

Compositional Analysis for Verification of Parameterized Systems

Samik Basu and C. R. Ramakrishnan

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400
E-mail: {`bsamik`, `cram`}@`cs.sunysb.edu`

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 μ -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 μ -calculus. We describe the verification of a number of parameterized systems using our technique to demonstrate its utility.

1 Introduction

Model checking is a widely used approach for verifying whether a system specification possesses a property expressed in temporal logic [10, 34]. Many efficient verification tools have been developed based on approaches such as explicit-state [24], symbolic [9] 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. [15]), symmetry reduction [25], data independence [38], and symbolic checking with rich assertional languages [27] have extended model checking techniques to certain classes of infinite-state systems.

The Driving Problem One class of infinite-state systems called *parameterized systems* is particularly interesting. 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 producer process P performs an action a and continues to behave as P . Similarly, the consumer process C repeatedly

$$\begin{array}{ll}
\mathbf{P} \stackrel{def}{=} \mathbf{a}.\mathbf{P} & \varphi \equiv X \text{ where } X =_{\nu} \langle \tau \rangle \mathbf{tt} \wedge [\tau]X \\
\mathbf{C} \stackrel{def}{=} \bar{\mathbf{a}}.\mathbf{C} & \varphi_c \equiv Y \text{ where } Y =_{\nu} \langle \tau, \mathbf{a} \rangle \mathbf{tt} \wedge [\tau, \mathbf{a}]Y \\
\mathbf{sys}(N) \stackrel{def}{=} (\mathbf{P}^N | \mathbf{C}) \setminus \{\mathbf{a}\} & \varphi_1 \equiv Z_1 \text{ where } Z_1 =_{\nu} [\tau, \mathbf{a}, \bar{\mathbf{a}}]Z_1 \\
& \varphi_2 \equiv Z_2 \text{ where } Z_2 =_{\nu} [\tau, \mathbf{a}, \bar{\mathbf{a}}]Z_2
\end{array}
\tag{a} \qquad \tag{b}$$

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

performs action $\bar{\mathbf{a}}$. The processes communicate by synchronization on $\bar{\mathbf{a}}$ and \mathbf{a} actions. The parameterized system $\mathbf{Sys}(M, N)$ is specified as parallel composition of M producers and N consumers. Our objective is to verify deadlock-freedom property for *all instances* of the system \mathbf{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 [6]. 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 [32]). 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 [28]). 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 \mathbf{P} in Figure 1(a). The process can move on \mathbf{a} transition only if there is a concurrent process ready to move on $\bar{\mathbf{a}}$ transition. In order for the process \mathbf{P} to execute the \mathbf{a} action, the environment must be capable of synchronizing with an $\bar{\mathbf{a}}$ action. Thus the process \mathbf{P} can be seen as transforming the property (“eventually do a transition”) to its environment (“eventually do $\bar{\mathbf{a}}$ transition”).

The property transformer for a given process is generated based on 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 $\mathbf{sys}(n)$ for all $n \geq 1$. The formula to be checked for the entire system is given in Figure 1(b) as φ .

Consider the system $\mathbf{sys}(n)$ with one consumer (\mathbf{C}) and n producer processes (\mathbf{P}^n). We compute the property expected of the producers alone, by transforming the property φ using the property transformer for \mathbf{C} process. The re-

sulting “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 \mathbf{C} if the environment can perform infinitely many \mathbf{a} or τ actions. Therefore, if $\mathbf{P}^n \models \varphi_c$ then $\mathbf{sys}(n) \models \varphi$. Next, transform φ_c using the property transformer for process \mathbf{P} . The resultant property left for the environment (\mathbf{P}^{n-1}) to satisfy is φ_1 . Quotienting further using process \mathbf{P} , the residue obtained is φ_2 . Further transformation of φ_2 using the property transformer for \mathbf{P} will leave it unaltered. Thus, we have reached the limit φ_ω of formula sequence generated by iterative transformation using property transformer of process \mathbf{P} . Then $\mathbf{0} \models \varphi_\omega$ implies $\forall n \in \mathbb{N} \mathbf{sys}(n) \models \varphi$. The above discussion presents a high level view of the technique used to verify properties for all or any members of a parameterized system. Actual technique, however, keeps track of various restriction and relabeling operations applied to the processes. See Section 3,4 for details.

Note that the domain of interpretation, the modal mu-calculus, has infinite ascending chains, and hence the limit computation may not terminate. Nevertheless, we find that the iterative computation of the limit does converge for a number of example parameterized systems. To handle a larger class of systems, we also define a widening operation to accelerate the convergence, and in some cases guarantee termination.

Related Work A number of techniques have been proposed to verify parameterized systems with varying amounts of user intervention ranging from fully automatic techniques (such as [27]) which focus on the domain of representations of system states, to program-transformation-based systems capable of inferring the structure of certain underlying induction proofs [35, 36].

One of the approaches is to reduce the infinite-state verification problem to an equivalent finite-state one, by identifying a representative finite-state system corresponding to a given parameterized system and temporal property (e.g. see [17, 18, 26]). Cache coherence protocols and unidirectional token ring protocols have been successfully verified using this approach. Recently there have been efforts to verify infinite families by choosing an appropriate finite representation (e.g. using regular languages or counting the number of components in particular states, see [19, 20, 14, 33]). All these approaches require a specialized way of specifying processes: as grammars [11], logic programs [35], or rewrite rules [27]. In contrast, our technique directly manipulates parameterized process specifications in CCS. Moreover, being based on program analysis, our technique can be applied with little or no knowledge of the internals of the system under consideration. This is in contrast to representation-based techniques [19, 20, 14, 27] whose success depends on a clever choice of representations. Another approach, which requires considerable user intervention, is to generate a network invariant for a system consisting of arbitrary number of identical components [11, 29]. For chains and circular networks [37] presents a method to generate such invariants automatically using a fixed point iteration procedure over two-dimensional strings automata. Our technique is also based on computing the limit of an infinite chain, but one of mu-calculus formulas, and does not restrict the network topology of the system to be verified.

An important aspect of our work is the generation of property transformers using techniques from compositional model checking. Considerable amount of research has been done on using assume-guarantee reasoning for constructing compositional proofs [21, 2, 30, 7, 23]. However, these methods typically need considerable user guidance. 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 to arbitrary CCS processes. We also present a bisimulation-based procedure to reduce the size of formulas generated by property transformers that results in smaller formulas than the method used in [3].

Contributions We present a technique for automatic verification of parameterized systems, representing an infinite family of finite-state systems. The technique views processes as property transformers and is based on computing the limit of a sequence of mu-calculus formula generated by these transformers.

1. We develop a compositional model checker for CCS [32] and use this model checker to generate property transformers (Section 3).
2. We use the property transformers to define a sequence of mu-calculus formula. The limit of this sequence is used to verify properties over infinite families of systems. (Section 4).
3. To guarantee convergence of iterative procedure, we define acceleration and widening operators (based on widening techniques used in type analysis) for mu-calculus formula. (Section 4.1).
4. 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 metalock [1]), and cache coherence protocols [14] (Section 5). Details of the examples are available at <http://www.cs.sunysb.edu/~lmc/compose>.

2 Preliminaries

We briefly outline the syntax of the process algebra CCS [32] and the logic modal mu-calculus [8] used in the rest of the paper.

2.1 CCS and labeled transition systems

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

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

In the above, 0 denotes a deadlocked process. A ranges over process names (agents) and a ranges over a set of actions $Act = \mathcal{L} \cup \bar{\mathcal{L}} \cup \tau$, where τ represents an internal action and \mathcal{L} is a set of labels and $\bar{\mathcal{L}}$ is such that $a \in \mathcal{L} \Leftrightarrow \bar{a} \in \bar{\mathcal{L}}$. Finally, L ranges over the powerset of \mathcal{L} , and $f : \mathcal{L} \rightarrow \mathcal{L}$. The operators ‘ \cdot ’,

‘+’, ‘|’, ‘\’ and ‘[.]’ are called prefix, choice, parallel, 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, \rightarrow) is specified by a set of *states* S and a transition relation $\rightarrow \subseteq S \times Act \times S$. The operational semantics of CCS expressions is given in terms of labeled transition systems where states represent CCS expressions. See [32] for a full definition of the semantics of CCS.

2.2 The modal mu-calculus

The modal mu-calculus [28] is an expressive temporal logic with explicit greatest and least fixed point operators. Following [12, 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 : $\Phi \rightarrow tt \mid ff \mid X \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \langle \alpha \rangle \Phi \mid [\alpha] \Phi$.

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. The variables used in a mu-calculus formula are *defined* using a *sequence* of simultaneous equations where the i th 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 F . The set of all mu-calculus equations is denoted by \mathcal{E} .

Model Checking. Given a labeled transition system (S, \rightarrow) , the semantics of mu-calculus formulas are stated such that each formula denotes a subset of S . Refer to [8] 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 (denoted by $s \models \varphi$).

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 $P|Q \models \varphi$ whenever $Q \models \varphi'$. 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 a LTSs.

In Figure 2 we define the property transformer using a function $\Pi : (\mathcal{P} \times \mathcal{L} \times \mathcal{F}) \rightarrow \Phi \rightarrow \Phi$ where \mathcal{L} is 2^{Act} and \mathcal{F} is a set of partial functions $f : Act \rightarrow Act$ such that $f(x) \neq x$. We use \perp to denote empty relabeling function which is undefined everywhere. We define composition of two relabeling functions $h = f \circ g$ such that $h(x)$ is undefined if $f(x)$ and $g(x)$ are undefined, $h(x) = f(x)$ if $g(x)$ is undefined, $h(x) = g(x)$ if $f(x)$ is undefined; if both are defined, then $h(x) = f(g(x))$. Φ is the set of modal mu-calculus formulas. Finally \mathcal{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 set of visible actions of process Q is disjoint from the range of function f .

The transformer $\Pi_f^L(P)$ 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 any actions in L . The transformer $\Pi_f^L(P)$ 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 , denoted by $n(\varphi)$, are the set of actions in φ and the names of all the formula variables appearing in φ . The names of formula variable X are the names of formula φ where $X =_\sigma \varphi$. Range of relabeling f is the set of actions v such that $f : x \rightarrow v$. The function $f' = f \setminus L$ is such that $f'(x) = f(x)$ if $f(x) \notin L$ and $f'(x)$ is undefined otherwise.

Rules 1 through 5 in Figure 2 define the property transformer for propositional constants, boolean connectives, formula variables. Rule 6 states that the property transformer for the zero (deadlocked) process, 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 the existing relabeling function with f_p (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. 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 restriction and relabels. First, consider process P_1 in Rule 10. The transformer function for P_1 is restricted on actions in L_1 , which are not visible to the environment of P_1 , *i.e.* P_2 . Therefore, transformer function for P_2 is restricted on actions in L_2 ($= L - L_1$). Further, note that, relabel mapping on process P_2 is transformed by projecting off the mappings concerning names in L_1 .

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

1. $\Pi_f^L(P)(tt) = tt$
2. $\Pi_f^L(P)(\text{ff}) = \text{ff}$
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. $\Pi_f^L(A)(\varphi) = \Pi_f^L(P)(\varphi) \quad \text{if } A \stackrel{\text{def}}{=} P \in D$
8. $\Pi_f^L(P[f_p])(\varphi) = \Pi_{f \circ f_p}^L(P)(\varphi)$
9. $\Pi_f^L(P \setminus L_p)(\varphi) = \Pi_{f \cup L'}^L(P[L'/L_p])(\varphi)$
where $L' \cap (n(\varphi) \cup \text{vn}(P) \cup \text{range}(f) \cup L) = \{\}$
10. $\Pi_f^L(P_1 | P_2)(\varphi) = \Pi_{f_2}^{L_2}(P_2)(\Pi_{f_1}^{L_1}(P_1)(\varphi))$
where $L_1 = L - \text{vn}(P_2)$, $L_2 = L - L_1$, $f_2 = f \setminus L_1$
11. $\Pi_f^L(a.P)(\langle \alpha \rangle \varphi) = \langle \alpha \rangle \Pi_f^L(a.P)(\varphi) \vee \left\{ \begin{array}{l} \Pi_f^L(P)(\varphi) \text{ if } f(a) \in \alpha \\ \text{ff} \quad \text{otherwise} \end{array} \right\}$
 $\vee \left\{ \begin{array}{l} \langle \overline{f(a)} \rangle \Pi_f^L(P)(\varphi) \text{ if } \tau \in \alpha \wedge \overline{f(a)} \notin L \\ \text{ff} \quad \text{otherwise} \end{array} \right\}$
12. $\Pi_f^L(a.P)([\alpha] \varphi) = [\alpha] \Pi_f^L(a.P)(\varphi) \wedge \left\{ \begin{array}{l} \Pi_f^L(P)(\varphi) \text{ if } f(a) \in \alpha \\ tt \quad \text{otherwise} \end{array} \right\}$
 $\wedge \left\{ \begin{array}{l} [\overline{f(a)}] \Pi_f^L(P)(\varphi) \text{ if } \tau \in \alpha \wedge \overline{f(a)} \notin L \\ tt \quad \text{otherwise} \end{array} \right\}$
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. $\Pi_f^L(P)(X =_{\sigma} \varphi \cup E) = X_{P,f,L} =_{\sigma} \Pi_f^L(P)(\varphi) \cup \Pi_f^L(P)(E) \cup$
 $\{\bigcup (\Pi_{P'}^{L'}(P')(X' =_{\sigma'} \varphi') \text{ s.t. } X'_{P',F',L'} \text{ is subformula of } \Pi_f^L(P)(\varphi), X' =_{\sigma'} \varphi' \in F)\}$
- B. $\Pi_f^L(P)(\{\}) = \{\}$

Fig. 2. Partial Model Checker for CCS

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 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 \bar{a} action by Q to produce the necessary τ action. This means that the obligation on Q is to first produce an \bar{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 $f(a)$ 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 disjunct represents 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 $\Pi : (\mathcal{P} \times \mathcal{L} \times \mathcal{F}) \rightarrow \mathcal{E} \rightarrow \mathcal{E}$ which defines 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.

The correctness of the quotienting operation, formally stated below, can be proved by induction on the structure of formula and process expressions.

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

$$\forall Q \quad Q|P \models \varphi \Leftrightarrow Q \models \Pi_{\perp}^{\{\}}(P)(\varphi)$$

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 i^{th} instance of the above system possesses property φ : *i.e.* whether $P_i \models \varphi$. Let

$$\varphi_i = \Pi_f^L(P_i)(\varphi),$$

where f and L are the relabelings 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

$$\varphi'_i = \begin{cases} \varphi_1 & \text{if } i = 1 \\ \varphi'_{i-1} \wedge \varphi_i & \text{if } i > 1 \end{cases} \quad (1)$$

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 $\varphi'_1, \varphi'_2, \dots$, then, $0 \models \varphi'_\omega \Leftrightarrow \forall i \geq 1. P_i \models \varphi$.

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

$$\varphi'_i = \begin{cases} \varphi_1 & \text{if } i = 1 \\ \varphi'_{i-1} \vee \varphi_i & \text{if } i > 1 \end{cases} \quad (2)$$

We say that φ'_i is said to be *contracting* if $\varphi'_i \Rightarrow \varphi'_{i-1}$ and *relaxing* if $\varphi'_{i-1} \Rightarrow \varphi'_i$. For systems indexed by a single parameter, the limit of the sequence of φ'_i s can be computed by a fixed point iteration procedure. For details of the proof refer to <http://www.cs.sunysb.edu/~lmc/compose>.

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 [16] 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 bisimilarity.

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 4.1. In [33], 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 [33].

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. This cannot be done if the parameters are interdependent; a method capable of handling such infinite families remains to be developed.

4.1 Accelerating Fixed Point Iterations

Widening [13] is a well-known technique for accelerating and guaranteeing termination over domains with infinite ascending chains. We first present an acceleration operation, inspired by the widening operators defined over type graphs in the area of type analysis [22], to accelerate the convergence, but this still does not guarantee termination. This operation can be modified to yield a widening operator for a class of mu-calculus formulas.

Consider the problem of computing the limit of the sequence ψ_0, ψ_1, \dots such that $\psi_{i+1} = f(\psi_i)$ and $\psi_{i+1} \geq \psi_i$. The acceleration operation, *accel*, is a monotonic function that views mu-calculus formulas as graphs. It determines a new formula $\psi' = \text{accel}(\psi_i, \psi_{i+1})$ based on the differences between ψ_i and ψ_{i+1} such that $\psi' \geq \psi_{i+1}$. The acceleration operation is defined by considering a graph representation of mu-calculus formulas as described below.

Formula Graph 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* is defined as a tuple $FG = (S, \circ \rightarrow, A)$, where S is the set of states labeled by a pair (α, σ) , $A \subseteq \alpha \times \beta \times \sigma$ is the set of labels on transitions, where $\alpha \in \{\#, \vee, \wedge\}$, $\beta \in \{[a], \langle a \rangle, \gamma\}$ and $\sigma \in \{\mu, \nu\}$. $\circ \rightarrow \subseteq S \times A \times S$ is the labeled transition relation between pairs of states. The transition relation $\circ \rightarrow$ is a least relation as defined in Figure 3.

Special Transition Rule for top variable X

$$[X]^{\#, \sigma} \xrightarrow{\#, \gamma, \sigma} [\phi]^{\#, \sigma} \text{ if } X =_{\sigma} \phi$$

General Transition Rules

1(a).	$[\varphi_1 \ b \ \varphi_2]^{b', \sigma} \xrightarrow{b, m, \sigma} [\psi]^{b, \sigma}$	if $[\varphi_1]^{b, \sigma} \xrightarrow{b, m, \sigma} [\psi]^{b, \sigma} \wedge (b = b' \vee b' = \#)$
1(b).	$[\varphi_1 \ b \ \varphi_2]^{b', \sigma} \xrightarrow{b, m, \sigma} [\psi]^{b, \sigma}$	if $[\varphi_2]^{b, \sigma} \xrightarrow{b, m, \sigma} [\psi]^{b, \sigma} \wedge (b = b' \vee b' = \#)$
2.	$[\varphi_1 \ b \ \varphi_2]^{b', \sigma} \xrightarrow{b, \gamma, \sigma} [\varphi_1 \ b \ \varphi_2]^{b, \sigma}$	if $b' \neq b \wedge b' \neq \#$
3(a).	$[(a)\varphi]^{b, \sigma} \xrightarrow{b, \langle a \rangle, \sigma} \varphi^{b, \sigma}$	
3(b).	$[[a]\varphi]^{b, \sigma} \xrightarrow{b, [a], \sigma} \varphi^{b, \sigma}$	
4.	$[Y]^{b, \sigma} \xrightarrow{b, \gamma, \sigma} \varphi^{b, \sigma_1}$	if $Y =_{\sigma_1} \varphi$

Fig. 3. Transition relation for F-graph

Each state in formula graph is labeled by (i) a boolean connective (b) stating whether the state is a part of “and” or “or” structure and (ii) a fixed point operator (σ) keeping track of fixed point nature of the current state’s ancestor. Note that the top variable X , 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 4 complete the definition of transition relation for all other cases. Rules 1(a) and 1(b) are defined by transitive closure relation and captures action label 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 to mark the first transition from a formula variable.

Note that F-graphs capture only some of the structure of a mu-calculus formula: for instance, the order of conjuncts in a disjunction is omitted. F-graphs can be viewed as labeled transition systems. This permits us to check for equivalence between two mu-calculus formulas based on the bisimulation [32] of their respective F-graphs.

Proposition 1 *Two mu-calculus formula φ_1 and φ_2 are equivalent if their corresponding F-graphs F_1 and F_2 are bisimilar.*

Acceleration based on F-graphs The widening operator over type graphs [31, 22] 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.

```

procedure widen( $F_{\varphi_1}, F_{\varphi_2}$ )
1. clash-set := null;
2. visited := null;
3. topoclash( $N_1, N_2$ );
   % $N_1$  &  $N_2$  are start nodes of  $F_{\varphi_1}$  &  $F_{\varphi_2}$ 
4. visited := null
5. foreach  $N_c \in$  clash-set do
6.    $N_a :=$  anc-of( $N_c, F_{\varphi_2}$ );
7.   rearrange( $N_a, N_c$ );
8. endforeach
9. return ( $F_{\varphi_2}$ );

procedure anc-of( $N_c, F_{\varphi_2}$ )
1. foreach  $N_a \in F_{\varphi_2}$  do
2.   if  $N_a \xrightarrow{*} N_c \wedge \text{sim}(N_a, N_c) \wedge$ 
      $N_a \& N_c$  are both  $\wedge$  or  $\vee$  nodes
3.   then return ( $N_a$ );
4. endforeach
5. return (null);

procedure topoclash( $N_1, N_2$ )
1. if ( $N_1, N_2$ )  $\in$  visited then
2.   return;
3. if  $\exists N_2 \xrightarrow{b, m, \sigma} M_2 \wedge \neg \exists N_1 \xrightarrow{b, m, \sigma} M_1$  then
4.   clash-set := clash-set  $\cup$   $\{N_2\}$ 
5. return;
6. foreach  $N_2 \xrightarrow{b, m, \sigma} M_2$  do
7.   foreach  $N_1 \xrightarrow{b, m, \sigma} M_1$  do
8.     visited := visited  $\cup$  ( $N_1, N_2$ );
9.     remove  $N_2$  from clash-set;
10.    topoclash( $M_1, M_2$ );
     endforeach
11. endforeach

procedure sim( $N_a, N_c$ )
1. if ( $N_a, N_c$ )  $\in$  visited then
2.   return 1;
3. ret-val := 1;
4. foreach  $N_c \xrightarrow{b, m, \sigma} M_c$  do
5.   ret-val1 := 0;
6.   foreach  $N_a \xrightarrow{b, \gamma, \sigma} * \xrightarrow{b, m, \sigma} \xrightarrow{b, \gamma, \sigma} * M_a$  do
7.     visited := visited  $\cup$  ( $N_a, N_c$ );
8.     ret-val1 := ret-val1  $\vee$  sim( $M_a, M_c$ );
9.   endforeach
10.  ret-val := ret-val & ret-val1;
11. endforeach
12. return (ret-val);

```

Fig. 4. Widening Algorithm

Following the same line, we develop an acceleration operator over mu-calculus formulas expressing safety and reachability properties as follows. Let F_φ be the formula graph corresponding to the formula φ . We first formalize the notion of a topological clash between the formula graphs of two formulas φ_1 and φ_2 .

Definition 2 *Formula φ_2 clashes with φ_1 (denoted by $\varphi_1 \ominus \varphi_2$) if there exists states N_1 in F_{φ_1} and N_2 in F_{φ_2} , such that the states N_1 and N_2 are reachable from the start states of F_{φ_1} and F_{φ_2} by identical sequences of transitions and there exists a transition from N_2 that has no matching transition from N_1 . This is called topological clash and N_2 is said to be a witness to the clash.*

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

Consider the case, where the sequence of φ_i generated is contracting (Equation 1); Let $\varphi_1 \ominus \varphi_2$ and N_2 be an ' \wedge ' node that is a witness to the topological clash. We identify an ancestor N_a of N_2 such that N_a *simulates* N_2 : i.e. every

action possible from N_2 is also possible from N_a and the corresponding successors of N_a simulate the successors of N_2 (See [32] for a definition of simulation relation over labeled transition systems). We construct the accelerated graph $F_{\varphi'}$ from F_{φ_2} by removing state N_2 and redirecting all incoming transitions to N_2 to N_a and introducing all outgoing transitions of N_2 to N_a . Figure 4 presents the pseudo-code for widening operation. It takes two formula graphs F_{φ_1} and F_{φ_2} and performs acceleration on F_{φ_2} . Procedure `rearrange` (Line 7 of procedure `widen`) removes the node N_a and redirects its incoming and outgoing edges to and from N_a respectively.

Note that such widening shortens the formula graph by merging one or more witness nodes with their respective ancestors. In terms of abstraction of formula, such merging amounts to discarding the exact sequence of modal actions that is preserved in un-abstracted formula.

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. Note however that the range of the acceleration operator is not a widening operator and its range contains infinite ascending chains. Two factors prevent it from being a widening operator. First, the nodes selected for discarding are restricted by the definition of generated formula (contraction or relaxation) and hence not all growth in 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 and-nodes or all or-nodes. Simple reachability and safety properties are of this form. Secondly, the selected witness N_2 may not have an ancestor N_a such that N_a simulates N_2 . Under this circumstance, we can simply replace N_2 with tt if N_2 is a ' \vee ' node and ff if N_2 is a ' \wedge ' node. This approximation, combined with the restriction of mu-calculus formulas proposed above, makes the acceleration operation a widening operation. The approximation, however appears to be very coarse and results in considerable information loss (see Section 5).

5 Case Studies

In this section, we discuss the applicability of our technique for automatic verification of mu-calculus properties for single-parameter systems. The examples show that our technique can be used to verify parameterized systems with different control structures like ring, chain and star networks.

Milner Scheduler Milner's Scheduler [32] consists of `cell` processes 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. Further each `cell` is also capable of performing autonomous actions. Initially all cells except `cell(0)` are waiting to synchronize on an `out` action from the previous cell in ring.

We consider the verification of the following mu-calculus property that encodes the existence of a deadlock : $\varphi_d : X =_{\mu} [\tau]ff \vee \langle \tau \rangle X$. Consider a system

consisting of $N+1$ `cell` processes, denoted by $\mathbf{sys}(N)$, and the problem of verifying $\exists N \mathbf{sys}(N) \models \varphi_d$. The sequence of formula as defined in Equation 2 does not converge. This is because ψ_i , the i^{th} formula in the chain explicitly represents all possible interleavings between actions of the `cell(i)` and the `cell(0)`. Equivalence reduction alone cannot discard such interleavings. When the acceleration operator (Section 4.1) is used, the resulting chain converges; the acceleration operator ignores the exact nature of interleaving between the actions. The limit after acceleration, φ_f , leaves for the environment the obligation to satisfy φ_d after an `out` action of the `cell(N)` or an `in` action of `cell(0)`. As `0` has no outgoing transition, $0 \not\models \varphi_f$. This implies $\forall N \mathbf{sys}(N) \not\models \varphi_d$.

Similar behavior is exhibited by *token-ring* protocol and *queues* with two or more buffers. In all these cases, while the iterative procedure for limit computation does not converge directly, the acceleration operator forces termination.

Cache Coherence Cache coherence protocols [5] are used in multi-processor systems with shared memory, where each processor possesses its own private cache. The protocol we considered (from [14]) defines four distinct states for each processor – invalid, valid, shared and exclusive. Processors in invalid state have an outdated copy of the memory block in their cache; processors in valid and shared states have the current copy of memory block in their cache; a processor in exclusive state is the exclusive owner of the memory block.

Previous efforts [14] 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 this system. In contrast, we modeled the parameterized system and used a least fixed point formula φ to detect the presence of more than one processor in each of valid, shared or exclusive states – objective being to check $\exists N \mathbf{sys}(N) \models \varphi$ where $\mathbf{sys}(N)$ consists of N processors. The limit 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 $0 \not\models \varphi_f$ implying data consistency is maintained for system consisting of any number of processors.

Java Meta-lock The Java Meta-lock is a distributed algorithm that ensures fast mutually exclusive access of objects by Java threads [1]. The protocol involves synchronous communication between objects and threads and also between the threads themselves.

We first consider the system consisting of fixed number of threads and arbitrary number of objects and a least fixed point deadlock formula φ_d . Our objective is to check $\exists N \mathbf{sys}(k, N) \models \varphi_d$, where k is the fixed number of threads and N is the number of objects. In this example, the limit computation converges in two iterations to yield φ_f , since each `object` process behaves independently of any other `object`. Finally, $0 \not\models \varphi_f$ ensuring freedom from deadlock for all the members of the parameterized system $\mathbf{sys}(k, N)$.

Let us now consider the dual case with an arbitrary number of thread processes and a fixed number of objects. Using the same φ_d , our aim is to verify

$\exists \mathbf{M} \text{ sys}(\mathbf{M}, \mathbf{k}) \models \varphi_d$. The sequence of formulas generated by property transformer for the threads does not converge even with application of acceleration operator. The sequence converges only after the coarse approximation performed by the widening operator. The reason is that in case of meta-lock protocol each `thread` process can directly communicate with any other `thread` process. Hence ψ_i , the i^{th} formula in the sequence contains actions related to synchronization between the i^{th} thread and any other thread. When attempting to accelerate the convergence of the limit computation, we find that the selected witnesses of topological clashes (see Section 4.1) do not have an ancestor that simulates them, and hence no reduction is possible. In this case, the widening operator approximates the witness state with tt . Intuitively, this approximation implies that any transition sequence leading to interaction between threads will satisfy φ_d . However this approximation is too coarse since the resultant limit φ_f is such that $\mathbf{0} \models \varphi_f$. Due to the approximation, we cannot determine whether φ_d is modeled by the $\text{sys}(\mathbf{M}, \mathbf{k})$.

6 Conclusion

We described an automatic technique, based on program analysis techniques, 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. 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 [39]. The utility of the technique has been demonstrated by verifying a number of example parameterized systems with diverse characteristics in a uniform manner. The technique, however, is too approximate to provide useful results in certain cases where induction-based verification techniques have been successful (metalock with multiple threads and single object [36]). Development of abstractions of property transformers and widening operators which perform more fine-grained approximations is a topic of future research.

Acknowledgments. We would like thank to Dr. K. Narayan Kumar for his insightful suggestions. We are also thankful to the anonymous reviewers for their valuable comments. This work is supported in part by NSF grants EIA-9705998, CCR-9876242, EIA-9805735, N000140110967, and IIS-0072927.

References

1. O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y.S. Ramakrishna, and D. White. An efficient meta-lock for ubiquitous synchronization. In *OOPSLA*, 1999.
2. R. Alur and T. Henzinger. Reactive modules. In *LICS*, 1996.
3. H. R. Andersen. Partial model checking. In *LICS*, 1995.

4. H. R. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal μ -calculus. In *LICS*, 1994.
5. J. Archibald and J.L. Baer. Cache coherence protocols: Evaluation using a multi-processor simulation model. In *ACM TOCS*, 1986.
6. S. Basu, S. A. Smolka, and O. R. Ward. Model checking the Java Meta-Locking algorithm. In *ECBS*, 2000.
7. S. Berezin and D. Gurov. A compositional proof system for the modal μ -calculus and CCS. Technical Report CMU-CS-97-105, CMU, 1997.
8. J. Bradfield and C. Stirling. Modal logics and μ -calculi: an introduction. In *Handbook of Process Algebra*. Elsevier, 2001.
9. 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 *LICS*, 1990.
10. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 1986.
11. E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. In *ACM transactions on programming languages and systems*, 1997.
12. R. Cleaveland and B. Steffen. A linear-time model checking algorithm for the alternation-free modal μ -calculus. *FMSD*, 1993.
13. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
14. G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *CAV*, 2000.
15. G. Delzanno and A. Podelski. Model checking in CLP. In *TACAS*, 1999.
16. E. A. Emerson and C. S. Jutla. The complexity of tree automata and logics of programs. In *FOCS*, pages 328–337, 1988.
17. E.A. Emerson and K.S. Namjoshi. Reasoning about rings. In *POPL*, 1995.
18. E.A. Emerson and K.S. Namjoshi. Automated verification of parameterized synchronous systems. In *CAV*, 1996.
19. E.A. Emerson and K.S. Namjoshi. On model checking for non-deterministic infinite state systems. In *LICS*, 1998.
20. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS*, 1999.
21. O. Grumberg and D.E. Long. Model checking and modular verification. In *TOPLAS*, 1994.
22. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. In *JLP*, 1994.
23. T. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee. In *CAV*, 1998.
24. G. J. Holzmann. The model checker SPIN. *IEEE TSE*, 1997.
25. C. N. Ip and D. L. Dill. Better verification through symmetry reduction. In *FMSD*, 1996.
26. C.N. Ip and D.L. Dill. Verifying systems with replicated components in murphi. In *FMSD*, 1999.
27. Y. Kesten and A. Pnueli. Control and data abstraction: the cornerstones of practical formal verification. In *Intl. Journal on STTT*, 2000.
28. D. Kozen. Results on the propositional μ -calculus. *TCS*, 1983.
29. D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of linear networks processes. In *POPL*, 1997.
30. K.L. McMillan. Compositional rule for hardware design refinement. In *CAV*, 1997.
31. P. Mildner. *Type Domains form Abstract interpretation: A critical study*. PhD thesis, Uppsala University, 1999.

32. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
33. A. Pnueli and E. Shohar. Liveness and acceleration in parameterized verification. In *CAV*, 2000.
34. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, 1982.
35. A. Roychoudhury, K.N. Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, and S.A. Smolka. Verification of parameterized systems using logic-program transformations. In *TACAS*, 2000.
36. A. Roychoudhury and I.V. Ramakrishnan. Automated inductive verification of parameterized protocols. In *CAV*, 2001.
37. A. P. Sistla and V. Gyuris. Parameterized verification of linear networks using automata as invariants. *Formal Aspects of Computing*, 1999.
38. P. Wolper. Expressing interesting properties in propositional temporal logic. In *POPL*, 1986.
39. The XSB Group. The XSB logic programming system v2.1, 2000. Available from <http://www.cs.sunysb.edu/~sbprolog>.