Compositional Analysis for Verification of Parameterized Systems

Samik Basu [a] and C. R. Ramakrishnan [b]

[a] *Dept. of Computer Science, Iowa State University, Ames, IA 50014*

[b] *Dept. of Computer Science, Stony Brook University, Stony Brook, NY 11794*

Abstract

Many safety-critical systems that have been considered by the verification community are parameterized by the number of concurrent components in the system, and hence describe an infinite family of systems. Traditional model checking techniques can only be used to verify specific instances of this family. In this paper, we present a technique based on compositional model checking and program analysis for automatic verification of infinite families of systems. The technique views a parameterized system as an expression in a process algebra (CCS) and interprets this expression over a domain of formulas (modal mu-calculus), considering a process as a property transformer. The transformers are constructed using partial model checking techniques. At its core, our technique solves the verification problem by finding the limit of a chain of formulas. We present a widening operation to find such a limit for properties expressible in a subset of modal mu-calculus. We describe the verification of a number of parameterized systems using our technique to demonstrate its utility.

Key words: Parameterized systems, compositional model checking, formula equivalence, acceleration, widening.

1 Introduction

Model checking is a widely used approach for verifying whether a system specification possesses a property expressed in temporal logic [13, 41]. Many efficient verification tools have been developed based on approaches such as

Email addresses: `sbasu@cs.iastate.edu` (Samik Basu), `cram@cs.sunysb.edu` (C. R. Ramakrishnan).

$$\mathtt{P} \stackrel{def}{=} a.\mathtt{P}$$
$$\mathtt{C} \stackrel{def}{=} \overline{a}.\mathtt{C}$$
$$\mathtt{sys}(n) \stackrel{def}{=} (\mathtt{P}^n|\mathtt{C})\backslash\{a\}$$

(a)

$$\begin{array}{lll} \varphi & \equiv X & \textit{where } X =_\nu \langle\tau\rangle tt \wedge [\tau]X \\ \varphi_c & \equiv Y & \textit{where } Y =_\nu \langle\tau, a\rangle tt \wedge [\tau, a]Y \\ \varphi_1 & \equiv Z_1 & \textit{where } Z_1 =_\nu [\tau, a, \overline{a}]Z_1 \\ \varphi_2 & \equiv Z_2 & \textit{where } Z_2 =_\nu [\tau, a, \overline{a}]Z_2 \end{array}$$

(b)

Fig. 1. (a) Parameterized System with one consumer and arbitrary number of producers; (b) Deadlock-freedom formula φ and property transformation results.

explicit-state [29], symbolic [12] and compositional [4] techniques. Traditionally, model checkers have been restricted to the verification of finite-state systems, although recent research on constraint-based techniques (e.g. [19]), symmetry reduction [30], data independence [44], and symbolic checking with rich assertional languages [32] have extended model checking techniques to certain classes of infinite-state systems.

The Driving Problem. In this paper, we focus on an interesting class of infinite state systems, *parameterized systems*. A parameterized system describes an infinite family of (typically finite-state) systems; instances of the family can be obtained by fixing the parameters. Consider a simple example of parameterized producer-consumer system shown in Figure 1(a). A producer process `P` performs an action a and continues to behave as `P`. Similarly, the consumer process `C` repeatedly performs action $\overline{a}$. The processes communicate by synchronization on $\overline{a}$ and a actions. The parameterized system `sys(`n`)` is specified as parallel composition of n producers, denoted by P^n, and a consumers (C). Our objective is to verify deadlock-freedom property for *all instances* of the system `sys`.

Models of many safety-critical systems are parameterized: e.g., resource arbitration protocols, communication protocols, etc. Traditionally, model checkers have been used to verify specific instances of the infinite family described by a parameterized system: e.g., to verify that a mutual exclusion protocol is correct for fixed numbers of objects and threads [9]. Clearly, this strategy cannot be used to verify *all instances* of the infinite family of systems.

Our Solution. In this paper we present an automatic technique for checking whether any or all arbitrary instances of an infinite family of systems possess a given temporal property. At a high level, our solution to the verification problem is analogous to program analysis. Each instance of a parameterized system is viewed as an expression in a process algebra (specifically, CCS [38]). We then interpret these process algebraic expressions over a domain consisting of formulas in an expressive temporal logic (specifically, the alternation-free modal mu-calculus [33]). The interpretation is based on associating a *property transformer* Π for each process p in the parameterized system. Given a system s consisting of p concurrently composed with an arbitrary environment e, Π captures the relationship between properties that hold in the environment

e and the properties that hold in the system s. For instance, consider the process P in Figure 1(a), and a system s consisting of P composed in parallel with an (unknown) environment e. Consider the verification of the property that s does an internal τ action. The τ action can either be solely due to the environment e, or a result of P making an a action that is synchronized with an $\overline{a}$ action of e. Thus, process P can be seen as transforming the property "do a τ action" on system s to the the property "do an $\overline{a}$ or a τ action" on the environment e.

The property transformer for a given process is generated using the notion of *quotienting* due to [3]. Based on the property transformer, we define a chain of mu-calculus formulas whose limit characterizes the behavior of an arbitrary instance of the parameterized system. Consider the problem of verifying deadlock-freedom for the parameterized system `sys(`n`)` for all $n \geq 1$. The formula to be checked for the entire system is given in Figure 1(b) as φ. φ is a greatest fixed point formula where the first conjunct is satisfied by states with at least one outgoing τ transition while the second disjunct requires that φ is satisfied at every destination states reachable after a τ transition.

We first compute the property expected of the producers alone, by transforming the property φ using the property transformer for C process. The resulting "quotient" property is the formula φ_c in the figure. Intuitively, φ_c states that φ can be modeled by an environment composed in parallel to process C if the environment can perform infinitely many a or τ actions. Therefore, if $\mathtt{P}^n \models \varphi_c$ then `sys(`n`)` $\models \varphi$. Next, we transform φ_c using the property transformer for process P. The resultant property, φ_1 in the figure, is the "residue" that captures the property $\mathtt{P}^{n-1}$ must satisfy, for `sys(`n`)` to satisfy φ. Repeated applications of the property transformer of P yields a sequence $\varphi_1, \varphi_2, \ldots$, which are "residues" corresponding to processes $\mathtt{P}^{n-1}$, $\mathtt{P}^{n-2}$, ... In our example, we see that this sequence converges immediately, with $\varphi_i = \varphi_1$ for all $i \geq 1$. We can thus conclude that if $\mathtt{0} \models \varphi_1$ then $\forall n \in N$ `sys(`n`)` $\models \varphi$ (0 denotes a deadlocked process which does not interact with its environment). The above discussion presents a high level view of the technique used to verify properties for all or any members of a parameterized system. The actual technique is a little more complex, keeping track of various restriction and relabeling operations applied to the processes (see Sections 3 and 4 for details).

The sequence of residues can be easily seen as a chain (i.e. the elements of sequence are nondecreasing with respect to a partial order). However, the sequence may not have a limit since the domain of interpretation, the modal mu-calculus, has infinite ascending chains. Nevertheless, we find that the iterative computation of the limit does converge for a number of example parameterized systems. Moreover, we define a widening operation to accelerate the convergence, and in some cases guarantee termination.

Related Work. Verification of parameterized systems is known to be undecidable in general [5]. A number of techniques have been proposed with varying degree of user intervention ranging from fully automatic techniques (mostly sound but incomplete), which focus on domain of representation of systems, to program transformation-based methods capable of inferring the underlying structure of induction proofs. Other than the degree of dependence on user guidance, the techniques can also be classified on the basis of parameterized systems on which they are applicable: (a) systems where subsystems interact via shared variables (asynchronous) and (b) systems where communication mechanism depends on message passing (synchronous).

One of the automatic approaches for synchronously communicating systems involves reduction of the infinite-state verification problem to an equivalent finite-state one by identifying an appropriate cut-off value for the parameter corresponding to the system and the temporal property [21, 22, 31, 8]. Cache coherence and unidirectional token ring protocols have been successfully verified using these techniques. Another approach focuses on identifying an appropriate representation technique for parameterized system: e.g. counting abstraction with arithmetic constraints [18], covering graphs [23, 24], and context-free grammars [14]. Such representation mechanisms have been effectively used to generate network invariants capturing the common aspects of the members of an infinite family. However, generation of network invariants can be automated only for a class of systems with restricted communication patterns (ring and linear topologies [34, 43]). Most of these techniques are not applicable directly to systems communicating by shared variables. In the realm of asynchronous systems, Kesten and Pnueli [32] present the importance of appropriate abstractions to generate invariants of parameterized systems. Pnueli and Shahar [40] use a representation mechanism based on regular languages to symbolically verify safety and liveness properties of parameterized systems. More recently, automatic techniques based on identification of cut-off of the parameters have been proposed for verifying a wide range of parameterized systems using a rich class of data objects and operations [39, 7].

In this paper we focus only on synchronously communicating systems. Our technique, unlike the representation-based approaches, directly manipulates processes specified in standard process algebra. Moreover, being based on program analysis, our technique can be applied with little or no knowledge of the internals of the system and without regard to the network topology of the system to be verified. This is in contrast to the representation-based techniques whose success depends on the clever choice of representations. Our technique is based on applying compositional model checking techniques for the automatic verification of infinite families of systems. Considerable amount of research has been done on using assume-guarantee reasoning for constructing compositional proofs [35, 25, 2, 36, 10, 27]. However, these methods typically need user guidance. Recently, [28] developed automatic assume-guarantee veri-

fication methodology in the setting of multi-threaded C programs where appropriate approximations of single thread behavior is identified using abstraction-refinement techniques. Another technique, proposed by [16], aims at automatically identifying the assumptions (obligations of the environment in a 2-process system) by iterative application of model checker. Closely related to our work are the compositional model checker of [4] and the partial model checker of [3]. The latter work defines property transformers for parallel composition of sequential automata, while we generalize the transformers for arbitrary CCS processes. We also present a simulation-based procedure to detect equivalence of formulas which is strictly more powerful than the equivalence detection technique proposed in [3].

Contributions. We present a technique for automatic verification of parameterized systems representing an infinite family of finite-state systems.

(1) We develop a compositional model checker for CCS [38] and use this model checker to generate property transformers (Section 3).
(2) Given a verification problem over a parameterized system, we use property transformers to define a sequence of mu-calculus formulas, whose limit characterizes the property of the parameterized system (Section 4).
(3) Computing the limit of a chain of mu-calculus formulas involves checking the equivalence of formulas. We present a novel polynomial-time heuristic for the equivalence checking problem based on constructing automata from the formulas and testing them for simulation (Section 5).
(4) To guarantee convergence of the iterative procedure, we define acceleration and widening operators (based on widening techniques used in type analysis) for mu-calculus formulas. (Section 6).
(5) We show the usefulness of the technique by presenting its application in verifying protocols over token passing rings (Milner's cycle of schedulers [3]), mutual exclusion protocols (Java meta-lock [1]), and cache coherence protocols [18] (Section 7).

2 Preliminaries

We briefly outline the syntax of the process algebra CCS [38] and the logic modal mu-calculus [11] used in the rest of the paper.

CCS and labeled transition systems. CCS is a simple process algebra that can be used to specify concurrent systems. Below we describe the syntax of expressions in *basic* CCS:

$$\mathcal{P} \rightarrow \mathsf{0} \mid A \mid a.\mathcal{P} \mid \mathcal{P} + \mathcal{P} \mid \mathcal{P}'|'\mathcal{P} \mid \mathcal{P}\backslash L \mid \mathcal{P}[f]$$

In the above, $\mathtt{0}$ denotes a deadlocked process. A ranges over process names (agents) and a ranges over a set of actions $Act = \mathcal{L} \cup \overline{\mathcal{L}} \cup \tau$, where τ represents an internal action and $\mathcal{L}$ is a set of labels and $\overline{\mathcal{L}}$ is such that $a \in \mathcal{L} \Leftrightarrow \overline{a} \in \overline{\mathcal{L}}$. Finally, L ranges over the powerset of $\mathcal{L}$, and $f : \mathcal{L} \to \mathcal{L}$. The operators '.', '+', '|', '\' and '[·]' are called prefix, choice, parallel composition, restriction and relabeling respectively. A CCS specification consists of a set of process definitions, denoted by D, of the form $A \stackrel{def}{=} P$, where $P \in \mathcal{P}$. Each agent used in P, in turn, appears on the left hand side of some process definition in D. Note that process definitions may be recursive.

A labeled transition system $(S, \to)$ is specified by a set of *states* S and a transition relation $\to \subseteq S \times Act \times S$. The operational semantics of CCS expressions is given in terms of labeled transition systems where states represent CCS expressions. See [38] for full treatment of the semantics of CCS.

The modal mu-calculus. The modal mu-calculus [33] is an expressive temporal logic with explicit greatest and least fixed point operators. Following [15, 3], we use the *equational* form of mu-calculus. The syntax of *formulas* in modal mu-calculus over a set of propositional variables X and actions Act is given by the following grammar :

$$\Phi \to tt \mid ff \mid X \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \langle \alpha \rangle \Phi \mid [\alpha]\Phi.$$

In the above, α specifies a set of actions in positive form (as $\beta \subseteq Act$) or negative form (as $-\beta$, where $\beta \subseteq Act$). $\langle \alpha \rangle \Phi$ states that there exists an action in α following which formula Φ holds true, while $[\alpha]\Phi$ states that after every action in α, Φ is satisfied. Propositional constants tt and ff represent true and false respectively. The variables used in a mu-calculus formula are defined using a *sequence* of equations where the ith equation has the form: $X_i =_\mu \varphi_i$ or $X_i =_\nu \varphi_i$, where $\varphi_i \in \Phi$. The least and greatest fixed point symbols μ and ν are said to represent the *sign* of the equation. In the remainder of the paper,we use σ, ranging over $\{\mu, \nu\}$ to denote the sign of an arbitrary equation. We assume that each variable occurs exactly once on the left hand side of an equation. The variable X_1 defined by the first equation is called the *top variable*. The set of equations representing some property is denoted by E. The set of all mu-calculus equations is denoted by $\mathcal{E}$.

Model Checking. Given a labeled transition system $(S, \to)$, the semantics of mu-calculus formulas are stated such that each formula denotes a subset of S. Refer to [11] for semantics of mu-calculus. We say that a mu-calculus formula φ holds at a state s, if s is in the model of φ ($s \models \varphi$).

3 Partial Model Checking

Our technique for verification of parameterized systems is based on viewing a process as a property transformer. We generate property transformers using a partial model checker [3]. Consider the verification of a formula φ over a process expression of the form $P|Q$. Given φ and P we generate the obligation φ' on Q such that $P|Q \models \varphi$ iff $Q \models \varphi'$. Thus we view P as *transforming* the obligation φ on $P|Q$ to the obligation φ' on Q. This transformation is called *quotienting* in [3], where it is defined for modal mu-calculus properties and systems specified by LTSs.

In Figure 2 we define the property transformer using a function $\Pi : (P \times L \times F) \to \Phi \to \Phi$ where L is 2^{Act} and F is a set of partial injective functions (relabeling functions) $f : Act \to Act$ such that $f(x) \neq x$. We use $\perp$ to denote empty relabeling function which is undefined everywhere. We define the composition of two relabeling functions $h = f \circ g$ such that h is undefined if f and g are both undefined, $h(x) = f(x)$ if g is undefined, $h(x) = g(x)$ if f is undefined; otherwise $h(x) = f(g(x))$. Φ is the set of modal mu-calculus formulas. Finally P is the set of all CCS process expressions. A process expression is said to be *well-named* if all relabeling operations of the form $Q[f]$ are such that the set of visible actions of process Q is disjoint from the range of function f.

The transformer $\Pi^L_f(P)$ considers process P under a set of restricted actions (L) and a relabeling function (f). The transformer generates a formula ψ as the obligation of the environment of process P such that (a) modal actions are suitably relabeled by f and (b) environment is not allowed to synchronize on actions in L. The transformer $\Pi^L_f(P)$ transforms φ and generates ψ defined over fixed point variables $X_{P,f,L}$, where φ is defined over variables in X.

The set of visible actions of process P is denoted by $vn(P)$. The names of formula φ are the set of modal actions in φ and is denoted by $n(\varphi)$. Note that, $n(X) = n(\varphi)$, where $X =_\sigma \varphi$. Range of relabeling f is the set of actions a such that $f : x \to a$.

Rules 1 through 5 (Figure 2) for the property transformer correspond to propositional constants, boolean connectives, formula variables. The property transformer for the zero/deadlocked process (Rule 6), which is the identity of the parallel composition operator of CCS, has the identity function as its property transformer. Rule 7 states that the property transformer for an agent is the property transformer of the process expression used to define the agent.

Property transformer of a process with relabeling function f_p is property transformer of the process under new relabeling function by composing f_p with

$$
\begin{array}{lrcl}
1. & \Pi_f^L(P)(tt) & = & tt \\
2. & \Pi_f^L(P)(f\!f) & = & f\!f \\
3. & \Pi_f^L(P)(\varphi_1 \vee \varphi_2) & = & \Pi_f^L(P)(\varphi_1) \vee \Pi_f^L(P)(\varphi_2) \\
4. & \Pi_f^L(P)(\varphi_1 \wedge \varphi_2) & = & \Pi_f^L(P)(\varphi_1) \wedge \Pi_f^L(P)(\varphi_2) \\
5. & \Pi_f^L(P)(X) & = & X_{P,f,L} \\
6. & \Pi_f^L(0)(\varphi) & = & \varphi \\
7. & \Pi_f^L(A)(\varphi) & = & \Pi_f^L(P)(\varphi) \quad \texttt{if } A \overset{def}{=} P \in D \\
8. & \Pi_f^L(P[f_p])(\varphi) & = & \Pi_{f \circ f_p}^L(P)(\varphi) \\
9. & \Pi_f^L(P \backslash L_p)(\varphi) & = & \Pi_f^{L \cup L'}(P[L'/L_p])(\varphi) \\
 & & & \texttt{where } L' \cap (n(\varphi) \cup vn(P) \cup range(f) \cup L) = \{\} \\
10. & \Pi_f^L(P_1|P_2)(\varphi) & = & \Pi_f^L(P_2)(\Pi_f^{\{\}}(P_1)(\varphi)) \\
11. & \Pi_f^L(a.P)(\langle\alpha\rangle\varphi) & = & \langle\alpha\rangle\Pi_f^L(a.P)(\varphi) \vee \left\{ \begin{array}{ll} \Pi_f^L(P)(\varphi) & \texttt{if } f(a) \in \alpha \\ f\!f & \texttt{otherwise} \end{array} \right\} \\
 & & & \vee \left\{ \begin{array}{ll} \langle\overline{f(a)}\rangle\Pi_f^L(P)(\varphi) & \texttt{if } \tau \in \alpha \ \wedge\ \overline{f(a)} \notin L \\ f\!f & \texttt{otherwise} \end{array} \right\} \\
12. & \Pi_f^L(a.P)([\alpha]\varphi) & = & [\alpha]\Pi_f^L(a.P)(\varphi) \wedge \left\{ \begin{array}{ll} \Pi_f^L(P)(\varphi) & \texttt{if } f(a) \in \alpha \\ tt & \texttt{otherwise} \end{array} \right\} \\
 & & & \wedge \left\{ \begin{array}{ll} [\overline{f(a)}]\Pi_f^L(P)(\varphi) & \texttt{if } \tau \in \alpha \ \wedge\ \overline{f(a)} \notin L \\ tt & \texttt{otherwise} \end{array} \right\} \\
13. & \Pi_f^L(P_1 + P_2)(\langle\alpha\rangle\varphi) & = & \langle\alpha\rangle\Pi_f^L(P_1 + P_2)(\varphi) \vee \Pi_f^L(P_1)(\langle\alpha\rangle\varphi) \vee \Pi_f^L(P_2)(\langle\alpha\rangle\varphi) \\
14. & \Pi_f^L(P_1 + P_2)([\alpha]\varphi) & = & [\alpha]\Pi_f^L(P_1 + P_2)(\varphi) \wedge \Pi_f^L(P_1)([\alpha]\varphi) \wedge \Pi_f^L(P_2)([\alpha]\varphi) \\
 & & & \\
A. & \Pi_f^L(P)(X =_\sigma \varphi \cup E) & = & X_{P,f,L} =_\sigma \Pi_f^L(P)(\varphi) \ \cup\ \Pi_f^L(P)(E) \ \cup \\
 & & & \{\bigcup(\Pi_{F'}^{L'}(P')(X' =_{\sigma'} \varphi') \texttt{ s.t } X'_{P',F',L'} \texttt{is present in} \\
 & & & \qquad \Pi_f^L(P)(\varphi),\ X' =_{\sigma'} \varphi' \in F\} \\
B. & \Pi_f^L(P)(\{\}) & = & \{\}
\end{array}
$$

Fig. 2. Partial Model Checker for CCS

existing relabeling function f (Rule 8). Rule 9 presents the property transformer for a process with restriction L_p. The restricted actions are mapped to a set of new names (L'). This set is disjoint from the set of actions in the formula ($n(\varphi)$), visible actions of process ($vn(P)$) and restricted(L) and relabeled($range(f)$) actions of the transformer.

Rule 10 captures the compositionality of property transformers: the property transformer for a parallel composition of processes is simply the function composition of the individual property transformers with appropriate restrictions and relabeling.

Rule 11 arises from the fact that $a.P|Q$ may satisfy $\langle\alpha\rangle\varphi$ in one of the following three ways:

(1) Q does an α action to Q' leaving $a.P|Q'$ to satisfy φ. In this case, the obligation on Q is to do an α action, followed by satisfying the obligation

$$\boxed{\Pi^{\{\}}_{\perp}(\mathtt{C})(X =_\nu \langle\tau\rangle tt \wedge [\tau]X)}$$

$$\begin{array}{llll}
\text{I.} & X_{\mathtt{C},\perp,\{\}} & =_\nu \Pi^{\{\}}_{\perp}(\mathtt{C})(\langle\tau\rangle tt \wedge [\tau]X) & \text{Rule A}\\
 & & =_\nu \Pi^{\{\}}_{\perp}(\mathtt{C})(\langle\tau\rangle tt) \wedge \Pi^{\{\}}_{\perp}(\mathtt{C})([\tau]X) & \text{Rule 4}\\
 & & =_\nu \Pi^{\{\}}_{\perp}(\overline{a}.\mathtt{C})(\langle\tau\rangle tt) \wedge \Pi^{\{\}}_{\perp}(\overline{a}.\mathtt{C})([\tau]X) & \text{Rule 7}\\
 & & =_\nu \langle\tau, a\rangle tt \wedge ([\tau]\Pi^{\{\}}_{\perp}(\overline{a}.\mathtt{C})(X) \wedge [a]\Pi^{\{\}}_{\perp}(\mathtt{C})(X)) & \text{Rules 11,12,1}\\
 & & =_\nu \langle\tau, a\rangle tt \wedge ([\tau]X_{\overline{a}.\mathtt{C},\perp,\{\}} \wedge [a]X_{\mathtt{C},\perp,\{\}}) & \text{Rule 5}\\
 & & & \\
\text{II.} & X_{\overline{a}.\mathtt{C},\perp,\{\}} & =_\nu \Pi^{\{\}}_{\perp}(\overline{a}.\mathtt{C})(\langle\tau\rangle tt \wedge [\tau]X) & \text{Rule A}\\
 & & =_\nu \Pi^{\{\}}_{\perp}(\overline{a}.\mathtt{C})(\langle\tau\rangle tt) \wedge \Pi^{\{\}}_{\perp}(\overline{a}.\mathtt{C})([\tau]X) & \text{Rule 4}\\
 & & =_\nu \langle\tau, a\rangle tt \wedge ([\tau]\Pi^{\{\}}_{\perp}(\overline{a}.\mathtt{C})(X) \wedge [a]\Pi^{\{\}}_{\perp}(\mathtt{C})(X)) & \text{Rules 11,12,1}\\
 & & =_\nu \langle\tau, a\rangle tt \wedge ([\tau]X_{\overline{a}.\mathtt{C},\perp,\{\}} \wedge [a]X_{\mathtt{C},\perp,\{\}}) & \text{Rule 5}
\end{array}$$

Fig. 3. Transformation of property φ in Figure 1(b) using process `C` in Figure 1(a)

left by $a.P$ due to φ (first disjunct in the rhs of Rule 11).

(2) $a \in \alpha$ and P does the a action, leaving $P|Q$ to satisfy φ. In this case the obligation on Q is simply the obligation left by P due to φ (second disjunct in the rhs of Rule 11).

(3) $\tau \in \alpha$, P does an a action that synchronizes with an $\overline{a}$ action by Q to produce the necessary τ action. This means that the obligation on Q is to first produce an $\overline{a}$ action and then satisfy whatever obligation is left by P due to φ (third disjunct of Rule 11).

Note that, property transformer of P, under a set of restricted actions L, does not permit the environment Q to synchronize on any action present in L. The third disjunct generates modal obligation for the environment on the action $\overline{f(a)}$ only when $\overline{f(a)} \notin L$. Rule 12 is the dual of Rule 11.

Rule 13 presents the property transformer for process with choice operator $(P_1 + P_2)$. It is defined by considering three different cases. In the first disjunct, selection of the processes P_1 and P_2 is postponed and the environment is provided with the obligation to satisfy diamond modality. The second and third disjuncts represent the cases when the choices are made in favor of process P_1 and process P_2 respectively. Rule 14 is the dual of Rule 13.

Rules A and B define a function $\Pi : (P \times L \times F) \rightarrow E \rightarrow E$ which denote property transformers over mu-calculus *equations*. To transform a sequence of equations E, we construct the set of equations as per Rules A and B.

Figure 3 presents the transformation of the formula φ using the property transformer for process `C` (see Figure 1(a,b) for definition of the process and the formula respectively).

Theorem 1 *Given a well-named process expression P the following identity*

holds for all process expressions Q and for all mu-calculus formula φ

$$Q \mid P \models \varphi \;\Leftrightarrow\; Q \models \Pi^{\{\}}_{\perp}(P)(\varphi)$$

Proof: The proof proceeds by induction on the size of the process expression and formula. For details see Appendix A.

4 Verification of Parameterized Systems

Consider a parameterized system P_n defined by parallel composition of processes P. The parameter (n) represents the number of processes P present in the system. Consider verifying whether the ith instance of the above system possesses property φ: *i.e.* whether $P_i \models \varphi$. Let

$$\varphi_i = \Pi^L_f(P_i)(\varphi),$$

where f and L are the relabels and restrictions applied to the process P_i. Therefore, from Theorem 1, $0 \models \varphi_i \Leftrightarrow P_i \models \varphi$.

Now consider verifying whether $\forall i.\ P_i \models \varphi$. Let φ'_i be defined as follows

$$\varphi'_i = \begin{cases} \varphi_1 & \textit{if } i = 1 \\ \varphi'_{i-1} \wedge \varphi_i & \textit{if } i > 1 \end{cases} \tag{1}$$

By definition of φ'_i, $\forall 1 \leq j \leq i.0 \models \varphi_j \Leftrightarrow 0 \models \varphi'_i$. Hence, $0 \models \varphi'_i$ means that $\forall 1 \leq j \leq i.P_j \models \varphi$. If φ'_ω is the limit of sequence $\varphi'_1, \varphi'_2 \ldots$, then, $0 \models \varphi'_\omega \Leftrightarrow \forall i \geq 1.P_i \models \varphi$.

A dual method can be used to determine whether $\exists i \geq 1.\ P_i \models \varphi$ simply by defining

$$\varphi'_i = \begin{cases} \varphi_1 & \textit{if } i = 1 \\ \varphi'_{i-1} \vee \varphi_i & \textit{if } i > 1 \end{cases} \tag{2}$$

We say that φ'_i is said to be *contracting* (Equation 1) if $\varphi'_i \implies \varphi'_{i-1}$ and *relaxing* (Equation 2) if $\varphi'_{i-1} \implies \varphi'_i$. For systems indexed by a single parameter, the limit of the sequence of φ'_is can be computed by a fixed point iteration procedure.

Two problems need to be solved before this method can be implemented. First of all, we need a procedure to check if the limit φ_ω has been reached: that is to

determine the equivalence of two mu-calculus formulas. Checking equivalence between mu-calculus properties is EXPTIME-hard [20] and hence we need an efficient procedure to compute an approximate equivalence relation. Moreover, as remarked in [3] the formulas resulting from property transformers tend to be large and effective simplification procedures are needed before this method becomes practical. While we use the simplification rules from [3], we use a more powerful procedure to test for equivalence between mu-calculus formulas by constructing graphs from the formulas and checking for their similarity (Section 5).

The second problem arises due to the existence of infinite ascending chains in the domain of modal mu-calculus formulas: the iteration procedure may not always terminate. We describe a widening operator (based on definitions of widening operators over type domains) to guarantee the termination of iteration procedure at the expense of completeness in Section 6. In [40], similar idea has been applied on regular transition relations to ensure convergence of transitive closures of parameterized systems. The distinguishing feature of our work is that widening (acceleration) is tailored to property representation (mu-calculus) unlike the acceleration on transition relations [40].

The approach presented above can be easily applied to infinite families of systems specified by two or more parameters by considering a multi-parameter system as a nesting of single parameter systems. However, this is not possible if the parameters are interdependent; a method capable of verifying such infinite families remains to be developed.

5 Formula Graph and the Equivalence of Formulas

A formula graph, called *F-graph*, is an and/or graph that captures the structure of a mu-calculus formula, and is defined as follows:

Definition 1 *F-graph for a set of mu-calculus equations representing a formula φ is a tuple $F_\varphi = (S, \circ\!\!\longrightarrow, A)$ where*

- *S is the set of states such that $S \subseteq \mathcal{F} \times \mathcal{B} \times \Sigma$ where $\mathcal{F}$ is the set of all sub-formulas of φ, $\mathcal{B} = \{\#, \wedge, \vee\}$ and $\Sigma = \{\mu, \nu\}$;*
- *A is the set of labels such that $A \subseteq \mathcal{B} \times \mathcal{M} \times \Sigma$ where $\mathcal{M} = \mathcal{A}(\varphi) \cup \{\gamma\} \cup 2^{Prop}$, $Prop$ is the set of propositions in φ and $\mathcal{A}(\varphi) = \{\langle a \rangle \mid \langle a \rangle \varphi' \in \mathcal{F}\} \cup \{[a] \mid [a]\varphi' \in \mathcal{F}\}$; and*
- *'$\circ\!\!\longrightarrow$' is the set of transitions such that $\circ\!\!\longrightarrow \subseteq S \times A \times S$.*

Each state in formula graph is labeled by (i) mu-calculus formula (in $\mathcal{F}$), (ii) a boolean connective ($B \in \mathcal{B}$) stating whether the state is a part of "and" or "or"

Special Transition Rule for top variable X

$[X]^{\#,\sigma} \overset{\#,\gamma,\sigma}{\circ\!\!\longrightarrow} [\varphi]^{\#,\sigma}$ `if` $X =_\sigma \varphi$

General Transition Rules

1(a).	$[\varphi_1\ B\ \varphi_2]^{B',\sigma} \overset{B,m,\sigma}{\circ\!\!\longrightarrow} [\psi]^{B,\sigma}$	`if`	$[\varphi_1]^{B,\sigma} \overset{B,m,\sigma}{\circ\!\!\longrightarrow} [\psi]^{B,\sigma} \wedge (B = B' \vee B' = \#)$
1(b).	$[\varphi_1\ B\ \varphi_2]^{B',\sigma} \overset{B,m,\sigma}{\circ\!\!\longrightarrow} [\psi]^{B,\sigma}$	`if`	$[\varphi_2]^{B,\sigma} \overset{B,m,\sigma}{\circ\!\!\longrightarrow} [\psi]^{B,\sigma} \wedge (B = B' \vee B' = \#)$
2.	$[\varphi_1\ B\ \varphi_2]^{B',\sigma} \overset{B,\gamma,\sigma}{\circ\!\!\longrightarrow} [\varphi_1\ B\ \varphi_2]^{B,\sigma}$	`if`	$B' \neq B \wedge B' \neq \#$
3(a).	$[\langle a\rangle\varphi]^{B,\sigma} \overset{B,\langle a\rangle,\sigma}{\circ\!\!\longrightarrow} \varphi^{B,\sigma}$		
3(b).	$[[a]\varphi]^{B,\sigma} \overset{B,[a],\sigma}{\circ\!\!\longrightarrow} \varphi^{B,\sigma}$		
4.	$[p]^{B,\sigma} \overset{B,p,\sigma}{\circ\!\!\longrightarrow}$ `sink`	`if`	p `is proposition`
5.	$[Y]^{B,\sigma} \overset{B,\gamma,\sigma}{\circ\!\!\longrightarrow} \varphi^{B,\sigma_1}$	`if`	$Y =_{\sigma_1} \varphi$

Fig. 4. Transition relation for F-graph

structure (inherited attribute) and (iii) a fixed point operator ($\sigma \in \Sigma$) keeping track of fixed point nature of the current state's ancestor. Note that the top variable X (outermost formula variable), thus, has no inherited attributes. We use a special symbol # as its B label and synthesize the fixed point attribute from the definition of X. Rules 1 to 5 in Figure 4 complete the definition of transition relation for all other cases. Rules 1(a) and 1(b) are defined by transitive closure relation and capture action label $m \in \mathcal{M}$ present in identical boolean structures and under same fixed point operators. Note that the special symbol # can match with both $\wedge$ and $\vee$ boolean operators. Rule 2 presents the nesting of boolean structures. In this case, we use another special marker γ to identify toggling between boolean operators. γ is also used mark the first transition from a formula variable (Rule 5). Figure 5 shows a set of mu-calculus formula equations with top variable X and the corresponding F-graph.

F-graphs are labeled transition systems (LTSs) representing the syntactic structure of the corresponding formula equations. Let φ and ψ be mu-calculus formulas represented by equation sets with top formula variable X_φ and X_ψ respectively; the corresponding F-graphs are F_φ and F_ψ with start states $[X_\varphi]^{\#,\sigma_1}$ and $[X_\psi]^{\#,\sigma_2}$. We can establish that if the start states of F_φ simulates that of F_ψ and vice versa, then the corresponding formulas φ and ψ are equivalent.

Definition 2 (Simulation) *Given a labeled transition system L, simulation is the largest relation $\mathcal{R}$ such that for all states s_1 and s_2 in L,*

$$s_1 \mathcal{R} s_2 \;\Rightarrow\; \forall a, t_2.\ s_2 \xrightarrow{a} t_2 \Rightarrow \exists t_1.\ s_1 \xrightarrow{a} t_1 \wedge t_1 \mathcal{R} t_2.$$

Given LTSs L_1 and L_2 with start states s_1 and s_2, L_1 is said to simulate L_2, denoted by $L_1 \rhd L_2$, if $s_1 \mathcal{R} s_2$. If $L_1 \rhd L_2$ and $L_2 \rhd L_1$, we will denote the relation as $L_1 \lhd\!\rhd L_2$ [1].

[1] $L_1 \lhd\!\rhd L_2$ does not imply that L_1 and L_2 are bisimilar as bisimulation imposes a stronger requirement of two-way simulation on every bisimilar states [38].

Theorem 2 (Safe Equivalence) *Given the formula graphs F_φ and F_ψ for sets of mu-calculus equations representing formulas φ and ψ respectively, the following identity holds for all process expressions P*

$$F_\varphi \Leftrightarrow F_\psi \Rightarrow \; P \models \varphi \;\Leftrightarrow\; P \models \psi.$$

Proof: The proof proceeds by induction on the size of the formulas. See Appendix A for details. □

Discussion. In [3], a similar approach is proposed to detect the equivalence between mu-calculus formulas. Informally, in the setting of [3], two formulas (with similar boolean structure and fixed point nature) are said to be equivalent if for every modal action present in one formula there is an identical modal action present in the other and vice versa. Furthermore, the formula to be satisfied after matching modal actions must be also equivalent. For example the following formulas can be identified to be equivalent using techniques described in [3] and the technique proposed in this section.

$$X =_\nu [a]X \wedge Z \qquad Y =_\nu [a]Y \wedge [a]X \wedge Z$$

The highlight of our technique is to extract syntactic information of formulas by analyzing the corresponding graphical representations (unlike textual representation). This enhances the ability to effectively detect dependencies between formula variables. For example consider the formulas in Figure 5. Applying our technique, we can successfully identify the equivalence between X and Y and between X and Z. Note that, we consider $\overset{B,\langle\alpha\rangle,\sigma}{\circ\!\!\longrightarrow}$ can be matched by $\overset{B,\gamma,\sigma}{\circ\!\!\longrightarrow} * \overset{B,\langle\alpha\rangle,\sigma}{\circ\!\!\longrightarrow} \overset{B,\gamma,\sigma}{\circ\!\!\longrightarrow}*$; the reason being γ labelling may be caused due to a new fixed point variable (Rule 5 in Figure 4). On the other hand, [3] fails to detect any such equivalence because of the absence of modal actions $\langle b\rangle$ and $\langle a\rangle$ in the textual definitions of Y and Z respectively.

As remarked in [3], the transformation generates redundant formulas. For example, in Figure 3, the formulas defined by the fixed point variables $X_{\mathtt{c},\bot,\{\}}$ and $X_{\overline{a}.\mathtt{c},\bot,\{\}}$ are equivalent. The redundant formulas grow exponentially with the number of applications of the transformation, and hence redundancy removal is necessary to make our transformation-based technique usable in practice.

6 Accelerating Fixed Point Iterations

Widening [17] is a well-known technique for accelerating and guaranteeing termination over domains with infinite ascending chains. We first present an

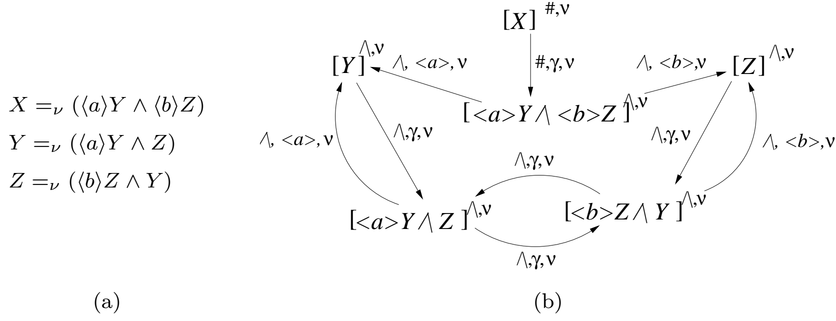

Fig. 5. (a) Mu-calculus formula equations and (b) the corresponding F-graph

acceleration operation, inspired by the widening operators defined over type graphs in the area of type analysis [26].

Consider the problem of computing the limit of the sequence $\psi_0, \psi_1, \ldots$ such that $\psi_{i+1} = f(\psi_i)$ and $\psi_{i+1} \geq \psi_i$. The acceleration operation, $\bigtriangledown$, is a monotonic function that views mu-calculus formulas as graphs. It determines a new formula $\psi' = \bigtriangledown(\psi_i, \psi_{i+1})$ based on the differences between ψ_i and ψ_{i+1} such that $\psi' \geq \psi_{i+1}$. Recall that equivalence of mu-calculus formulas are checked using similarities between their corresponding graphical representations (Section 5); these graphs are used in the definition of $\bigtriangledown$.

Acceleration based on Formula Graphs. The widening operator over type graphs [37, 26] identifies topological differences between two graphs and detects the state (in the graph to be widened) which leads to such a disparity between the two graphs. This node is termed as witness to *topological clash*. In the next step, an ancestor of the witness is selected with some specific property. Finally all the transitions from the witness is directed to the ancestor resulting in a loop. This removes the sub-graph of the witness and shortens the graph.

Following the same line, we develop an acceleration operator over mu-calculus formulas expressing safety and reachability properties as follows. We first formalize the notion of a topological clash between the formula graphs (F_{φ_1} and F_{φ_2}) of two formulas φ_1 and φ_2.

Definition 3 (Topological Clash) *Formula φ_2 clashes with φ_1 (denoted by $\varphi_2 \ominus \varphi_1$) if there exists a state N_2 in F_{φ_2} reachable by sequence of transitions (`seq`) from start state of F_{φ_2} such that for all states N_1 in F_{φ_1} reachable from the start state of F_{φ_1} by the same sequence `seq`, N_2 has an outgoing transition that cannot be matched by any outgoing transition from N_1s. State N_2 is said to be a witness to the clash.*

Intuitively, the above relation identifies the situation when φ_2 has an new subformula that is not present in φ_1. This type of divergence in the formula arises when a formula keeps *count* of modal actions needed to reach a particular state. We discard such counts as follows.

```
%N_1 & N_2 are start states of F_1 & F_2

procedure widen(N_1, N_2)
1: clash-set := null;
2: visited := null;
3: topoclash(N_1, N_2);
4: visited := null
5: foreach (N_c, M_c) ∈ clash-set do
6:    merge(N_c, M_c);
7: endforeach
8: return(N_2);
```

```
procedure topoclash(N_1, N_2)
 1: if (N_1, N_2) ∈ visited then
 2:   return;
 3: if ∃N_2 --b,m,σ--> M_2 ∧ ∄N_1 --b,m,σ--> M_1 then
 4:   clash-set := clash-set ∪ {(N_2, M_2)}
 5:   return;
 6: foreach N_2 --b,m,σ--> M_2 do
 7:   foreach N_1 --b,m,σ--> M_1 do
 8:     visited:=visited ∪ (N_1, N_2);
 9:     removeall (N_2, _) from clash-set;
10:     topoclash(M_1, M_2);
11:   endforeach
12: endforeach
```

Fig. 6. Widening algorithm

Consider the case, where the sequence of φ_i generated is contracting (Equation 1 in Section 4); Let N_2, an $\wedge$-node, be a witness to the topological clash $\varphi_2 \ominus \varphi_1$. From Definition 3, we state that N_2 has at least one outgoing transition to N_2' which cannot be matched by any transition from N_1s. We refer to such a transition as *clash-transition*. The witness is detected because of the introduction of new sub-formulas of the form $\langle a \rangle \psi$ (or $[a]\psi$) in φ_2. Acceleration is performed by merging the nodes N_2 and N_2', *i.e.* by merging the witness node with all the nodes reachable by clash-transitions. The F-graph obtained after merging represents a new formula φ_a. Such merging operation leads to shortening of the F-graph and in terms of abstraction of formula, a restricted φ_a ($\Rightarrow \varphi_2$) is generated (recall that sequence considered is contracting and N_2 is a $\wedge$-node). If the sequence of φ_i is relaxing, then the witness N_2 selected for discarding will be a $\vee$ node. Thus we ensure that the acceleration operator applied to φ_i generates its relaxed approximation. Approximation results in incompleteness of our technique. Recall from Theorem 1, $Q \models \varphi' \Leftrightarrow Q|P \models \varphi$ where $\varphi' = \Pi_{\perp}^{\{\}}(P)(\varphi)$. If φ_a is restricted or relaxed approximation of φ' then either $Q \models \varphi_a \Rightarrow Q|P \models \varphi$ or $Q|P \models \varphi \Rightarrow Q \models \varphi_a$ respectively holds true. In other words, if φ_a results from restriction (relaxation) and $Q \not\models \varphi_a$ ($Q \models \varphi_a$) then it cannot be inferred that $P|Q \not\models \varphi$ ($P|Q \models \varphi$).

Note however that the range of the acceleration operator is not a widening operator. The nodes selected for discarding are restricted by the definition of generated formulas (contraction or relaxation) and hence not all disparities between the formula graphs are even considered for pruning. For instance, sequence may be contracting but a formula can grow under an '$\vee$' node. This factor for divergence disappears when we restrict the mu-calculus formulas under consideration to those whose F-graphs have all $\wedge$-nodes or all $\vee$-nodes. Simple reachability and safety properties are of this form. This restriction on mu-calculus formulas makes the acceleration operation a widening operation.

Figure 6 presents the pseudo-code of the widening algorithm. Procedure `widen` is invoked with the start states N_1 and N_2 of the graphs F_1 and F_2; objective is to accelerate F_2 using its clashes/differences with F_1. In Line 3, procedure

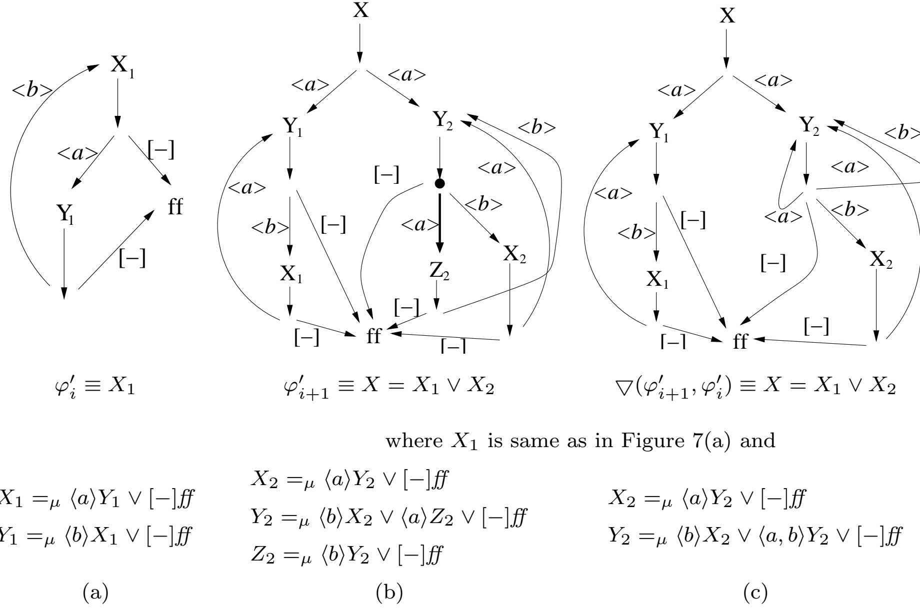

Fig. 7. Simplified F-graph in (a) i^{th} and (b) $i+1^{th}$ iteration, (c) accelerated F-graph with the corresponding formulas

`topoclash` is invoked which generates a set of witness states N_c paired with the corresponding states M_c such that there exists a clash-transition between them. Members of each such pair are then merged to realize required acceleration (Lines 5 to 7). Algorithmic complexity of this naive algorithm is exponential with respect to the number of nodes in the graphs, as it considers all possible transition sequences (paths) from the start states of each graph. For practical purposes, we restrict the search in procedure `topoclash` to detect witness nodes within certain pre-specified depth from the start states. A better algorithm in terms of complexity is still to be developed.

Let us consider a simple formula acceleration example and illustrate the acceleration mechanism. Consider the formula φ'_i in Figure 7(a). It defines a behavior where a deadlocked state (with no enabled transition) is reached after a and/or b actions. The box modality [-] corresponds to modality on any action denoted by "-". The formula graph is shown above the corresponding formula equations. We have simplified the graph for the sake of clarity and brevity by only labeling nodes with the fixed point variables. Furthermore, transition labels μ (fixed point sign) and $\vee$ (boolean connective) are omitted. Let φ'_{i+1} (Figure 7(b)) be the next formula that is generated as a result of transformation using the process $\mathtt{P} \stackrel{def}{=} b.a.\mathtt{P}$ (see definition of X_2 in Figure 7(b)) followed by disjunction with φ'_i. The witness to the topological clash $\varphi'_{i+1} \ominus \varphi'_i$ is the node shown with a $\bullet$ and the corresponding clash transition labeled by $\langle a \rangle$ leads to the node Z_2. Acceleration is performed by merging the witness node with Z_2 and the resultant accelerated formula obtained after simplification is shown in Figure 7(c). Note that acceleration led to merging of two $\vee$-structures which in turn led to relaxation of the formula. Further note

that acceleration discards all ordering of modal actions caused by the clash transition $\langle a \rangle$.

7 Case Studies

In this section, we discuss the applicability of our technique for automatic verification of mu-calculus properties for single-parameter systems.

Milner Scheduler. Milner's Scheduler [38] consists of a number of processes (called *cells*) connected in the form of a cycle where the i-th cell waits on synchronization with $(i-1)$-th cell and then communicates with the $(i+1)$-th cell. Each cell is also capable of performing autonomous actions. Figure 8(a) shows `N` cells in a ring topology (a simplified version of Milner's Scheduler where all the autonomous actions are discarded). Initially all cells except the first are waiting to synchronize on a $\overline{b}$ action from the previous cell in ring.

We consider the verification of the following mu-calculus property that encodes the existence of a deadlock:

$$\varphi_d : \quad X =_\mu \ [-]\mathit{ff} \vee \langle - \rangle X \tag{3}$$

Consider a system consisting of `N` cell processes, denoted by `sys(N)` (Figure 8(a)), and the problem of verifying $\exists$`Nsys(N)` $\models \varphi_d$, *i.e.* checking whether deadlock property is satisfied by a member of parameterized family `sys(N)`. The sequence of formulas as defined in Equation 2 (relaxing sequence in Section 4) does not converge. This is because φ'_i, the i-th formula in the sequence, captures all possible interleavings between actions of the first cell and the i-th cell. In fact, the interleavings that cause the divergence of the formulas in the sequence are result of *infeasible* interleaving of actions of the first and the i-th cell. More precisely, interleavings where $\overline{a}$ of the first cell appears before $\overline{a}$ of the i-th cell are represented in the generated formula. Clearly, such sequences are infeasible as the first cell can only make a move on $\overline{a}$ when the last cell is ready to make the synchronized move on a, which, in turn, is possible after all the actions of the intermediate cells (e.g. i-th cell). Equivalence reduction alone cannot discard such interleavings. However, when the widening operator (Section 6) is used, the resulting sequence converges after three cells (other than the first one) have been used to transform the formula; the acceleration operator ignores the exact nature of interleaving that causes of divergence. The fixed point after acceleration leaves for the environment the obligation to satisfy $\varphi_f \equiv X =_\mu \langle - \rangle \varphi$. As `0` has no outgoing transition, `0` $\not\models \varphi_f$. This implies $\forall$`N sys(N)` $\not\models \varphi_d$.

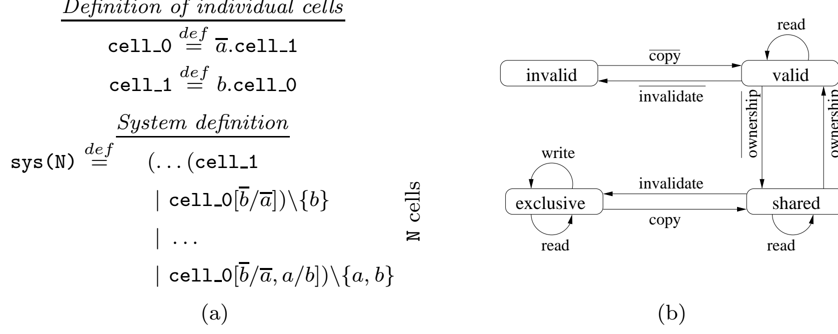

Fig. 8. (a) Simple ring structure. (b) Behavior of single processor in cache coherence protocol

Note here that similar behavior is exhibited by *token-ring* protocol and *queues* with two or more buffers. In all these cases, while the transformation sequence does not converge directly, the widening technique forces termination.

Cache Coherence Protocol. Cache coherence protocols [6] are used in multi-processor systems with shared memory, where each processor possesses its own private cache and maintains its own copy of same memory block in its private cache. The main concern is to ensure that at any point of time, multiple cached copies of same memory block are consistent in their data content. Cache coherence protocol defines four distinct states for each processor – invalid, valid, shared and exclusive. Invalid processor state implies that the processor's cached copy of memory block is outdated. Valid and shared states imply that processor has current copy of memory block in its cache, while exclusive state denotes that the processor is exclusive owner of the memory block. Each processor can either perform autonomous *read* (in all states except invalid) or *write* (only in exclusive state) actions, or can synchronize with another processor using *invalidate*, *copy* or *ownership* actions (Figure 8(b)).

To prove consistency of data, we need to ensure that each *read* action to a memory address reads the last value written to that location. Previous efforts [42, 18] to verify data consistency involved abstracting the parameterized system into a single infinite state system by counting the number of processors in various states. Model checking was performed by reachability analysis of the system; reachability of any global state, where the number of processors in each of valid, shared and exclusive states is greater than or equal to 2, implies violation of data consistency. In contrast we model the processors such that two of them in exclusive state can synchronize and lead to an "error" state – error state can perform autonomous action `err`. A least fixed point formula is used to detect an `err` action as follows:

$$\varphi_{\texttt{err}}: \quad X =_\mu \langle \texttt{err} \rangle tt \vee \langle - \rangle X \tag{4}$$

We model the components of parameterized system and check $\exists$N sys(N) $\models$

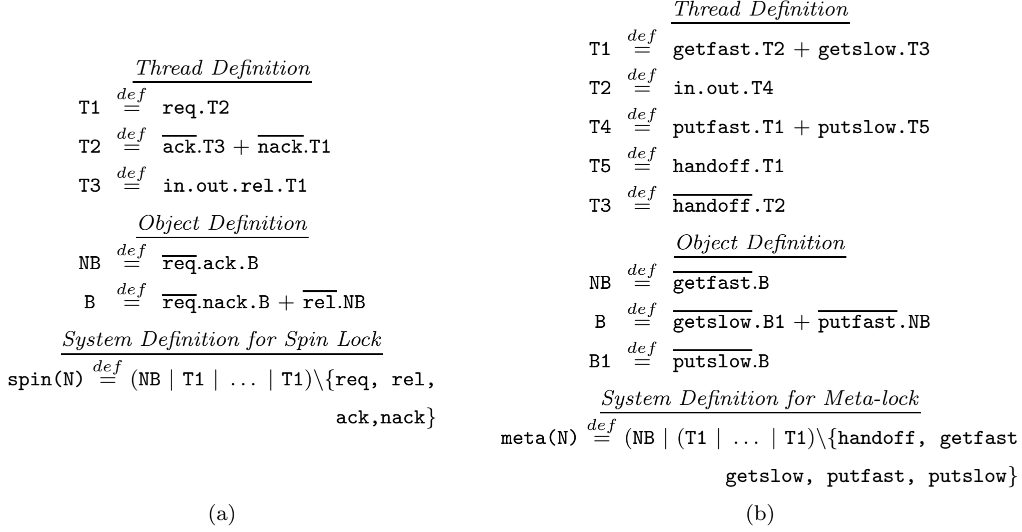

Fig. 9. Specification of (a) Spin Lock and (b) Meta-lock

$\varphi_{\texttt{err}}$ where `sys(N)` consists `N` processors. The limit of a relaxing sequence of formulas, generated by iterative transformation and disjunction, φ_f is obtained after three iterations, since at any point of time at most two processors can share the ownership of cached data. Finally $\texttt{0} \not\models \varphi_f$ implying data consistency is maintained for system consisting of any number of processors.

Shared-Memory Mutual Exclusion Protocols We consider two protocols that aim to ensure mutually exclusive access to shared memory in a multi-threaded system: (a) spin lock, where each thread can communicate only with the object it is trying to access and (b) a simplified version of Java meta-lock, where each thread either communicates with the object of interest or with another thread. We use a least fixed point formula φ_m to represent the failure of mutual exclusion property:

$$\begin{aligned} \varphi_m \; &\equiv Y =_\mu \langle \texttt{in} \rangle Z \;\vee\; \langle - \rangle Y \\ & \quad Z =_\mu \langle \texttt{in} \rangle tt \;\vee\; \langle -\texttt{out} \rangle Z \end{aligned} \tag{5}$$

where the modal actions `in` and `out` denote respectively the cases when a thread is accessing and has released the object. The modal operator $\langle -\texttt{out} \rangle$ represents any action other than `out`. Two `in` actions with no intermediate `out` indicate simultaneous access of the object by two threads – violation of mutual exclusion property.

Spin Lock. Spin locks offer a simple mechanism to realize mutually exclusive access of objects by threads (Figure 9(a)). The object has two states: *not-busy* (when it is not accessed by any thread, process `NB`) and *busy* (when

it accessed by some thread, process `B`). A not-busy object, upon receiving a `req` from a thread, replies back with an `ack` message and behaves like a busy object. A busy object, on the other hand, denies all requests from threads using `nack` message or goes to a not-busy state on receiving a `rel` (release) signal from the lock releasing thread. Each thread process can lock an object if it receives `ack` in response to a `req` signal. Once a thread has locked an object, it can perform autonomous actions `in` and `out` indicating it has acquired and is going to release object lock respectively. Using the system definition `spin(N)` consisting of one object and `N` thread processes, we verified the deadlock (φ_d in Equation 3) and non mutually exclusive access (φ_m in Equation 5) properties; the objective is to check whether $\exists$`N` `spin(N)` $\models \varphi_d$ and $\exists$`N` `spin(N)` $\models \varphi_m$.

In both cases, we transform the formulas using the common member of the system, i.e. the object process. The residue is subsequently iteratively transformed using the thread processes. At each iteration the residue is or-ed with the transformation result of the previous iteration leading to the generation of a relaxing sequence (Equation 2 in Section 4). The result is a diverging sequence of mu-calculus formulas. Widening is employed after two threads are used as transformers and iterative procedure is forced to terminate. The limit obtained by transformation and widening, φ_l, is used to check `0` $\models \varphi_l$. As `0` does not model the limits obtained in both the cases, (deadlock and no mutual exclusion), we infer $\forall$`N` `spin(N)` does not model φ_d and φ_m.

Simplified Java Meta-Lock. The Java Meta-Lock is a distributed algorithm designed by SUN Microsystems to ensure fast mutually exclusive access of objects by Java threads. Meta-locking can be viewed as a two-tiered scheme for exclusive access to object monitor. To ensure fairness and fast access each object maintains a *synchronization data*, which can be viewed as a FIFO queue of threads waiting to enter the object's monitor. Meta-lock is designed to provide exclusive access of per-object synchronization data by threads. A thread process can communicate with an object process or another thread processes via dedicated channels identified by the participating processes' identification numbers. Similar to the spin lock, the pattern of synchronization involves requests from threads and reply from objects. However in this case, a thread can directly communicate with another thread and exchange object lock. For details of the protocol refer to [1].

We consider here a simplified version [2] of meta-lock (Figure 9(b)). A thread

[2] The actual specification of meta-lock ([1, 9]) uses a queue to model list of waiting threads. Specifically, the process `B1` in Figure 9(b) can receive $\overline{\texttt{getslow}}$ requests from threads and record the number of such requests, subsequently requiring an infinite domain variable to keep track of the number of queued requests. In the current setting we avoid maintaining such queue.

can either obtain an object lock by a fast path (`getfast`) or via a slow path (`getslow`). In the former case, the object is not accessed by any other thread and current thread becomes the exclusive owner of the object lock. In the latter case, the thread waits to synchronize with the lock releasing thread via `handoff`. Release of object lock also follows similar pattern. A thread can release the lock following fast path when no other thread is waiting to access the object; otherwise, the thread follows slow path, where it relays the object lock to the waiting thread by synchronizing on `handoff`.

As in spin lock, we transform the given formulas (φ_d Equation 3 and φ_m Equation 5) using the object process followed by iterative transformations using the thread processes. The result of each iterative step is or-ed with the result in the previous iteration leading to the generation of relaxing sequence (similar to spin lock case). The sequences of mu-calculus formulas generated by iterative transformation of both φ_d and φ_m diverge and widening is used to force termination. Finally, `0` does not satisfy the accelerated formula obtained as limit of the sequences implying that the formulas φ_d and φ_m are not satisfied by any member of infinite family of systems defined using `meta(N)`.

Summary. Table 1 summarizes the results of verifying the examples described in this section. Observe from the table that the verification technique has been successfully applied to parameterized systems with different connection topologies and for different properties. Table 2 summarizes the performance of the technique on these examples. Space and time measurements reported in this table were taken using an implementation of the technique using XSB Prolog 2.5/Debian Linux 2.4.25 running on a 1.7GHz Xeon processor with 2GB memory. The third column in the table shows the number of compositional analysis iterations (of the common component, e.g. cells, processors or threads) required to reach the limit of the chain of mu-calculus formulas. The fourth column presents the maximum size of formula (before and after reduction) in terms of the number of fixed point equations used to denote the formula. Note that the technique converges within 3 iterations without the use of acceleration for the cache coherence protocol, while widening was used to force convergence for the scheduler, spin lock and meta-lock examples. The technique verifies most of the example systems within 10 seconds; the only exception being the mutual exclusion property of spin lock. Recall that in spin lock specification (Figure 9(a)), a thread is allowed to loop forever by sending and receiving `req` and `nack` respectively to and from the object. This phenomenon of starvation increases the number of possible interleaved behavior of threads in a spin lock. As such large formulas are generated at each iteration of compositional analysis which increases the reduction and equivalence checking time. Note that such starvation is not present in our specification of simplified meta-lock protocol.

System	Topology	Property	Result
Milner's Scheduler	Ring	Deadlock	false
Cache Coherence	Mesh	Data consistency	true
Spin Lock	Star	Mutual Exclusion	true
		Deadlock	false
Simplified Meta Lock	Star-wired Ring	Mutual Exclusion	true
		Deadlock	false

Table 1
Summary of results from the case studies.

System	Property	Iterations	Max. Formula Size		Time (sec)
			Raw	Reduced	
Milner's Scheduler	Deadlock	3	12	6	0.30
Cache Coherence	Data consistency	3	34	9	0.71
Spin Lock	Mutual Exclusion	4	329	56	384.24
	Deadlock	4	48	14	7.15
Simplified Meta Lock	Mutual Exclusion	4	56	10	5.52
	Deadlock	4	18	7	1.25

Table 2
Performance of compositional verification.

8 Conclusion

We described an automatic technique for the verification of infinite families of concurrent systems. At the core of the technique is the use of partial model checking for generating property transformers over modal mu-calculus formulas from system specifications in CCS. In our technique, the problem of verifying an infinite family is posed as a problem of finding the limit of a chain of modal mu-calculus formulas (similar to program analysis techniques). We also presented a widening operator to guarantee termination of the analysis for a subclass of modal mu-calculus formulas. We have implemented this technique in the XSB tabled logic programming system [45]. The utility of the technique has been demonstrated by verifying a number of example parameterized systems with diverse characteristics in a uniform manner. The widening technique, however, can be too approximate to provide useful results in certain cases. Development of widening operators which perform more fine-grained approximations is a topic of future research.

Acknowledgments. We would like to thank K. Narayan Kumar and Lenore Zuck for their detailed and insightful suggestions. We also thank the anonymous reviewers for their valuable comments.

This work is supported in part by NSF grants EIA-9705998, CCR-9876242, EIA-9805735, N000140110967, IIS-0072927, and CCF-0205376.

References

[1] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y.S. Ramakrishna, and D. White. An efficient meta-lock for ubiquitous synchronization. In *Proceedings of ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications*, 1999.

[2] R. Alur and T. Henzinger. Reactive modules. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 1996.

[3] H.R. Andersen. Partial model checking. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 1995.

[4] H.R. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal mu-calculus. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 1994.

[5] K.R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22:307–309, 1986.

[6] J. Archibald and J-L Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[7] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proceedings of Computer Aided Verification*, 2001.

[8] T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, 2001.

[9] S. Basu, S.A. Smolka, and O.R. Ward. Model checking the java meta-locking algorithm. In *Proceedings of Engineering of Computer-Based Systems*, 2000.

[10] S. Berezin and D. Gurov. A compositional proof system for the modal mu-calculus and CCS. Technical Report CMU-CS-97-105, CMU, 1997.

[11] J. Bradfield and C. Stirling. Modal logics and mu-calculi: an introduction. In *Handbook of Process Algebra*. Elsevier, 2001.

[12] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 1990.

[13] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.

[14] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Transactions on programming languages and systems*, 19(5), 1997.

[15] R. Cleaveland and B. Steffen. A linear-time model checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 1993.

[16] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[17] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of Principles of Programming Languages*, 1977.

[18] G. Delzanno. Automatic verification of parameterized cache coherence proto-

cols. In *Proceedings of Computer Aided Verification*, 2000.

[19] G. Delzanno and A. Podelski. Model checking in CLP. In *Proceedings of Tools and algorithms for Construction and Analysis of Systems*, 1999.

[20] E.A. Emerson and C.S. Jutla. The complexity of tree automata and logics of programs. In *Proceedings of Foundations of Computer Science*, 1988.

[21] E.A. Emerson and K.S. Namjoshi. Reasoning about rings. In *Proceedings of Principles of Programming Languages*, 1995.

[22] E.A. Emerson and K.S. Namjoshi. Automated verification of parameterized synchronous systems. In *Proceedings of Computer Aided Verification*, 1996.

[23] E.A. Emerson and K.S. Namjoshi. On model checking for non-deterministic infinite state systems. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 1998.

[24] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 1999.

[25] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[26] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, 1994.

[27] T. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee. In *Proceedings of Computer Aided Verification*, 1998.

[28] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proceedings of Computer Aided Verification*, 2003.

[29] G.J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[30] C.N. Ip and D.L. Dill. Better verification through symmetry reduction. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[31] C.N. Ip and D.L. Dill. Verifying systems with replicated components in murphi. *Formal Methods in System Design*, 14(3), 1999.

[32] Y. Kesten and A. Pnueli. Control and data abstraction:the cornerstones of practical formal verification. *International Journal on Software Tools for Technology Transfer*, 2(4):328–342, 2000.

[33] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[34] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *Proceedings of Principles of Programming Languages*, 1997.

[35] D.E. Long. *Model checking, abstraction and compositional verification.* PhD thesis, CMU, 1993.

[36] K.L. McMillan. Compositional rule for hardware design refinement. In *Proceedings of Computer Aided Verification*, 1997.

[37] P. Mildner. *Type Domains form Abstract interpretation: A critical study.* PhD thesis, Uppsala University, 1999.

[38] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[39] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems*, 2001.

[40] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *Proceedings of Computer Aided Verification*, 2000.
[41] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, 1982.
[42] A. Roychoudhury. *Program Transformations for Verifying Parameterized Systems*. PhD thesis, SUNY Stony Brook, 2000.
[43] A.P. Sistla and V. Gyuris. Parameterized verification of linear networks using automata as invariants. *Formal Aspects of Computing*, 11(4):402–425, 1999.
[44] P. Wolper. Expressing interesting properties in propositional temporal logic. In *Proceedings of Principles of Programming Languages*, 1986.
[45] XSB. The XSB logic programming system v2.6, 2003. Available from `http://xsb.sourceforge.net`.

A Appendix

Definition 4 (Process Ordering) *Process P_1 is said to be smaller than P_2, denoted by $P_1 \prec P_2$, if either of the following conditions hold: (a) P_1 is a sub-process expression of P_2, (b) for all relabeling f and restriction L, $P_1 = P[f]$ and $P_2 = P\backslash L$ and (c) $P_2(\equiv A) \stackrel{def}{=} P_1$.*

We only allow prefix-guarded process definitions, i.e., $A \stackrel{def}{=} A|P$ is not a permitted process definition.

Definition 5 (Formula Ordering) *Formula φ_1 is said to be smaller than formula φ_2, denoted by $\varphi_1 \prec \varphi_2$, if φ_1 is a sub-formula of φ_2.*

Theorem 1 *Given a well-named process expression P the following identity holds for all process expressions Q and for all mu-calculus formula φ*

$$Q \mid P \models \varphi \;\Leftrightarrow\; Q \models \Pi_{\perp}^{\{\}}(P)(\varphi)$$

Proof: The proof proceeds by induction on the size of the process expression and formula. Below we itemize the proof for the Rules 1–14 and A, B in Figure 2.

(1) Rules 1 & 2: The theorem is trivially true when φ is a propositional constant (tt or $f\!f$).

(2) Rule 3: $\varphi = \varphi_1 \vee \varphi_2$.

$$\begin{aligned} Q \mid P \models \varphi_1 \vee \varphi_2 &\Leftrightarrow Q \mid P \models \varphi_1 \vee Q \mid P \models \varphi_2 \\ &\Leftrightarrow Q \models \Pi_{\perp}^{\{\}}(P)(\varphi_1) \vee Q \models \Pi_{\perp}^{\{\}}(P)(\varphi_2) \\ &\quad\;\; \text{induction on formula size} \\ &\Leftrightarrow Q \models \Pi_{\perp}^{\{\}}(P)(\varphi_1 \vee \varphi_2) \end{aligned}$$

The proof for conjunctive formula (Rule 4) proceeds in identical fashion.

(3) Rule 6: Process expression $P = \mathtt{0}$. Considering the fact that process $\mathtt{0}$ is the identity of the parallel composition operator in CCS, we infer

$$Q \mid \mathtt{0} \models \varphi \;\Leftrightarrow\; Q \models \varphi \;\Leftrightarrow\; Q \models \Pi_{\perp}^{\{\}}(\mathtt{0})(\varphi)$$

(4) Rule 7: Process expression P is a process name A. By induction on the size of process expression ($D \prec A$ if $A \stackrel{def}{=} D$),

$$Q \mid A \models \varphi \;\Leftrightarrow\; Q \mid D \models \varphi \;\Leftrightarrow\; Q \models \Pi_{\perp}^{\{\}}(D)(\varphi)$$

(5) Rule 8: $P = R[f]$ where f is the relabeling function.

$$Q \mid R[f] \models \varphi \;\Leftrightarrow\; Q[f^{-1}] \mid R \models \varphi[f^{-1}]$$

Recall that, we consider well-named process expressions with injective relabiling functions. We define function f^{-1} as the inverse of f and $\varphi[f^{-1}]$ as the formula obtained from φ by replacing modal actions ($a \in n(\varphi)$) by $f^{-1}(a)$. By induction on process size ($R \prec R[f]$),

$$\begin{aligned} Q[f^{-1}] \models \Pi_{\perp}^{\{\}}(R)(\varphi[f^{-1}]) \;&\Leftrightarrow\; Q \models \Pi_{f}^{\{\}}(R)(\varphi) \\ &\Leftrightarrow\; Q \models \Pi_{\perp}^{\{\}}(R[f])(\varphi) \end{aligned}$$

(6) Rule 9: $P = R\backslash L$ where L is the set of restricted actions.

$$Q \mid (R\backslash L) \models \varphi \;\Leftrightarrow\; Q \mid R[L'/L] \models \varphi \;\Leftrightarrow\; Q \models \Pi_{\perp}^{\{\}}(R[L'/L])(\varphi)$$

where $L' \cap (L \cup vn(Q) \cup vn(R) \cup n(\varphi)) = \{\}$. Proceeding further, we cosider the relabeling $[L_r/L']$ where $L_r \cap (vn(R) \cup n(\varphi)) = \{\}$ (see Rule 9 in Figure 2). Proceeding further,

$$Q \models \Pi_{\perp}^{L_r}(R[L_r/L'] \circ [L'/L])(\varphi) \;\Leftrightarrow\; Q \models \Pi_{\perp}^{L_r}(R[L_r/L])(\varphi)$$

Note that $L_r \cap vn(Q)$ may not be empty according to the Rule 9. Process Q is not permitted to synchronize with $R[L_r/L]$ on any actions $\in L_r$.

(7) Rule 10: $P = (P_1 \mid P_2)\backslash L$. We consider the restricted set of actions L to show the anotations applies to transformation function Π in Rule 10.

$$\begin{aligned} Q \mid (P_1 \mid P_2)\backslash L \models \varphi &\Leftrightarrow Q \mid (P_1 \mid P_2)[L'/L] \models \varphi \\ &\Leftrightarrow Q \mid P_1[L'/L] \mid P_2[L'/L] \models \varphi, \end{aligned}$$

where $L' \cap (L \cup vn(Q) \cup vn(P_1|P_2) \cup n(\varphi)) = \{\}$ (see the proof for Rule 9). By induction on the size of the process expression

$$\begin{aligned} Q \mid P_1[L'/L] \mid P_2[L'/L] \models \varphi \;&\Leftrightarrow\; Q \mid P_1[L'/L] \models \Pi_{\perp}^{\{\}}(P_2[L'/L])(\varphi) \\ &\Leftrightarrow\; Q \models \Pi_{\perp}^{\{\}}(P_1[L'/L])(\Pi_{\perp}^{\{\}}(P_2[L'/L])(\varphi)) \\ &\Leftrightarrow\; Q \models \Pi_{\perp}^{L_p}(P_1[L_p/L])(\Pi_{\perp}^{\{\}}(P_2[L_p/L])(\varphi)) \end{aligned}$$

where $L_p \cap (vn(P_1|P_2) \cup n(\varphi)) = \{\}$. Note that the inner transformation function $\Pi(P_2)$ is not annotated with the restriction set L_p as P_1 is a present in its environment and P_2 can synchronize with P_1 on actions $\in L_p$.

(8) Rule 11: $P = a.R$ and $\varphi = \langle\alpha\rangle\psi$.

$$\begin{aligned} Q \mid a.R \models \langle\alpha\rangle\psi \Leftrightarrow\ & Q' \mid a.R \models \psi \ \ \textit{if}\ Q \xrightarrow{b} Q', b \in \alpha \\ \vee\ & Q \mid R \models \psi \ \ \textit{if}\ a \in \alpha \\ \vee\ & Q'' \mid R \models \psi \ \ \textit{if}\ \tau \in \alpha, Q \xrightarrow{\overline{a}} Q'' \end{aligned}$$

Next consider each disjunct separately and proceed by induction on the size of the formula ($\psi \prec \langle\alpha\rangle\psi$),

$$\begin{aligned} Q' \mid a.R \models \psi \ &\Leftrightarrow\ Q' \models \Pi_{\perp}^{\{\}}(a.R)(\psi) \\ &\Leftrightarrow\ Q \models \langle\alpha\rangle\Pi_{\perp}^{\{\}}(a.R)(\psi) \end{aligned} \tag{A.1}$$

Considering the second disjunct

$$Q \mid R \models \psi \ \Leftrightarrow\ Q \models \Pi_{\perp}^{\{\}}(R)(\psi) \tag{A.2}$$

and the third disjunct

$$Q'' \models R \models \psi \ \Leftrightarrow\ Q \models \langle\overline{a}\rangle\Pi_{\perp}^{\{\}}(R)(\psi) \tag{A.3}$$

Finally,

$$\begin{aligned} Q \models a.R \models \langle\alpha\rangle\psi \ \Leftrightarrow\ & Q \models \langle\alpha\rangle\Pi_{\perp}^{\{\}}(a.R)(\psi) \quad \text{Equation } A.1 \\ \vee\ \ & Q \models \Pi_{\perp}^{\{\}}(R)(\psi) \quad \text{Equation } A.2 \\ \vee\ \ & Q \models \langle\overline{a}\rangle\Pi_{\perp}^{\{\}}(R)(\psi) \quad \text{Equation } A.3 \end{aligned}$$

Similarly, we can prove for box modality formula (Rule 12).

(9) Rule 13: $P = P_1 + P_2$ and $\varphi = \langle\alpha\rangle\psi$.

$$\begin{aligned} Q \mid (P_1 + P_2) \models \langle\alpha\rangle\psi \ \Leftrightarrow\ & Q' \mid (P_1 + P_2) \models \psi \quad \textit{where}\ Q \xrightarrow{a} Q', a \in \alpha \\ \vee\ \ & Q \mid P_1 \models \langle\alpha\rangle\psi \\ \vee\ \ & Q \mid P_2 \models \langle\alpha\rangle\psi \end{aligned}$$

By induction on the size of the formula (first disjunct) and induction on the size of the process expressions (second and third disjuncts),

$$\begin{aligned} Q \mid (P_1 + P_2) \models \langle\alpha\rangle\psi \ \Leftrightarrow\ & Q \models \langle\alpha\rangle\Pi_{\perp}^{\{\}}(P_1 + P_2)(\psi) \\ \vee\ \ & Q \models \Pi_{\perp}^{\{\}}(P_1)(\langle\alpha\rangle\psi) \\ \vee\ \ & Q \models \Pi_{\perp}^{\{\}}(P_2)(\langle\alpha\rangle\psi) \end{aligned}$$

The theorem is proved in similar fashion for Rule 14.

(10) Rules 5 & A: $\varphi = X$ where $X =_\sigma \psi$. $Q \mid P \models X$ iff corresponding state represented by $Q \mid P$ is present in the interpretation of equation set E with top variable X. The equation set E is interpreted by the fixed point semantics of definitions of equations with appropriate initialization of environments following the sign of the equations. The result of transforming X using the transformation function Π for process P is $X_{P,\perp,\{\}}$ where $X_{P,\perp,\{\}} =_\sigma \Pi_\perp^{\{\}}(P)(\psi)$ (an equation with same sign as X, see Rule A in Figure 2). From the transformation function for basic formulas (leading to $Q \mid P \models \psi \;\Leftrightarrow\; Q \models \Pi_\perp^{\{\}}(P)(\psi)$), we conclude $Q \mid P \models X \;\Leftrightarrow\; Q \models \Pi_\perp^{\{\}}(P)(X)$. □

Theorem 2 (Safe Equivalence) *Given the formula graphs F_φ and F_ψ for sets of mu-calculus equations representing formulas φ and ψ respectively, the following identity holds for all process expressions P*

$$F_\varphi \triangleleft\!\triangleright F_\psi \Rightarrow \; P \models \varphi \;\Leftrightarrow\; P \models \psi.$$

Proof: The proof proceeds by induction on the size of the formulas. We consider below the transition rules presented in Figure 4.

(1) Rule 4: This rule corresponds to the case where φ and ψ are atomic propositions p and q respectively.

$$F_p \triangleleft\!\triangleright F_q \;\Rightarrow\; p = q \;\Rightarrow\; \forall P.(P \models p \Leftrightarrow\; P \models q).$$

(2) Rule 3a: $\varphi = \langle a \rangle \varphi'$ and $\psi = \langle b \rangle \psi'$. For diamond modality formulas the proof is as follows:

$$\begin{aligned} F_{\langle a \rangle \varphi'} \triangleleft\!\triangleright F_{\langle b \rangle \psi'} \;&\Rightarrow\; a = b \;\wedge\; F_{\varphi'} \triangleleft\!\triangleright F_{\psi'} \\ &\Rightarrow\; a = b \;\wedge\; \forall P.(P \models \varphi \;\Leftrightarrow\; P \models \psi) \\ &\qquad\qquad\qquad \text{Induction on formula size} \\ &\Rightarrow\; \forall P.(P \models \langle a \rangle \varphi' \;\Leftrightarrow\; P \models \langle b \rangle \psi') \end{aligned}$$

The proof for the box modality formula (Rule 3b) is realized in similar fashion.

(3) Rules 1a, b & 2: Let $\varphi = \bigvee \varphi_i$ and $\psi = \bigvee \psi_j$.

$$\begin{aligned} F_{\bigvee \varphi_i} \triangleleft\!\triangleright F_{\bigvee \psi_j} \;&\Rightarrow\; \forall j \exists i.(F_{\varphi_i} \triangleright F_{\psi_j}) \;\wedge\; \forall i \exists j.(F_{\psi_j} \triangleright F_{\varphi_i}) \\ &\Rightarrow\; \forall P \forall j \exists i.(P \models \varphi_i \Rightarrow P \models \psi_j) \;\wedge\; \forall i \exists j.(P \models \psi_j \Rightarrow P \models \varphi_i) \\ &\Rightarrow\; \forall P.(P \models \bigvee \varphi_i \;\Leftrightarrow\; P \models \bigvee \psi_j). \end{aligned}$$

Proof for conjunctive formulas is same as above.

(4) Special Rule for top variable & Rule 5: $\varphi = X$ & $\psi = Y$ where $X =_{\sigma_x} \varphi_x$ & $Y =_{\sigma_y} \psi_y$ respectively.

$$\begin{aligned} F_X \mathrel{\diamond\!\!|} F_Y &\Rightarrow \sigma_x = \sigma_y \ \wedge \ F_{\varphi_x} \mathrel{\diamond\!\!|} F_{\psi_y} \\ &\Rightarrow \sigma_x = \sigma_y \ \wedge \ \forall P.(P \models \varphi_x \ \Leftrightarrow \ P \models \psi_y) \\ &\Rightarrow \forall P.(P \models X \ \Leftrightarrow \ P \models Y). \end{aligned}$$

□