if (a) Q can do an input action on x to synchronize with $p(x)$'s bound output action $\overline{x}\nu y$, or (b) Q itself can do a τ-action. Thus the term $\langle\tau\rangle\varphi'$ holds in $p(x)|Q$ if $(\langle xy\rangle\varphi'' \vee \langle\tau\rangle\varphi'')$ holds in Q. The other modalities and operations in the formula are derived along the same lines using the property transformer for $p(x)$. The resulting property φ_1, defined in $\mathtt{C}\mu$-calculus using the formula variable X_1, is shown in Figure 1(b). It states that it is always possible to input from x or perform a τ action after any such action. Observe that free name x is the parameter to the formula variable X_1. We now check if φ_1 holds in $q^n(x)$, by checking if $\varphi_2 = \Pi(q(x))(\varphi_1)$ holds in $q^{n-1}(x)$. Observe that φ_2 does not have the conjunct $\langle xy'\rangle tt \vee \langle\tau\rangle tt$ since a single instance of $q(x)$ can satisfy it. Using the terminology of assume-guarantee proof techniques [19], we can say that the obligation of $\langle xy'\rangle tt \vee \langle\tau\rangle tt$ on $q^n(x)$ is satisfied by one instance of $q(x)$ and hence is not passed on to $q^{n-1}(x)$. Continuing further, we can check if φ_2 holds in $q^{n-1}(x)$ by checking if $\varphi_3 = \Pi(q(x))(\varphi_2)$ holds in $q^{n-2}(x)$.

Observe from the figure that φ_3 and φ_2 differ only in the names of formula variables and hence represent the same property. We thus conclude that the sequence φ_i converges to φ_2. Moreover, since $\mathtt{0}$ satisfies φ_2 we can conclude that the original formula φ_0 is satisfied by *sys*(n) for sufficiently large n. It should also be noted that since φ_2 is a greatest fixed point formula and involves a conjunction of universal modalities, it is equivalent to tt; hence the last iteration (to compute φ_3) is redundant. Techniques to simplify formulas and to find equivalences will in general enable us to detect convergence earlier. A more careful analysis of the sequence of formulas reveals that it converges after *one* instance of $q(x)$ is considered, and hence we can conclude that $\forall n \geq 1\ sys(n) \models \varphi_0$.

Contributions. The main contributions of this paper are as follows.

- *A compositional model checker for the π-calculus.* The model checker works for finite-control π-calculus processes, as well as value-passing calculus with equality $(=)$ and dis-equality $(\neq)$ constraints between names (see Section 3).
- *Operations to efficiently check for convergence of formula sequences, and to accelerate convergence.* The verification technique for parameterized systems is based on computing the limit of a sequence of $\mathtt{C}\mu$-calculus formulas. We describe effective techniques to check if two $\mathtt{C}\mu$-calculus formulas are equivalent. We also describe a widening operator to extrapolate the sequence to estimate (approximately) its limit (Section 4).
- *Optimizations to compositional model checking.* We develop a number of lightweight optimization techniques to reduce the size of formulas generated in the intermediate steps of compositional model checking. We find that such optimizations are necessary and effective. Without these, parameterized system verification based on compositional model checking appears infeasible (see Section 5).

We also demonstrate the utility of our technique by applying it on a variety of parameterized π-calculus systems: ranging from simple ones that can also be expressed as parameterized CCS systems, to those that exhibit π-calculus-specific features of name creation, link passing and scope extrusion (Section 6).

Related work. A number of model checking techniques for the π-calculus have been developed. Examples include the model checking technique for polyadic π-calculus [11]; the Mobility Workbench (MWB) [33], a model checker and bisimulation checker for the π-calculus; a system [32] to translate a subset of π-calculus specifications into Promela for verification using Spin [20]; and MMC [35, 36] model checker for the π-calculus based on logic programming. All these techniques, however, apply only to finite-control π-calculus, and cannot be used for verifying parameterized systems.

Type systems for the verification of π-calculus processes [9, 21] handle the replication operator and appear to be a promising alternative to the verification of parameterized mobile systems. The PIPER system [9] generates CCS processes as "types" for π-calculus processes (based on user-supplied type signatures), and formulates the verification problem in terms of these process types. In [21], a generic type system for the π-calculus is proposed as a framework for analyzing properties such as deadlock- and race-freedom. The replication operator alone is insufficient to model many parameterized systems where the repeated instances may have different free variables.

The area of compositional verification has received considerable attention. Most techniques for compositional verification are based on assume-guarantee reasoning [18, 1, 26, 7, 19], and need user guidance. An approach to learn assumptions using automata learning techniques is proposed in [2]; but the technique is limited to the verification of systems with a fixed number of finite-state components. The technique presented in this paper is broadly based on our earlier technique [6] which is restricted to parameterized CCS systems and does not support dynamic change of communication topology. Other closely-related works include the compositional model checker for synchronous CCS [4] and the partial model checker of [3]. The latter defines property transformers for parallel composition of sequential automata, while we generalize the transformers for arbitrary π-calculus processes. These papers also proposed techniques to reduce the size of formulas, but the optimizations are done after the formulas are generated in the first place; in contrast, we apply our optimizations during the model checking process, thereby reducing the size of formulas generated.

Verification of parameterized systems has been recognized as an important problem and significant progress has been made in the recent years [37, e.g.]. One popular approach to the verification of a parameterized system of the form P^n is to identify a finite cut off k for a property φ such that $\forall n.P^n \models \varphi \Leftrightarrow P^k \models \varphi$, thereby reducing it to a finite-state verification problem. Techniques following this approach range from those that provide cutoffs for particular communication topologies [13, 14, e.g.], to those based on symmetries and annotations in the system specification [22]. Later works, such as [30, 5] have proposed automatic techniques, based on identification of appropriate cut-off of the parameters, for verification of wide range of parameterized systems using rich class of data objects and operations (inequalities, incrementations). Another approach is to identify an appropriate representation technique for a given parameterized system; e.g. counting abstraction with arithmetic constraints [12], covering graphs [15, 16], and context-free grammars [10], and regular languages [31]. The use of abstractions to generate invariants of parameterized systems is explored in [23]. None of these techniques, however, consider dynamically changing communication topologies.

2 A Logic for Compositional Analysis of π-Calculus Processes

In this section, we present the fundamentals of π-calculus (Section 2.1) and property specification logic, which we will refer to as Cμ-calculus (Section 2.2), followed by our technique of compositional analysis (Section 3).

2.1 Syntax and Semantics of the π-Calculus

Process algebra π-calculus [28] is used to represent behavior of systems whose interconnection pattern changes dynamically. Let $x, y, z, \ldots$ range over names, $p, q, r, \ldots$ range over process identifiers, and $\overrightarrow{x}$ represent comma-separated list of names $x_1, \ldots, x_n$. In the following, we recall the syntax of the calculus.

$$\begin{aligned}
\alpha &::= x(y) \mid \overline{x}y \mid \tau \\
\mathcal{P} &::= 0 \mid \alpha.\mathcal{P} \mid (\nu x)\mathcal{P} \mid \mathcal{P} \mid \mathcal{P} \mid \mathcal{P} + \mathcal{P} \mid [x = y]\mathcal{P} \mid p(\overrightarrow{y}) \\
\mathcal{D}_p &::= p(\overrightarrow{x}) \overset{\text{def}}{=} \mathcal{P} \text{ (where } i \neq j \Rightarrow x_i \neq x_j \text{ and } fn(\mathcal{P}) \subseteq \{\overrightarrow{x}\})
\end{aligned}$$

In the above, α denotes the set of actions where $x(y)$, $\overline{x}y$ and τ represent input, (free) output and internal actions. Input action $x(y)$ has binding occurrence of variable y. All other variables in every action are *free*. The set of process expressions is represented by $\mathcal{P}$. Process 0 represents a deadlocked process. Process $\alpha.P$ can perform an α action and subsequently behave as P. Process $(\nu x)P$ behaves as P with the scope of x initially restricted to P; x is called a local name. Process $[x = y]P$ behaves as P if the names x and y are the same name, and as 0 otherwise. The operators $+$ and $|$ represent non-deterministic choice and parallel composition, respectively. The expression $p(\overrightarrow{y})$ denotes a *process invocation* where *p* is a process name (having a corresponding definition) and $\overrightarrow{y}$ is the actual parameters of the invocation. Finally, $\mathcal{D}_p$ is the set of process definitions where each definition is of the form $p(\overrightarrow{x}) \overset{\text{def}}{=} P$. A definition associates a process name p and a list of formal parameters $\overrightarrow{x}$ with process expression P.

The operational semantics of the π-calculus is given in terms of *symbolic transition systems* where each state denotes a process expression and each transition is labeled by a boolean guard and action [25]. The operational semantics is standard and is omitted.

2.2 Syntax and Semantics of the Cμ-Calculus

For the purpose of compositional analysis, we extend value-passing μ-calculus in two ways: (i) with explicit syntactic structures to specify and manipulate local names, and (ii) with actions that are closed under complementation. We will refer to this logic as Cμ-calculus. The set of formula expressions $\mathcal{F}$ in the Cμ-calculus is defined as follows:

$$\begin{aligned}
\mathcal{F} ::= {} & tt \mid f\!f \mid x = y \mid x \neq y \mid loc(x) \mid nloc(x) \mid (\nu x)\mathcal{F} \mid \mathcal{F} \vee \mathcal{F} \mid \mathcal{F} \wedge \mathcal{F} \\
& \mid \langle \mathcal{A} \rangle \mathcal{F} \mid [\mathcal{A}]\mathcal{F} \mid \langle x(y) \rangle \exists y.\mathcal{F} \mid \langle x(y) \rangle \forall y.\mathcal{F} \mid [x(y)]\forall y.\mathcal{F} \mid [x(y)]\exists y.\mathcal{F} \\
& \mid X(\overrightarrow{e}) \mid (\mu X(\overrightarrow{z}).\mathcal{F})(\overrightarrow{e}) \mid (\nu X(\overrightarrow{z}).\mathcal{F})(\overrightarrow{e}) \\
\mathcal{A} ::= {} & xy \mid \overline{x}y \mid \overline{x}\{y\} \mid \overline{x}\nu y \mid \tau
\end{aligned}$$

$$\text{1a: } [\![x = y]\!]\xi\delta l = \begin{cases} \{s\delta \mid s \in S\} & \text{if } \delta \models x = y \\ \emptyset & \text{otherwise.} \end{cases} \qquad \text{1b: } [\![x \neq y]\!]\xi\delta l = \begin{cases} \{s\delta \mid s \in S\} & \text{if } \delta \models x \neq y \\ \emptyset & \text{otherwise.} \end{cases}$$

$$\text{2a: } [\![loc(x))]\!]\xi\delta l = \begin{cases} \{s\delta \mid s \in S\} & \text{if } x \in l \\ \emptyset & \text{otherwise.} \end{cases} \qquad \text{2b: } [\![nloc(x))]\!]\xi\delta l = \begin{cases} \{s\delta \mid s \in S\} & \text{if } x \notin l \\ \emptyset & \text{otherwise.} \end{cases}$$

$$\text{3: } [\![\varphi_1 \vee \varphi_2]\!]\xi\delta l = [\![\varphi_1]\!]\xi\delta l \cup [\![\varphi_2]\!]\xi\delta l \qquad \text{4: } [\![\varphi_1 \wedge \varphi_2]\!]\xi\delta l = [\![\varphi_1]\!]\xi\delta l \cap [\![\varphi_2]\!]\xi\delta l$$

$$\text{5: } [\![(\nu x)\varphi]\!]\xi\delta l = \{s \mid s \in [\![\varphi\{x'/x\}]\!]\xi\delta(l \cup \{x'\}) \text{ where } x' \notin fn(s)\}$$

$$\text{6: } [\![\langle\tau\rangle\varphi]\!]\xi\delta l = \{s \mid \exists s'.s \xrightarrow{b,\tau} s' \wedge (\delta, l \models b) \wedge s' \in [\![\varphi]\!]\xi\delta l\}$$

$$\text{7: } [\![\langle\overline{x_1}v\rangle\varphi]\!]\xi\delta l = \{s \mid \exists s'.s \xrightarrow{b,\overline{x_2}v} s' \wedge (\delta, l \models b \wedge (x_1 = x_2)) \wedge s' \in [\![\varphi]\!]\xi\delta l\}$$

$$\text{8: } [\![\langle\overline{x_1}\{y\}\rangle\varphi]\!]\xi\delta l = \{s \mid \exists s'.s \xrightarrow{b,\overline{x_2}v} s' \wedge (\delta, l \models b \wedge (x_1 = x_2)) \wedge s' \in [\![\varphi\{v/y\}]\!]\xi\delta l\}$$

$$\text{9: } [\![\langle\overline{x_1}\nu y\rangle\varphi]\!]\xi\delta l = \{s \mid \exists s'.s \xrightarrow{b,\overline{x_2}\nu v} s' \wedge v \notin fn(\varphi) - \{y\} \wedge (\delta, l \models b \wedge (x_1 = x_2)) \\ \wedge s' \in [\![\varphi\{v/y\}]\!]\xi\delta(l \cup \{v\})\}$$

$$\text{10: } [\![\langle x_1 y\rangle\varphi]\!]\xi\delta l = \{s \mid \exists s'.s \xrightarrow{b,x_2(w)} s' \wedge (\delta, l \models b \wedge (x_1 = x_2)) \wedge s'\{y/w\} \in [\![\varphi]\!]\xi\delta l\}$$

$$\text{11: } [\![\langle x_1(y)\rangle\exists y.\varphi]\!]\xi\delta l = \{s \mid \exists s'.s \xrightarrow{b,x_2(w)} s' \wedge (\delta, l \models b \wedge (x_1 = x_2)) \wedge \exists v.s'\{v/w\} \in [\![\varphi\{v/y\}]\!]\xi\delta l\}$$

$$\text{12: } [\![\langle x_1(y)\rangle\forall y.\varphi]\!]\xi\delta l = \{s \mid \exists s'.s \xrightarrow{b,x_2(w)} s' \wedge (\delta, l \models b \wedge (x_1 = x_2)) \wedge \forall v.s'\{v/w\} \in [\![\varphi\{v/y\}]\!]\xi\delta l\}$$

$$\text{13: } [\![X(\vec{e})]\!]\xi\delta l = \xi(X)(\vec{e}\,\delta)$$

$$\text{14: } [\![(\mu X(\vec{z}).\varphi)(\vec{e})]\!]\xi\delta l = (\cap\{f \mid [\![\varphi]\!](\xi \circ \{X \mapsto f\}) \subseteq f\})\delta[\vec{e}/\vec{z}]l$$

$$\text{15: } [\![(\nu X(\vec{z}).\varphi)(\vec{e})]\!]\xi\delta l = (\cup\{f \mid f \subseteq [\![\varphi]\!](\xi \circ \{X \mapsto f\})\})\delta[\vec{e}/\vec{z}]l$$

Fig. 2. Semantics of the Cμ-calculus

In the above, *tt* and *ff* stand for propositional constants true and false, respectively. *loc*(x) is true iff x is a local name, and *nloc*(x) is true iff x is not a local name. The scope of names can be specified by formulas of the form $(\nu x)\mathcal{F}$ which means that x is a local name in the formula. Formulas can be constructed using conjunction, disjunction, diamond (existential) and box (universal) modalities and quantifiers. The modal actions $x(y)$, xy, and τ represent input, free input and internal actions, respectively. $\overline{x}y$ is a free output action where y is a free name and $\overline{x}\{y\}$ is an output action that has binding occurrence of variable y. In input and output actions $x(y)$ and $\overline{x}\{y\}$, x is free and y is bound; in free input and free output actions, all names are free. $\overline{x}\nu y$ is a bound output action; in such an action x is free and y is bound. Bound names of a formula are either bound names in the modalities or names bound by the ν operator. $\langle x(y)\rangle\exists y.\mathcal{F}$ and $\langle x(y)\rangle\forall y.\mathcal{F}$ represent basic and late diamond modalities for input action $x(y)$, respectively. $[x(y)]\forall y.\mathcal{F}$ and $[x(y)]\exists y.\mathcal{F}$ represent the basic and late box modalities for input action $x(y)$, respectively.

The least and greatest fixed point formulas are specified as $(\mu X(\vec{z}).\mathcal{F})(\vec{e})$ and $(\nu X(\vec{z}).\mathcal{F})(\vec{e})$, respectively, where $\vec{z}$ represents formal parameters and $\vec{e}$ represents actual parameters. For convenience, we often represent a formula as a sequence of fixed point equations [17]. We assume that all formulas are *closed*, i.e., all free names in a formula appear in the parameters of the definition.

Semantics of the Cμ-calculus. The semantics of formulas in the Cμ-calculus is given using four structures: (i) a symbolic transition system $\mathcal{S} = \langle S, \rightarrow\rangle$ where S represents the set of symbolic states and '$\rightarrow$' is the symbolic transition relation; (ii) a substitution δ over which the equality $(=)$ and disequality $(\neq)$ constraints between names are interpreted; (iii) a function ξ that maps formula variables to sets of symbolic states of $\mathcal{S}$; and a set of local names l used to assign meaning to *loc* and *nloc* predicates. The semantic function is written as $[\![\varphi]\!]\xi\delta l$ and maps each formula to a set of states in S. The symbolic transition system is used as an implicit parameter in the definition: all rules are evaluated w.r.t. the same transition system. The treatment of boolean connectives is straightforward. The set of local names, l, is updated in Rules 5 and 9 to include names bound by ν operator. Similarly, the substitution δ is updated to capture the mapping of formal parameters (free names) to actual arguments in Rules 14 and 15. Constraints of the form $x = y$ and $x \neq y$ are evaluated under this substitution. Rules 6–12 give the semantics for the diamond modality. The semantics of the box modality can be easily obtained by considering it as the dual of the diamond modality. For instance, the semantics for $[\tau]\varphi$ is: $[\![[\tau]\varphi]\!]\xi\delta l = \{s \mid \forall s'. \text{ if } s \xrightarrow{b,\tau} s' \wedge \delta, l \models b \text{ then } s' \in [\![\varphi]\!]\xi\delta l\}$. For brevity, we will henceforth discuss only about the diamond modality. The details related to the box modality are given in [34]. We will use $s \models_{\delta,l} \varphi$ to denote $s \in [\![\varphi]\!]\xi\delta l$.

3 Compositional Model Checker for the π-Calculus

In this section, we define the transformation function $\Pi : \mathcal{P} \to \mathcal{F} \to \mathcal{F}$ which is the core of our technique. Given a process $P \in \mathcal{P}$, a formula $\varphi \in \mathcal{F}$, a set of substitutions δ and a set of local names l, we define Π such that

$$P \mid Q \mid 0 \models_{\delta,l} \varphi \Leftrightarrow Q \mid 0 \models_{\delta,l} \Pi(P)(\varphi) \Leftrightarrow 0 \models_{\delta,l} \Pi(Q)(\Pi(P)(\varphi))$$

In words, the main objective of Π is to generate a $\mathrm{C}\mu$-calculus formula which represents the temporal obligation of the environment of the process used for transformation. This process of transforming formula iteratively by each process in the parallel composition is similar to the one proposed in [3, 6], where the transformation operation is defined for labeled transitions system or process algebra CCS and the technique of model checking is referred to as *partial model checking*.

The function Π for each formula expression is presented in Fig. 3. Here, we illustrate only those rules that are not obvious. Rules 3(a) and 3(b) leave the formula expressions $loc(x)$ and $nloc(x)$ unchanged; evaluation of these formulas is performed when all but the 0 processes are used to transform the formula iteratively. Rule 6 transforms a parameterized formula variable $X(\overrightarrow{e})$ into new formula variable $X_p(\overrightarrow{e_1})$ (the definition is in Rule A) where $\overrightarrow{e_1}$ is formed by concatenation of $\overrightarrow{e}$ and free names of P. Transformation using a process identifier is equivalent to transformation using its definition (Rule 9).

Rule 10 captures the compositionality of property transformers; the order of transformation using P_1 or P_2 does not matter. Rule 11 presents the property transformer for process $(\nu x)P$ where (νx) is moved from the process side to the transformed formula. In order to avoid name clash, x is renamed to x' ($\{x'/x\}$)that is different from any free names in φ. Note that, x' is a local name in the context of the transformed formula.

Rule 12 deals with the formulas with local name restrictions (possibly generated via Rule 11). Transformation using P results in the extension of the scope of x to the transformed formula. Similar to Rule 11, name x in φ is renamed to a new name x' (not present as a free name in P). Observe that, Rules 11 and 12 have a similar effect

$$\begin{array}{lrcl}
1(a) & \Pi(P)(tt) & = & tt \qquad 1(b)\ \Pi(P)(\mathit{ff}) = \mathit{ff} \\
2(a) & \Pi(P)(x = y) & = & \begin{cases} tt & \texttt{if } x = y \\ x = y & \texttt{otherwise} \end{cases} \quad 2(b)\ \Pi(P)(x \neq y) = \begin{cases} \mathit{ff} & \texttt{if } x = y \\ x \neq y & \texttt{otherwise} \end{cases} \\
3(a) & \Pi(P)(loc(x)) & = & loc(x) \qquad 3(b)\ \Pi(P)(nloc(x)) = nloc(x) \\
4 & \Pi(P)(\varphi_1 \vee \varphi_2) & = & \Pi(P)(\varphi_1) \vee \Pi(P)(\varphi_2) \\
5 & \Pi(P)(\varphi_1 \wedge \varphi_2) & = & \Pi(P)(\varphi_1) \wedge \Pi(P)(\varphi_2) \\
6 & \Pi(P)(X(\vec{e})) & = & X_P(\vec{e_1}) \text{ where } \vec{e_1} = \vec{e} + fn(P) \\
7 & \Pi(P)(\exists x.\varphi) & = & \exists x.\Pi(P)(\varphi) \quad \Pi(P)(\forall x.\varphi) = \forall x.\Pi(P)(\varphi) \\
8 & \Pi(0)(\varphi) & = & \varphi \\
9 & \Pi(p(\vec{x}))(\varphi) & = & \Pi(P)(\varphi) \quad \text{where } p(\vec{x}) \stackrel{\text{def}}{=} P \\
10 & \Pi(P_1 \mid P_2)(\varphi) & = & \Pi(P_2)(\Pi(P_1)(\varphi)) \\
11 & \Pi((\nu x)P)(\varphi) & = & (\nu x')\Pi(P\{x'/x\})(\varphi) \texttt{ where } x' \cap n(\varphi) = \emptyset \\
12 & \Pi(P)((\nu x)\varphi) & = & (\nu x')(\Pi(P)(\varphi\{x'/x\})) \texttt{ where } x' \notin fn(P) \\
13 & \Pi(a.P)(\langle\alpha\rangle\varphi) & = & \langle\alpha\rangle\Pi(a.P)(\varphi) \texttt{ where } bn(\alpha) \cap fn(a.P) = \emptyset \\
& & & \vee \left\{ \begin{array}{ll} \Pi(P)(\varphi) & \texttt{if } a = \tau \wedge \alpha = \tau \\ x_1 = x_2 \wedge nloc(y_1) \wedge \Pi(P)(\varphi) & \texttt{if } a = \overline{x_1}y_1 \wedge \alpha = \overline{x_2}y_1 \\ x_1 = x_2 \wedge nloc(y_1) \wedge \Pi(P)(\varphi\{y_1/y_2\}) & \texttt{if } a = \overline{x_1}y_1 \wedge \alpha = \overline{x_2}\{y_2\} \\ x_1 = x_2 \wedge loc(y_1) \wedge \Pi(P)(\varphi\{y_1/y_2\}) & \texttt{if } a = \overline{x_1}y_1 \wedge \alpha = \overline{x_2}\nu y_2 \\ x_1 = x_2 \wedge \Pi(P)(\varphi\{y_1/y_2\}) & \texttt{if } a = x_1(y_1) \wedge \alpha = x_2(y_2) \\ x_1 = x_2 \wedge \Pi(P\{y_2/y_1\})(\varphi) & \texttt{if } a = x_1(y_1) \wedge \alpha = x_2y_2 \\ \mathit{ff} & \texttt{otherwise} \end{array} \right\} \\
& & & \vee \left\{ \begin{array}{ll} \langle\overline{a}\rangle\Pi(P)(\varphi), \texttt{ where } bn(a) \cap n(\varphi) = \emptyset & \texttt{if } \alpha = \tau \\ \mathit{ff} & \texttt{otherwise} \end{array} \right\} \\
14 & \Pi(P_1 + P_2)(\langle\alpha\rangle\varphi) & = & \langle\alpha\rangle\Pi(P_1 + P_2)(\varphi) \vee \Pi(P_1)(\langle\alpha\rangle\varphi) \vee \Pi(P_2)(\langle\alpha\rangle\varphi) \\
15 & \Pi([x = y]P)(\varphi) & = & C \wedge \Pi(P)(\varphi) \texttt{ where } C = \begin{cases} tt & \texttt{if } x = y \\ x = y & \texttt{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{ll}
\text{A.} & \Pi(P)(X(\vec{z}) =_\sigma \varphi \cup E) = \{X_P(\vec{z_1}) =_\sigma \Pi(P)(\varphi) \text{ where } (n(\varphi) - \vec{z}) \cap fn(P) = \emptyset) \texttt{ and } \vec{z_1} = \vec{z} + fn(P)\} \\
& \qquad \cup\, \Pi(P)(E) \cup \bigcup\{\Pi(P')(X'(\vec{z_2}) =_{\sigma'} \varphi') \texttt{ s.t } X'_{P'}(\vec{z_3}) \texttt{is a subformula of} \\
& \qquad \Pi(P)(\varphi), \vec{z_3} = \vec{z_2} + fn(P') \texttt{ and } (n(\varphi') - \vec{z_2}) \cap fn(P') = \emptyset)\} \\
\text{B.} & \Pi(P)(\{\}) = (\{\})
\end{array}$$

Fig. 3. Partial Model Checker for π-Calculus

as pulling the ν out using the *structural congruence* rule: $(\nu x)P \mid Q \equiv (\nu x)(P \mid Q)$ *where x does not appear in Q*. Renamings in these two rules correspond to the side condition of the congruence rule.

Rule 13 presents the transformation $\langle\alpha\rangle\varphi$ using prefix process expression $a.P$. The rule relies on three different possibilities following which $a.P$, when composed with an environment, can satisfy $\langle\alpha\rangle\varphi$.

1. The environment makes a move on α satisfying the modal obligation (1st disjunct).
2. $a.P$ satisfies the modal obligation α (2nd disjunct).
3. $\alpha = \tau$ and the environment synchronizes with $a.P$ (the 3rd disjunct), i.e., performs an $\overline{a}$ action.

In Case 1, the side condition demands that the bindings in modal action α does not bind any free names of prefixed process expression. As such we apply alpha-conversion to satisfy the side condition: alpha-conversion renames all the binding occurrences in formula with new names that are disjoint from the free names of the process. In Case 2 there are multiple possibilities depending on the nature of modal action α. Note that if

α is an output or a free output, then formula expression $nloc(y_1)$ is generated meaning that y_1 must not be a local name to satisfy the modal obligation. This is because at the time of transformation, it is not known whether y_1 is a local name or not. Similarly, when α is a bound output modal action, the formula expression $loc(y_1)$ is generated.

In Rule 14, a diamond modal formula is transformed using choice process expression. The result is a disjunction where (a) the first disjunct corresponds to the case where the environment is left with the obligation to satisfy the modal action and (b) the second and the third disjunct, respectively, corresponds to the case where the first or the second process is selected for subsequent transformation.

Finally, Rules A and B correspond to transformation of formula equations. Observe that, we are using equational syntax of the $\mathsf{C}\mu$-calculus. Any property with formula expressions of the form $\sigma X(\overrightarrow{z}).\varphi$ can be converted in linear time to set of equations of the form $X(\overrightarrow{z}) =_\sigma \varphi$. Specifically, given a $\mathsf{C}\mu$-calculus formula φ where each fixed point variable has distinct names, the number of equations in the corresponding equational set is equal to the number of fixed point sub-formulas of φ. Each such sub-formula of the form $\sigma_x X.\varphi_x$ is translated to a equation $X =_{\sigma_x} \psi_x$ where ψ_x is obtained by replacing every occurrences of its sub-formula $\sigma_y Y.\varphi_y$ with Y. For example the formula expression: $\nu X.(\mu Y.([a_1]X \wedge [a_2]Y))$ is translated to $X =_\nu Y$ and $Y =_\mu [a_1]X \wedge [a_2]Y$ where X is the outer-fixed point variable and Y is the inner one. The use of equational form is driven by the fact that transformation can be done in a per-equation basis, instead of keeping track of all the sub-formula expressions of a formula if the transformation was done for non-equational form.

Let $\mathcal{E}$ represent the sets of formula equations. Rules A and B define a function $\Pi : \mathcal{P} \rightarrow \mathcal{E} \rightarrow \mathcal{E}$ that represents the transformer over a set of $\mathsf{C}\mu$-calculus equations. Rule A states that given a formula equation of the form $X(\overrightarrow{z}) =_\sigma \varphi$, transformation leads to the generation of a new equation of the form $X_P(\overrightarrow{z_1}) =_\sigma \Pi(P)(\varphi)$ where $\overrightarrow{z_1}$ is formed by concatenation of $\overrightarrow{z}$ and free names of P. Moreover, if there is a formula expression $X'_{P'}(\overrightarrow{z_3})$ present in $\Pi(P)(\varphi)$, then the corresponding formula equation for $X'(\overrightarrow{z_2})$ is transformed using P', where $\overrightarrow{z_2}$ is formed by removing free names of P' from $\overrightarrow{z_3}$. Rule A also requires that names in the right-hand side of the equation that do not appear in the parameters should be different from any free names of P.

Theorem 1. *Let P and Q be two process expressions, δ a set of substitutions, and l a set of local names. Then for all formulas φ, the following holds:*

$$Q \mid P \models_{\delta,l} \varphi \Leftrightarrow Q \models_{\delta,l} \Pi(P)(\varphi)$$

The proof is by induction on the size of the process expression and the formula. □

Computing Constraints. Given a process $P|\mathsf{0}$ and a formula φ, let $\psi = \Pi(P)(\varphi)$. According to Theorem 1, given a set of constraints δ and a set of local names l, $P \models_{\delta,l} \varphi \Leftrightarrow \mathsf{0} \models_{\delta,l} \psi$. In Figure 4, we present a function $f^l(\psi)$ that, given a set of local names l, computes a set of constraints δ under which $\mathsf{0} \models_{\delta,l} \psi$.

Rules 1 and 2 in Figure 4 are straightforward. In Rules 3 and 4, if one of x and y is a local name, then since local names are different from any other names in the system, $x = y$ is false. In Rule 7, if x occurs in l, then $loc(x)$ is true, otherwise false. Rule 11 evaluates $\langle\alpha\rangle\varphi$ to *ff* because 0 cannot perform any action. In Rule 12, the local name

1. $f^l(tt) = tt$

2. $f^l(ff) = ff$

3. $f^l(x = y) = \left\{ \begin{array}{ll} tt & \texttt{if } x = y \\ ff & \texttt{if } \{x, y\} \cap l \neq \emptyset \\ x = y & \texttt{otherwise} \end{array} \right\}$

4. $f^l(x \neq y) = \left\{ \begin{array}{ll} ff & \texttt{if } x = y \\ tt & \texttt{if } \{x, y\} \cap l \neq \emptyset \\ x \neq y & \texttt{otherwise} \end{array} \right\}$

5. $f^l(\exists x.\varphi) = \exists x.f^l(\varphi)$

6. $f^l(\forall x.\varphi) = \forall x.f^l(\varphi)$

7. $f^l(loc(x)) = \left\{ \begin{array}{l} ff \texttt{ if } x \notin l \\ tt \texttt{ if } x \in l \end{array} \right\}$

8. $f^l(nloc(x)) = \left\{ \begin{array}{l} tt \texttt{ if } x \notin l \\ ff \texttt{ if } x \in l \end{array} \right\}$

9. $f^l(\varphi_1 \wedge \varphi_2) = f^l(\varphi_1) \wedge f^l(\varphi_2)$

10. $f^l(\varphi_1 \vee \varphi_2) = f^l(\varphi_1) \vee f^l(\varphi_2)$

11. $f^l(\langle\alpha\rangle\varphi) = ff$

12. $f^l((\nu x)\varphi) = f^{l \cup \{x\}}(\varphi)$

13. $f^l(X(\overrightarrow{e})) = f^l(\varphi\{\overrightarrow{e}/\overrightarrow{z}\})$ where $X(\overrightarrow{z}) =_\sigma \varphi$

Fig. 4. Computing $f^l(\varphi)$

x is added to l in order to evaluate the $loc(x)$ and $nloc(x)$ predicates. Note that $f^l(\psi)$ generates a formula over equality and disequality expressions and standard constraint solving algorithms are applied to solve the constraints of the form $\exists x.\varphi$ and $\forall x.\varphi$.

Following example illustrates the use of loc and $nloc$ formula expressions.

Example 1. *Given a process* $p(x) \overset{\text{def}}{=} (\nu y)\overline{x}y.p(x)$ *and a formula* $\varphi \equiv X(x) =_\nu \langle\overline{x}\nu z\rangle tt$*:*

$$\begin{aligned} \Pi(p(x))(\varphi) \equiv X_1(x) &=_\nu \Pi((\nu y)\overline{x}y.p(x))(\langle\overline{x}\nu z\rangle tt) \\ &=_\nu (\nu y)\Pi(\overline{x}y.p(x))(\langle\overline{x}\nu z\rangle tt) =_\nu (\nu y)loc(y) \end{aligned}$$

As $f^{\emptyset}((\nu y)loc(y)) = f^{\{y\}}(loc(y)) = tt$*, therefore,* $0 \models_{tt,\emptyset} \Pi(p(x))(\varphi)$ □

In Example 1, when computing $\Pi(\overline{x}y.p(x))(\langle\overline{x}\nu z\rangle tt)$, since (νy) is not in the scope of transformation, the model checker cannot determine if y is a local name. Thus, we generate the constraint $loc(y)$. After the transformation is done, we verify if $\mathtt{0}$ satisfies the resulting formula $(\nu y)loc(y)$. Since y is a local name, $(\nu y)loc(y)$ is evaluated to tt.

4 Verification of Parameterized π-Calculus Systems

We outline here the compositional analysis based technique for verification of parameterized systems where instances of subsystems are represented by finite control π-calculus processes. Let P^n be a system with n instances of π-calculus process P. Consider verifying that the i^{th} instance of above system satisfies a property φ. The result of transforming φ using the i^{th} instance is $\varphi_i = \Pi(P^i)(\varphi)$. Therefore, from Theorem 1, given a set of substitutions δ and a set of local names l, $0 \models_{\delta,l} \varphi_i \Leftrightarrow P^i \models_{\delta,l} \varphi$.

Now consider verifying whether $\forall i.\ P^i \models \varphi$. Let φ'_i be defined as:

$$\varphi'_i = \left\{ \begin{array}{ll} \varphi_1 & \textit{if } i = 1 \\ \varphi'_{i-1} \wedge \varphi_i & \textit{if } i > 1 \end{array} \right.$$

By definition of φ'_i, $(\forall 1 \leq j \leq i.0 \models_{\delta,l} \varphi_j) \Leftrightarrow 0 \models_{\delta,l} \varphi'_i$. Thus, $0 \models_{\delta,l} \varphi'_i$ means that $\forall 1 \leq j \leq i.P^j \models_{\delta,l} \varphi$. If φ'_ω is the limit of sequence $\varphi'_1, \varphi'_2 \ldots$, then, $0 \models_{\delta,l} \varphi'_\omega \Leftrightarrow \forall i \geq 1.P^i \models_{\delta,l} \varphi$.

A dual technique is applied for the verification problem $\exists i.P^i \models \varphi$. Let φ_i'' be defined as:

$$\varphi_i'' = \begin{cases} \varphi_1 & \textit{if } i = 1 \\ \varphi_{i-1}'' \vee \varphi_i & \textit{if } i > 1 \end{cases}$$

In this case, if φ_ω'', the limit of the sequence $\varphi_1'', \varphi_2'', \ldots$, is satisfied by $\mathtt{0}$ under the substitution δ, then $\exists n.P^n \models \varphi$. We say that the series of φ_i' is *contracting* since $\varphi_i' \Rightarrow \varphi_{i-1}'$ and the series of φ_i'' is *relaxing* as $\varphi_{i-1}'' \Rightarrow \varphi_i''$.

Before deploying the above technique for solving verification of parameterized systems, we need to solve the following problems:

1. *Entailment*: To detect whether a limit is reached requires developing the equivalence relation between $\mathtt{C}\mu$-calculus formulas.
2. *Convergence acceleration*: The limit in the chain of $\mathtt{C}\mu$-calculus formulas may not be realized in general. As such, we need to identify a suitable abstraction to the generated formulas to ensure termination of the iterative process.

Entailment. Equivalence checking of formula expressions in logic with explicit fixed points is an EXPTIME-hard problem. Hence we use an approximate, conservative technique for equivalence detection which is safe and can be efficiently applied. First, we check if two formulas are equivalent based on the algorithm in [3]. The algorithm states that syntactically identical formula expressions are semantically equivalent. If the equivalence between formula expressions is not readily understood from their structure, we apply the technique developed in [6]. This technique relies on converting the formula into a labeled transition system, called *formula graphs*, where each state is annotated by a formula expressions and transitions are labeled by various syntactic constructs of $\mathtt{C}\mu$-calculus, e.g., diamond modal action. The equivalence between two formula expressions are determined by checking whether the corresponding formula graphs refine each other. Such graph-based equivalence detection algorithm is more powerful than that relying on textual representation of syntax [3] as the former can effectively extract dependencies between formula variables (see [6]).

Convergence acceleration. To ensure convergence and termination, we develop a widening algorithm that over-approximates a relaxing sequence of φ_i'' and under-approximates the contracting sequence of φ_i'. The core of the technique is to examine two consecutive formula expressions φ_i and φ_j in a sequence and determine their differences. For example, if the formulas are members of a relaxing sequence ($\varphi_i \Rightarrow \varphi_j$), the difference is identified as a disjunct in φ_j. Widening amounts to removing this disjunct and generate a new formula φ_a such that $\varphi_j \Rightarrow \varphi_a$. Similarly, for contracting sequence, we remove the divergence-causing conjuncts. Note that, this type of widening is only applicable to safety and reachability properties where all the boolean connectives in the formula are either $\wedge$ or $\vee$, respectively.

Note that widening leads to an approximation of the limit of the sequence. As such, given a parameterized system P^n and formula φ, if limit φ_ω of a relaxing sequence is realized via widening and $\mathtt{0} \models \varphi_\omega$, we cannot infer that $\exists n.P^n \models \varphi$. However, $\mathtt{0} \not\models \varphi_\omega \Rightarrow \forall n.P^n \not\models \varphi$. Similarly, for contracting sequence, if φ_ω is the limit reached after widening, then $\mathtt{0} \models \varphi_\omega \Rightarrow \forall n.P^n \models \varphi$, while $\mathtt{0} \not\models \varphi_\omega \not\Rightarrow \exists n.P^n \not\models \varphi$.

5 Optimizations

In general, the transformation rules may generate a number of redundant formulas, e.g., two sub-formulas that are equivalent. Redundancies result in formulas that are large and virtually un-manageable. In order to apply the partial model checker to any practical application, we need to develop techniques to remove such redundancies.

In this section, we propose several optimization techniques to reduce the number of formulas generated by transformation. In [6], the redundancy removal technique was solely focused on removing equivalent sub-formulas and used heavy-weight bisimulation checking algorithm on graphical representation of formulas. Such a technique was used off-line, after the formulas have been generated in the first place. In contrast, here we present a number of light-weight techniques that are tightly-coupled with the transformation rules and help to significantly reduce the size of the resulting formulas.

Symmetry Reduction. When the partial model checker generates new formula variables, it names them based on the corresponding process expressions (see Rule 6 in Figure 3). The number of formulas generated can be reduced considerably by exploiting a form of symmetry reduction. For instance, let X be a formula variable, and P and Q be arbitrary process expressions. Note that $\Pi(P|Q)(X) = \Pi(P)(\Pi(Q)(X))$ is a new formula variable of the form $X_{Q,P}$. On the other hand, $\Pi(Q|P)(X) = \Pi(Q)(\Pi(P)(X))$ is $X_{P,Q}$. Hence $X_{P,Q}$ and $X_{Q,P}$ are semantically identical. We avoid creating the two formula variables in the first place, by reducing the suffix process expression to a symmetrically equivalent canonical form. This is done by first reducing the expression to a sequence of parallel-free process expressions (exploiting the associativity of parallel composition), and sorting the sequence by imposing a global total order on the elements (exploiting the commutativity of parallel composition). This optimization is light-weight and may dramatically reduce the number of formulas generated even for applications where symmetry is not obvious (see Section 6).

Optimizing the Choice Rule. The choice rule in Figure 3 may generate redundant formulas. Consider the process definition $p(x,y) \stackrel{\text{def}}{=} x(v).p(x,y) + y(w).p(x,y)$ and the formula $\varphi =_\nu \langle\tau\rangle tt$. $\Pi(p(x,y))(\varphi)$ generates the following formulas.

$$X_1(x,y) =_\nu \langle\tau\rangle tt \vee X_2(x,y) \vee X_3(x,y)$$
$$X_2(x,y) =_\nu \langle\tau\rangle tt \vee \langle\overline{x}\{v\}\rangle X_1(x,y) \quad X_3(x,y) =_\nu \langle\tau\rangle tt \vee \langle\overline{y}\{w\}\rangle X_1(x,y)$$

From the above, we can infer that $X_1(x,y) = \langle\tau\rangle tt \vee \langle\tau\rangle tt \vee \langle\overline{x}\{v\}\rangle X_1(x,y) \vee \langle\tau\rangle tt \vee \langle\overline{y}\{w\}\rangle X_1(x,y)$. We can, however, avoid generating the two redundant sub-formulas $\langle\tau\rangle tt$ using the following revised "+" rule.

$$\Pi(P_1+P_2)(\langle\alpha\rangle\varphi) = \langle\alpha\rangle\Pi(P_1+P_2)(\varphi) \ \vee\ \Pi'(P_1)(\langle\alpha\rangle\varphi) \ \vee\ \Pi'(P_2)(\langle\alpha\rangle\varphi)$$

Π' differs from Π in Rule 13 where modal obligation $\langle\alpha\rangle$ is not imposed on the environment.

Simplification Techniques. Apart from symmetry-based simplification, we also remove redundant sub-formulas and use the simplifying equations originally proposed in [3]. The most frequently used simplification techniques are constant propagation (e.g. $X = \langle\alpha\rangle X_1, X_1 = tt \Rightarrow X = \langle\alpha\rangle tt$), and unguardedness removal(e.g. $X = \langle\alpha\rangle X_1, X_1 =$

$X_2 \Rightarrow X = \langle\alpha\rangle X_2$). These simplification techniques help to quickly detect if two formulas are equivalent.

Environment-Based Reduction. Consider Rule 13 in Figure 3. Process $a.P$ either leaves the environment to perform an α action (1st disjunct) or an $\overline{a}$ action if $\alpha = \tau$ (3rd disjunct), or $a.P$ itself performs an α action (2nd disjunct). However, if the environment cannot perform an α or an $\overline{a}$ action, then the 1st and the 3rd disjuncts need not be generated. For instance, consider the example given in Figure 1. Given a formula φ, we first use $p(x)$ to transform φ under the environment $q(x) \mid \ldots \mid q(x)$. From the specification, process $q(x)$ cannot synchronize with itself, thus the model checker does not need to leave the environment to perform a τ action. However, this optimization requires the knowledge of the environment, thereby rendering the model checker of Figure 3 no longer compositional. Moreover, the assertion $(P|Q) \models \varphi \Leftrightarrow Q \models \Pi(P)(\varphi)$ now holds only for those Q that are consistent with the knowledge of the environment used to perform this optimization.

When using P to transform a formula under the environment Q, we check: 1) What are the actions of P with which Q cannot synchronize? 2) Can Q perform a τ transition? These can be easily determined for value-passing calculus by parsing the specification, but are more difficult for the π-calculus due to link passing. Thus we compute the set of actions conservatively: if we do not know whether one process can synchronize with another, then we conservatively assume that such synchronization exists between the two processes. The environment information is propagated in the model checker. The details are given in [34]. This optimization may reduce the size of each formula and sometimes reduces the number of formulas generated (see Section 6).

Eliminating Constraints Based on the Types of Channels. This optimization is applied to whenever the formulas generated are guarded by equality and disequality constraints. Under certain conditions, we can determine whether a constraint generated is unsatisfiable. For instance, assume that we keep track of the set of all names that have been extruded from their initial scope. Then if x has never been extruded and y is a bound name of an input action, then $x = y$ is never true. We use a simple type system to determine whether a channel could have been extruded.

6 Preliminary Experimental Results

In this section, we show the effectiveness of our technique to verify parameterized versions of several small but non-trivial examples. The examples include those with a fixed process interconnection, namely, *Token ring*, a ring of n token-passing processes, and *Spin lock*, a simple locking protocol where n processes compete to acquire a single common resource. We also include examples with dynamically changing interconnection between processes, namely *Printer*, where n clients use a single print server to mediate access to a printer, and *Server* [9], where n file readers serve web page read requests. We also evaluate the performance of our model checker on the *Handover procedure* [29] (which maintains the connectedness of a mobile station in a cellular network when the station crosses cell boundaries) to verify a single instance of the system.

The experimental results are shown in Fig. 5. All reported performance data were obtained on a 1.4GHz Pentium M machine with 512MB of memory running Red Hat

Benchmark	Property	Summary		# Formulas					Time (sec.)				
		# Iter	Widen (Y/N)	Orig	Sym	Env	All	Conv	Orig	Sym	Env	All	Conv
Token ring	deadlock freedom	3	Y	86	45	–	45	40	1.93	0.56	–	0.56	0.37
Spin lock	mutual exclusion	3	N	398	192	364	181	181	34.96	7.8	23.37	5.11	5.29
	deadlock freedom	3	Y	160	80	160	80	64	6.89	1.49	4.62	0.99	1.35
Printer	deadlock freedom	3	Y	55	29	–	29	22	1.03	0.29	–	0.29	0.20
Server	order preservation	4	Y	1440	1270	241	239	172	361.58	280	10.17	10.07	5.24

Fig. 5. Experimental Results

Linux 9.0. The figure is divided broadly into three parts. The verification results for the different systems and properties are summarized in the first part (columns under "Summary"). In that part, the number of iterations for the sequence to converge, and whether widening was needed appear in columns "# Iter", and "Widen" respectively. For all the cases listed in the figure, we can conclude that the property holds for all instances of the parameterized system, even when widening was used to enforce convergence.

The second and third parts of the table, namely, columns under "# Formulas" and "Time", present the performance results (number of formulas processed and the CPU time taken, resp.) for the examples. The columns "Conv" list the total number of formulas and time to compute the formula sequence, including the time taken to perform convergence check and widening (when needed). The other columns list the same statistics to compute the formula sequence (length of the sequence is same as the number of iterations) but without checking for convergence or applying widening. The columns "Orig", "Sym", "Env" and "All" list the statistics when no optimizations, symmetry reduction, environment-based reduction and all optimizations described in Section 5 (resp.) are applied. In the table "–" indicates that the optimization is inapplicable. The performance results show the effectiveness of the optimizations: the overheads of performing the optimizations are easily offset by the reductions enabled by the optimizations. Widening sometimes reduces formula sizes sufficiently (see Token Ring, Printer, Server), consequently saving enough time to offset that needed to perform the operation. In all benchmarks, the memory requirement of the model checker without optimizations is always higher than that with optimizations (all $<$ 12MB), and hence the corresponding results are not shown.

Finally, we applied the compositional model checker to verify a single instance of the Handover protocol (1 mobile and 2 base stations). Even with all optimizations enabled, it takes 12s to verify the deadlock freedom property for this instance. In contrast, the non-compositional model checker MMC can verify this instance in less than a second. This indicates that the compositional checker is unsuitable for use, as it stands, for routine verification of non-parameterized systems. When we attempted to verify another instance of the protocol (with 2 mobile stations), the compositional checker generated more formulas than can be handled by our prototype implementation.

7 Conclusion

In this paper, we presented an automatic technique for verifying parameterized systems that consist of a number of instances of finite-control π-calculus processes. This technique uses a sufficiently expressive logic, Cμ-calculus, to represent properties, and is based on a compositional model checker for the π-calculus.

Since the technique is based on a compositional model checker, each process instance is verified in an “open” (unknown) environment. Hence in this approach, we consider a lot more potential system behaviors than any instance of the parameterized system can exhibit. This leads to generation of large number of formulas at each step. Optimization aim at reducing this potential blow-up. Among these, the environment-based reduction attempts to construct an environment for each process that is significantly more restricted than the open environment. This is based on the capabilities of the other processes in the parameterized system (e.g. channels they can communicate on). Even a relatively simple version of this optimization presented in this paper, which is based on a very coarse notion of capabilities of processes, results in significant reduction in verification time (e.g. Server example in Fig. 5). We are currently investigating heavier-weight but more effective optimizations that would make it possible to use our technique on realistic parameterized systems such as the Handover protocol.

References

1. R. Alur and T. Henzinger. Reactive modules. In *LICS*, 1996.
2. R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *CAV*, pages 548–562, 2005.
3. H.R. Andersen. Partial model checking (extended abstract). In *LICS*, 1995.
4. H.R. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal mu-calculus. In *LICS*, 1994.
5. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *Computer Aided Verification*, 2001.
6. S. Basu and C. R. Ramakrishnan. Compositional analysis for verification of parameterized systems. In *Proceedings of TACAS*, pages 315–330, 2003.
7. S. Berezin and D. Gurov. A compositional proof system for the modal mu-calculus and CCS. Technical Report CMU-CS-97-105, CMU, 1997.
8. J. Bradfield and C. Stirling. *Modal logics and mu-calculi: an introduction (In the Handbook of Process Algebra)*, pages 293–330. Elsevier, 2001.
9. S. Chaki, S.K.Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *Proceedings of POPL*, pages 45 – 57, 2002.
10. E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Transactions on Programming Languages and Systems*, 1997.
11. M. Dam. Proof systems for pi-calculus logics. *Logic for Concurrency and Synchronisation*, 2001.
12. G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Computer Aided Verification*, 2000.
13. E.A. Emerson and K.S. Namjoshi. Reasoning about rings. In *POPL*, 1995.
14. E.A. Emerson and K.S. Namjoshi. Automated verification of parameterized synchronous systems. In *Computer Aided Verification*. Lecture Notes in Computer Science, 1996.
15. E.A. Emerson and K.S. Namjoshi. On model checking for non-deterministic infinite state systems. In *LICS*, 1998.
16. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS*, 1999.
17. R. Cleaveland G. Bhat. Efficient model checking via the equational μ-calculus. In *LICS*, pages 304–312, 1996.
18. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 1994.

19. T. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee. In *CAV*, 1998.
20. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
21. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1–3):121–163, 2004.
22. C.N. Ip and D.L. Dill. Verifying systems with replicated components in murphi. *Formal Methods in System Design*, 1999.
23. Y. Kesten and A. Pnueli. Control and data abstraction:the cornerstones of pratical formal verification. *International Journal on Software tools for Technology*, 2000.
24. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 1983.
25. H. Lin. Symbolic bisimulation and proof systems for the π-calculus. Technical report, School of Cognitive and Computer Science, U. of Sussex, UK, 1994.
26. K.L. McMillan. Compositional rule for hardware design refinement. In *CAV*, 1997.
27. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
28. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–77, 1992.
29. F. Orava and J. Parrow. An algebraic verification of a mobile network. *Journal of Formal Aspects of Computing*, 4:497–543, 1992.
30. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2001.
31. A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *Computer Aided Verification*, 2000.
32. H. Song and K. J. Compton. Verifying pi-calculus processes by Promela translation. Technical Report CSE-TR-472-03, Univ. of Michigan, 2003.
33. B. Victor. The Mobility Workbench user's guide. Technical report, Department of Computer Systems, Uppsala University, Sweden, 1995.
34. P. Yang, S. Basu, and C. R. Ramakrishnan. Parameterized verification of π-calculus systems, 2006. Available at `http://www.lmc.cs.sunysb.edu/~pyang/ptech.pdf`.
35. P. Yang, C. R. Ramakrishnan, and S. A. Smolka. A logical encoding of the π-calculus: Model checking mobile processes using tabled resolution. In *Proceedings of VMCAI*, 2003. Extended version in *Software Tools for Technology Transfer*, 6(1):38-66,2004.
36. P. Yang, C. R. Ramakrishnan, and S. A. Smolka. A provably correct compiler for efficient model checking of mobile processes. In *Proceedings of PADL*, 2005.
37. L. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures*, 30(3–4):139–169, 2004.