Constrained Monotonic Abstraction: A CEGAR for Parameterized Verification

Parosh Aziz Abdulla[1], Yu-Fang Chen[2], Giorgio Delzanno[3], Frédéric Haziza[1], Chih-Duo Hong[2], and Ahmed Rezine[1]

[1] Uppsala University, Sweden
[2] Academia Sinica, Taiwan
[3] Università di Genova, Italy

Abstract. In this paper, we develop a counterexample-guided abstraction refinement (CEGAR) framework for *monotonic abstraction*, an approach that is particularly useful in automatic verification of safety properties for *parameterized systems*. The main drawback of verification using monotonic abstraction is that it sometimes generates spurious counterexamples. Our CEGAR algorithm automatically extracts from each spurious counterexample a set of configurations called a "Safety Zone" and uses it to refine the abstract transition system of the next iteration. We have developed a prototype based on this idea; and our experimentation shows that the approach allows to verify many of the examples that cannot be handled by the original monotonic abstraction approach.

1 Introduction

We investigate the analysis of safety properties for *parameterized systems*. A parameterized system consists of an arbitrary number of identical finite-state processes running in parallel. The task is to verify correctness regardless of the number of processes.

One of the most widely used frameworks for infinite-state verification uses *systems that are monotonic w.r.t. a well-quasi ordering* $\preceq$ [2,22]. This framework provides a scheme for proving termination of backward reachability analyses, which has already been used for the design of verification algorithms of various infinite-state systems (e.g., Petri nets, lossy channel systems) [8,20,21]. The main idea is the following. For a class of models, we find a preorder $\preceq$ on the set of configurations that satisfies the following two conditions (1) the system is monotonic w.r.t. $\preceq$ and (2) $\preceq$ is a well-quasi ordering (WQO for short). Then, backward reachability analysis from an upward closed set (w.r.t. $\preceq$) is guaranteed to terminate, which implies that the reachability problem of an upward closed set (w.r.t. $\preceq$) is decidable.

However, there are several classes of systems that do not fit into this framework, since it is hard to find a preorder that meets the aforementioned two conditions at the same time. An alternative solution is to first find a WQO $\preceq$ on the set of configurations and then apply *monotonic abstraction* [6,4,7] in order to *force* monotonicity. Given a preorder $\preceq$ on configurations, monotonic abstraction

P. Gastin and F. Laroussinie (Eds.): CONCUR 2010, LNCS 6269, pp. 86–101, 2010.

defines an abstract transition system for the considered model that is monotonic w.r.t. $\preceq$. More precisely, it considers a transition from a configuration c_1 to a configuration c_2 to be possible if there exists some smaller configuration $c_1' \preceq c_1$ that has a transition to c_2. The resulting abstract transition system is clearly monotonic w.r.t $\preceq$ and is an over-approximation of the considered model. Moreover, as mentioned, if $\preceq$ is a WQO, the termination of backward reachability analysis is guaranteed in the abstract transition system.

Monotonic abstraction has shown to be useful in the verification of heap manipulating programs [1] and parameterized systems such as *mutual exclusion* and *cache coherence* protocols [4,6]. In most of the benchmark examples for these classes, monotonic abstraction can generate abstract transition systems that are safe w.r.t to the desired properties (e.g. mutual exclusion). The reason is that, for these cases, we need only to keep track of simple constraints on individual variables in order to successfully carry out verification. However, there are several classes of protocols where we need more complicated invariants in order to avoid generating spurious counterexamples. Examples include cases where processes synchronize via *shared counters* (e.g. readers and writers protocol) or *reference counting* schemes used to handle a common set of resources (e.g. virtual memory management). For these cases, monotonic abstraction often produces spurious counterexamples, since it is not sufficiently precise to preserve the needed invariants. Therefore, we introduce in this paper a *counterexample-guided abstraction refinement (CEGAR)* approach to *automatically* and *iteratively* refine the abstract transition system and remove spurious counterexamples.

The idea of the CEGAR algorithm is as follows. It begins with an initial preorder $\preceq_0$, which is the one used in previous works on monotonic abstraction [6]. In the i-th iteration, it tries to verify the given model using monotonic abstraction w.r.t. the preorder $\preceq_{i-1}$. Once a *counterexample* is found in the abstract transition system, the algorithm simulates it on the concrete transition system. In case the counterexample is spurious, the algorithm extracts from it a set S of configurations called a "Safety Zone". The computation of the "Safety Zone"is done using *interpolation* [28,26]. The set S ("Safety Zone") is then used to *strengthen* the preorder that will be used in the next iteration. Monotonic abstraction produces a more accurate abstract transition system with the strengthened preorder. More precisely, in the $(i+1)$-th iteration, the algorithm works on an abstract transition system induced by monotonic abstraction and a preorder $\preceq_i := \{(c, c') |\ c \preceq_{i-1} c' \text{ and } c' \in S \Rightarrow c \in S\}$. Intuitively, the strengthened preorder forbids configurations inside a "Safety Zone"to use a transition from some smaller configuration (w.r.t $\preceq_{i-1}$) outside the "Safety Zone".

The strengthening of the preorder has an important property: It preserves WQO. That is, if $\preceq_{i-1}$ is a WQO, then $\preceq_i$ is also a WQO, for all $i > 0$. Therefore, the framework of monotonic systems w.r.t. a WQO can be applied to each abstract transition system produced by monotonic abstraction and hence termination is guaranteed for each iteration. Based on the method, we have implemented a prototype, and successfully used it to automatically verify several non-trivial examples, such as protocols synchronizing by shared counters and

reference counting schemes, that cannot be handled by the original monotonic abstraction approach.

Outline. We define parameterized systems and their semantics in Section 2. In Section 3, we first introduce monotonic abstraction and then give an overview of the CEGAR algorithm. In Section 4, we describe the details of the CEGAR algorithm. We introduce a *symbolic representation* of infinite sets of configurations called *constraint*. In Section 4, we show that all the *constraint* operations used in our algorithm are computable. In Section 6, we show that the termination of backward reachability checking is guaranteed in our CEGAR algorithm. Section 7 describes some extension of our model for parameterized system. In Section 8 we describe our experimentation. Finally, in Section 9, we conclude with a discussion of related tools and future works.

2 Preliminaries

In this section, we define a model for parameterized systems. We use $\mathbb{B}$ to denote the set $\{true, false\}$ of Boolean values, $\mathbb{N}$ to denote the set of natural numbers, and $\mathbb{Z}$ to denote the set of integers. Let P be a set and $\preceq$ be a binary relation on P. The relation $\preceq$ is a *preorder* on P if it is reflexive and transitive. Let $Q \subseteq P$, we define a *strengthening* of $\preceq$ by Q, written $\preceq_Q$, to be the binary relation $\preceq_Q := \{(c, c') |\ c \preceq c' \text{ and } c' \in Q \Rightarrow c \in Q\}$. Observe that $\preceq_Q$ is also a preorder on P.

Let X_N be a set of *numerical* variables ranging over $\mathbb{N}$. We use $\mathcal{N}(X_N)$ to denote the set of formulae which have the members of $\{x - y \diamond c, x \diamond c \mid x, y \in X_N, c \in \mathbb{Z}, \diamond \in \{\geq, =, \leq\}\}$ as atomic formulae, and which are closed under the Boolean connectives $\neg$, $\wedge$, $\vee$. Let X_B be a finite set of *Boolean* variables. We use $\mathcal{B}(X_B)$ to denote the set of formulae which have the members of X_B as atomic formulae, and which are closed under the Boolean connectives $\neg$, $\wedge$, $\vee$. Let X' be the set of *primed* variables $\{x' \mid x \in X\}$, which refers to the "next state" values of X.

2.1 Parameterized System

Here we describe our model of parameterized systems. A simple running example of a parameterized system is given in Fig. 1. More involved examples can be found in the tech. report [3]. The example in Fig. 1 is a readers and writers protocol that uses two shared variables; A numerical variable *cnt* (the read counter) is used to keep track of the number of processes in the "read" state and a Boolean variable *lock* is used as a semaphore. The semaphore is released when the writer finished writing or all readers finished reading (*cnt* decreased to 0).

A *parameterized system* consists of an unbounded but finite number of identical processes running in parallel and operating on a finite set of shared Boolean and numerical variables. At each step, one process changes its local state and checks/updates the values of shared variables. Formally, a *parameterized system* is a triple $\mathcal{P} = (Q, T, X)$, where Q is the set of *local states*, T is the set of

transition rules, and X is a set of shared variables. The set of shared variables X can be partitioned to the set of variables X_N ranging over $\mathbb{N}$ and X_B ranging over $\mathbb{B}$.

A transition rule $t \in T$ is of the form $\left[\, q \to r : stmt \,\right]$, where $q, r \in Q$ and *stmt* is a *statement* of the form $\phi_N \wedge \phi_B$, where $\phi_N \in \mathcal{N}(X_N \cup X'_N)$ and $\phi_B \in \mathcal{B}(X_B \cup X'_B)$. The formula ϕ_N controls variables ranging over $\mathbb{N}$ and ϕ_B controls Boolean variables. Taking the rule r_1 in Fig. 1 as an example, the statement says that: if the values of shared variables $cnt = 0$ and $lock = true$, then we are allowed to increase the value of *cnt* by 1, negate the value of *lock*, and change the local state of a process from t to r.

shared lock: Boolean, cnt: nat	
r_1:	$\left[\, t \to r : cnt = 0 \wedge cnt' = cnt + 1 \wedge lock \wedge \neg lock' \,\right]$
r_2:	$\left[\, t \to r : cnt >= 1 \wedge cnt' = cnt + 1 \,\right]$
r_3:	$\left[\, r \to t : cnt >= 1 \wedge cnt' = cnt - 1 \,\right]$
r_4:	$\left[\, r \to t : cnt = 1 \wedge cnt' = cnt - 1 \wedge \neg lock \wedge lock' \,\right]$
w_1:	$\left[\, t \to w : lock \wedge \neg lock' \,\right]$
w_2:	$\left[\, w \to t : \neg lock \wedge lock' \,\right]$
Initial: $t, lock$	

Fig. 1. Readers and writers protocol. Here t, r, w are "think", "read", and "write" states, respectively.

2.2 Transition System

A parameterized system $\mathcal{P} = (Q, T, X)$ induces an infinite-state transition system $(C, \longrightarrow)$ where C is the set of *configurations* and $\longrightarrow$ is the set of *transitions*.

A *configuration* $c \in C$ is a function $Q \cup X \to \mathbb{N} \cup \mathbb{B}$ such that (1) $c(q) \in \mathbb{N}$ gives the number of processes in state q if $q \in Q$, (2) $c(x) \in \mathbb{N}$ if $x \in X_N$ and (3) $c(x) \in \mathbb{B}$ if $x \in X_B$. We use $[x_1^{v_1}, x_2^{v_2}, \ldots, x_n^{v_n}, b_1, b_2, \ldots, b_m]$ to denote a configuration c such that (1) $c(x_i) = v_1$ for $1 \leq i \leq n$ and (2) $c(b) = true$ iff $b \in \{b_1, b_2, \ldots, b_m\}$.

The set of *transitions* is defined by $\longrightarrow := \bigcup_{t \in T} \xrightarrow{t}$. Let $c, c' \in C$ be two configurations and $t = \left[\, q \to r : stmt \,\right]$ be a transition rule. We have $(c, c') \in \xrightarrow{t}$ (written as $c \xrightarrow{t} c'$) if (1) $c'(q) = c(q) - 1$, (2) $c'(r) = c(r) + 1$, and (3) substituting each variable x in *stmt* with $c(x)$ and its primed version x' in *stmt* with $c'(x)$ produces a formula that is valid. For example, we have $\left[r^0, w^0, t^3, cnt^0, lock\right] \xrightarrow{r_1} \left[r^1, w^0, t^2, cnt^1\right]$ in the protocol model of Fig. 1. We use $\xrightarrow{*}$ to denote the transitive closure of $\longrightarrow$.

3 Monotonic Abstraction and CEGAR

We are interested in reachability problems, i.e., given sets of initial and bad configurations, can we reach any bad configuration from some initial configuration in the transition system induced by a given parameterized system.

We first recall the method of *monotonic abstraction* for the verification of parameterized systems and then describe an iterative and automatic CEGAR approach. The approach allows to produce more and more precise over-approximations of a given transition system from iteration to iteration. We assume a transition system $(C, \longrightarrow)$ induced by some parameterized system.

3.1 Monotonic Abstraction

Given an ordering $\trianglelefteq$ defined on C, monotonic abstraction produces an *abstract transition system* $(C, \rightsquigarrow)$ that is an over-approximation of $(C, \longrightarrow)$ and that is *monotonic* w.r.t. $\trianglelefteq$.

Definition 1 (Monotonicity). *A transition system* $(C, \rightsquigarrow)$ *is* monotonic *(w.r.t.* $\trianglelefteq$*) if for each* $c_1, c_2, c_3 \in C$, $c_1 \trianglelefteq c_2 \wedge c_1 \overset{t}{\rightsquigarrow} c_3 \Rightarrow \exists c_4.\ c_3 \trianglelefteq c_4 \wedge c_2 \overset{t}{\rightsquigarrow} c_4$.

The idea of monotonic abstraction is the following. A configuration c is allowed to use the outgoing transitions of any smaller configuration c' (w.r.t $\trianglelefteq$). The resulting system is then trivially monotonic and is an over-approximation of the original transition system. Formally, the abstract transition system $(C, \rightsquigarrow)$ is defined as follows. The set of configurations C is identical to the one of the concrete transition system. The set of *abstract transitions* is defined by $\rightsquigarrow := \bigcup_{t \in T} \overset{t}{\rightsquigarrow}$, where $(c_1, c_3) \in \overset{t}{\rightsquigarrow}$ (written as $c_1 \overset{t}{\rightsquigarrow} c_3$) iff $\exists c_2 \trianglelefteq c_1.\ c_2 \xrightarrow{t} c_3$. It is clear that $\overset{t}{\rightsquigarrow} \supseteq \xrightarrow{t}$ for all $t \in T$, i.e., $(C, \rightsquigarrow)$ over-approximates $(C, \longrightarrow)$.

In our previous works [4,6], we defined $\trianglelefteq$ to be a particular ordering $\preceq \subseteq C \times C$ such that $c \preceq c'$ iff (1) $\forall q \in Q. c(q) \leq c'(q)$, (2) $\forall n \in X_N.\ c(n) \leq c'(n)$, and (3) $\forall b \in X_B.\ c(b) = c'(b)$. Such an ordering has shown to be very useful in *shape analysis* [1] and in the verification of safety properties of *mutual exclusion* and *cache coherence* protocols [4,6]. In the CEGAR algorithm, we use $\preceq$ as the initial preorder.

3.2 Refinement of the Abstraction

Figure 2 gives an overview of the counterexample-guided abstraction refinement (CEGAR) algorithm. The algorithm works fully automatically and iteratively.

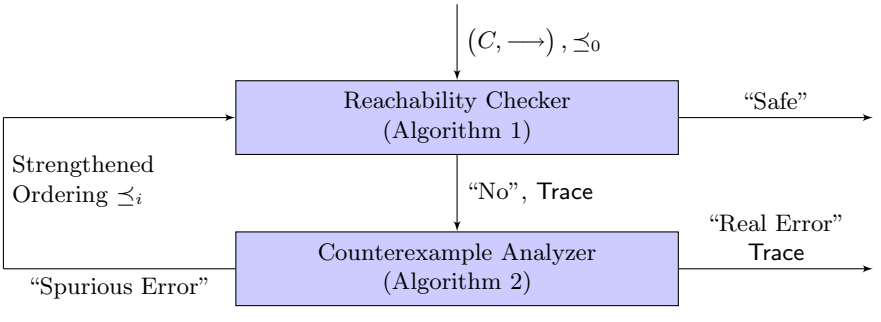

Fig. 2. An overview of the CEGAR algorithm (Algorithm 3)

In the beginning, a transition system $(C, \longrightarrow)$ and an initial preorder $\preceq_0$ (which equals the preorder $\preceq$ defined in the previous subsection) are given. The CEGAR algorithm (Algorithm 3) consists of two main modules, the *reachability checker* (Algorithm 1) and the *counterexample analyzer* (Algorithm 2). In the i-th iteration of the CEGAR algorithm, the *reachability checker* tests if bad configurations are reachable in the abstract transition system obtained from monotonic abstraction with the preorder $\preceq_{i-1}$. In case bad configurations are reachable, a *counterexample* is sent to the *counterexample analyzer*, which reports either "Real Error" or "Spurious Error". The latter comes with a strengthened order $\preceq_i$ (i.e., $\preceq_i \subset \preceq_{i-1}$). The strengthened order $\preceq_i$ will then be used in the $(i+1)$-th iteration of the CEGAR loop. Below we describe informally how $\preceq_{i-1}$ is strengthened to $\preceq_i$. The formal details are given in Section 4.

Strengthening the Preorder. As an example, we demonstrate using the protocol of Fig. 1 how to obtain $\preceq_1$ from $\preceq_0$. The set of bad configurations $Bad = \{c \mid c(r) \geq 1 \wedge c(w) \geq 1\}$ contains all configurations with at least one process in the "write" state and one process in the "read" state. The set of initial configurations $Init = \{c \mid c(w) = c(r) = c(cnt) = 0 \wedge c(lock)\}$ contains all configurations where all processes are in the "think" state, the value of the "cnt" equals 0, and the "lock" is available.

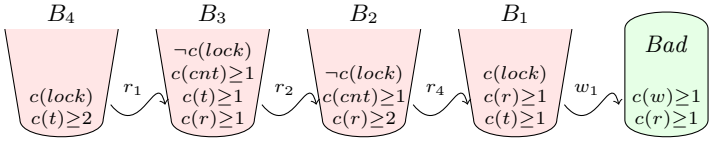

Fig. 3. The counterexample produced by backward reachability analysis on the readers and writers protocol. Notice that in the counterexample, $Init \cap B_4 \neq \emptyset$.

In iteration 1 of the CEGAR algorithm, the *reachability checker* produces a counterexample (described in Fig. 3) and sends it to the *counterexample analyzer*. More precisely, the *reachability checker* starts from the set Bad and finds the set B_1 contains all configurations that have (abstract) transitions $\overset{w_1}{\rightsquigarrow}$ to the set Bad. That is, each configuration in B_1 either has a concrete transition $\xrightarrow{w_1}$ to Bad or has some smaller configuration (w.r.t $\preceq_0$) with a concrete transition $\xrightarrow{w_1}$ to Bad. It then continues the search from B_1 and finds the set B_2 that have (abstract) transitions in $\overset{r_4}{\rightsquigarrow}$ to B_1. The sets B_3 and B_4 can be found in a similar way. It stops when B_4 is found, since $B_4 \cap Init \neq \emptyset$.

The *counterexample analyzer* simulates the received counterexample in the concrete transition system. We illustrate this scenario in Fig. 4. It starts from the set of configuration $F_4 = Init \cap B_4$[1] and checks if any bad configurations can be reached following a sequence of transitions $\xrightarrow{r_1};\xrightarrow{r_2};\xrightarrow{r_4};\xrightarrow{w_1}$. Starting from F_4, it

[1] The set of initial configurations that can reach bad configurations follows the sequence of transitions $\overset{r_1}{\rightsquigarrow};\overset{r_2}{\rightsquigarrow};\overset{r_4}{\rightsquigarrow};\overset{w_1}{\rightsquigarrow}$ in the abstract transition system

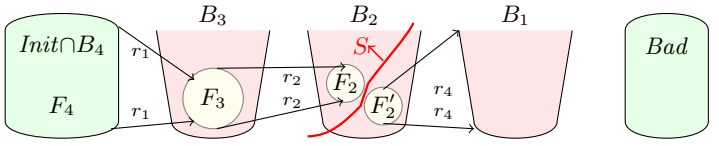

Fig. 4. Simulating the counterexample on the concrete system. Here $F_4 = Init \cap B_4 = \{c \mid c(t) \geq 2 \wedge c(w) = c(r) = c(cnt) = 0 \wedge c(lock)\}$, $F_3 = \{c \mid c(t) \geq 1 \wedge c(w) = 0 \wedge c(r) = c(cnt) = 1 \wedge \neg c(lock)\}$, $F_2 = \{c \mid c(cnt) = c(r) = 2 \wedge c(w) = 0 \wedge \neg c(lock)\}$, and $F_2' = \{c \mid c(cnt) = 1 \wedge c(r) \geq 1 \wedge \neg c(lock)\}$

finds the set F_3 which is a subset of B_3 and which can be reached from F_4 via the transition $\xrightarrow{r_1}$. It continues from F_3 and then finds the set F_2 in a similar manner via the transition $\xrightarrow{r_2}$. However, there exists no transition $\xrightarrow{r_4}$ starting from any configuration in $F_2 = \{c \mid c(cnt) = c(r) = 2 \wedge c(w) = 0 \wedge \neg c(lock)\}$. Hence the simulation stops here and concludes that the counterexample is *spurious*.

In the abstract transition system, all configurations in F_2 are able to reach B_1 via transition $\overset{r_4}{\rightsquigarrow}$ and from which they can reach Bad via transition $\overset{w_1}{\rightsquigarrow}$. Notice that there exists no concrete transition $\xrightarrow{r_4}$ from F_2 to B_1, but the abstract transition $\overset{r_4}{\rightsquigarrow}$ from F_2 to B_1 does exist. The reason is that all configurations in F_2 have some smaller configuration (w.r.t. $\preceq_0$) with a transition $\xrightarrow{r_4}$ to B_1. Let F_2' be the set of configurations that indeed have some transition $\xrightarrow{r_4}$ to B_1. It is clear that F_2 and F_2' are disjoint.

Therefore, we can remove the spurious counterexample by preventing configurations in F_2 from falling to some configuration in F_2' (thus also preventing them from reaching B_1). This can be achieved by first defining a set of configurations S called a "Safety Zone" with $F_2 \subseteq S$ and $F_2' \cap S = \emptyset$ and then use it to *strengthen* the preorder $\preceq_0$, i.e., let $\preceq_1 := \{(c, c') \mid c \preceq_0 c' \text{ and } c' \in S \Rightarrow c \in S\}$. In Section 4, we will explain how to use interpolation techniques [28,26] in order to automatically obtain a "Safety Zone" from a counterexample.

4 The Algorithm

In this section, we describe our CEGAR algorithm for monotonic abstraction. First, we define some concepts that will be used in the algorithm. Then, we explain the two main modules, *reachability checker* and *counterexample analyzer*. The *reachability checker* (Algorithm 1) is the backward reachability analysis algorithm on monotonic systems [2], which is possible to apply since the abstraction induces a monotonic transition system. The *counterexample analyzer* (Algorithm 2) checks a counterexample and extracts a "Safety Zone" from the counterexample if it is spurious. The CEGAR algorithm (Algorithm 3) is obtained by composing the above two algorithms. In the rest of the section, we assume a parameterized system $\mathcal{P} = (Q, T, X)$ that induces a transition system $(C, \longrightarrow)$.

4.1 Definitions

A *substitution* is a set $\{x_1 \leftarrow e_1, x_2 \leftarrow e_2, \ldots, x_n \leftarrow e_n\}$ of pairs, where x_i is a variable and e_i is a variable or a value of the same type as x_i for all $1 \leq i \leq n$. We assume that all variables are distinct, i.e., $x_i \neq x_j$ if $i \neq j$. For a formula θ and a substitution S, we use $\theta[S]$ to denote the formula obtained from θ by simultaneously replacing all free occurrences of x_i by e_i for all $x_i \leftarrow e_i \in S$. For example, if $\theta = (x_1 > x_3) \wedge (x_2 + x_3 \leq 10)$, then $\theta[x_1 \leftarrow y_1, x_2 \leftarrow 3, x_3 \leftarrow y_2] = (y_1 > y_2) \wedge (3 + y_2 \leq 10)$.

Below we define the concept of a *constraint*, a symbolic representation of configurations which we used in our algorithm. In this section, we define a number of operations on constraints. In Section 5, we show how to compute those operations.

We use $Q^{\#}$ to denote the set $\{q^{\#} \mid q \in Q\}$ of variables ranging over $\mathbb{N}$ in which each variable $q^{\#}$ is used to denote the number of processes in the state q. Define the set of formulae $\Phi := \{\phi_N \wedge \phi_B \mid \phi_N \in \mathcal{N}(Q^{\#} \cup X_N), \phi_B \in \mathcal{B}(X_B)\}$ such that each formula in Φ is a *constraint* that characterizes a potentially infinite set of configurations. Let ϕ be a constraint and c be a configuration. We write $c \vDash \phi$ if $\phi[\{q^{\#} \leftarrow c(q) \mid q \in Q\}][\{x \leftarrow c(x) \mid x \in X_N\}][\{b \leftarrow c(b) \mid b \in X_B\}]$ is a valid formula. We define the set of configurations characterized by ϕ as $[\![\phi]\!] := \{c \mid c \in C \wedge c \vDash \phi\}$. We define an *entailment relation* $\sqsubseteq$ on constraints, where $\phi_1 \sqsubseteq \phi_2$ iff $[\![\phi_1]\!] \subseteq [\![\phi_2]\!]$. We assume that the set of initial configurations *Init* and bad configurations *Bad* can be characterized by constraints ϕ_{Init} and ϕ_{Bad}, respectively.

For a constraint ϕ, the function $\mathsf{Pre}_t(\phi)$ returns a constraint characterizing the set $\{c \mid \exists c' \in [\![\phi]\!] \wedge c \xrightarrow{t} c'\}$, i.e., the set of configurations from which we can reach a configuration in $[\![\phi]\!]$ via transitions in $\xrightarrow{t}$; and $\mathsf{Post}_t(\phi)$ returns a constraint characterizing the set $\{c \mid \exists c' \in [\![\phi]\!] \wedge c' \xrightarrow{t} c\}$, i.e., the set of configurations that can be reached from some configuration in $[\![\phi]\!]$ via transitions in $\xrightarrow{t}$. For a constraint ϕ and a preorder $\preceq$ on the set of configurations, the function $\mathsf{Up}_{\preceq}(\phi)$ returns a constraint such that $[\![\mathsf{Up}_{\preceq}(\phi)]\!] = \{c' \mid \exists c \in [\![\phi]\!] \wedge c \preceq c'\}$, i.e., the upward closure of $[\![\phi]\!]$ w.r.t. the ordering $\preceq$. A *trace* (from ϕ_1 to ϕ_{n+1}) in the abstract transition system induced by monotonic abstraction and the preorder $\preceq$ is a sequence $\phi_1; t_1; \ldots; \phi_n; t_n; \phi_{n+1}$, where $\phi_i = \mathsf{Up}_{\preceq}(\mathsf{Pre}_{t_i}(\phi_{i+1}))$ and $t_i \in T$ for all $1 \leq i \leq n$. A *counterexample* (w.r.t. $\preceq$) is a trace $\phi_1; t_1; \ldots; \phi_n; t_n; \phi_{n+1}$ with $[\![\phi_1]\!] \cap [\![\phi_{Init}]\!] \neq \emptyset$ and $\phi_{n+1} = \phi_{Bad}$.

We use $\mathsf{Var}(\phi)$ to denote the set of variables that appear in the constraint ϕ. Given two constraints ϕ_A and ϕ_B such that $\phi_A \wedge \phi_B$ is unsatisfiable. An *interpolant* ϕ of (ϕ_A, ϕ_B) (denoted as $\mathsf{ITP}(\phi_A, \phi_B)$) is a formula that satisfies (1) $\phi_A \implies \phi$, (2) $\phi \wedge \phi_B$ is unsatisfiable, and (3) $\mathsf{Var}(\phi) \subseteq \mathsf{Var}(\phi_A) \cap \mathsf{Var}(\phi_B)$. Such an interpolant can be automatically found, e.g., using off-the-shelf interpolant solvers such as FOCI [28] and CLP-prover [29]. In particular, since $\phi_A, \phi_B \in \Phi$, if we use the "split solver" algorithm equipped with theory of difference bound [26] to compute an interpolant, the result will always be a formula in Φ (i.e., a constraint).

4.2 The Reachability Checker

Algorithm 1. The reachability checker

input : A preorder $\preceq$ over configurations, constraints ϕ_{Init} and ϕ_{Bad}
output: "Safe" or "No" with a *counterexample* $\phi_1; t_1; \ldots; \phi_n; t_n; \phi_{Bad}$

1 $\mathsf{Next} := \{(\phi_{Bad}, \phi_{Bad})\}$, $\mathsf{Processed} := \{\}$;
2 **while** Next *is not empty* **do**
3 Pick and remove a pair $(\phi_{Cur}, \mathsf{Trace})$ from Next and add it to Processed;
4 **if** $[\![\phi_{Cur} \wedge \phi_{Init}]\!] \neq \emptyset$ **then return** *"No",* Trace;
5 **foreach** $t \in T$ **do**
6 $\phi_{Pre} = \mathsf{Up}_{\preceq}(\mathsf{Pre}_t(\phi_{Cur}))$;
7 $old = \exists(\phi, \bullet) \in \mathsf{Next} \cup \mathsf{Processed}.\phi \sqsubseteq \phi_{Pre}$;
8 **if** $\neg old$ **then** Add $(\phi_{Pre}, \phi_{Pre}; t; \mathsf{Trace})$ to Next;
9 **return** *"Safe"*;

Let $\preceq$ be a preorder on C and $(C, \rightsquigarrow)$ be the abstract transition system induced by the parameterized system $\mathcal{P}$ and the preorder $\preceq$. Algorithm 1 checks if the set $[\![\phi_{Init}]\!]$ is backward reachable from $[\![\phi_{Bad}]\!]$ in the abstract transition system $(C, \rightsquigarrow)$. It answers "Safe" if none of the initial configurations are backward reachable. Otherwise, it answers "No". In the latter case, it returns a *counterexample* $\phi_1; t_1; \ldots; \phi_n; t_n; \phi_{Bad}$. The algorithm uses a set Next to store constraints characterizing the sets of configurations from which it will continue the backward search. Each element in Next is a pair (ϕ, Trace), where ϕ is a constraint characterizing a set of backward reachable configurations (in the abstract transition system) and Trace is a trace from ϕ to ϕ_{Bad}. Initially, the algorithm puts in Next the constraint ϕ_{Bad}, which describes the bad configurations, together with a trace contains a singleton element namely ϕ_{Bad} itself (Line 1). In each loop iteration (excepts the last one), it picks a constraint ϕ_{Cur} (together with a trace to ϕ_{Bad}) from Next (Line 3). For each transition rule $t \in T$, the algorithm finds a constraint ϕ_{Pre} characterizing the set of configurations backward reachable from $[\![\phi_{Cur}]\!]$ via $\overset{t}{\rightsquigarrow}$ (Line 6). If there exists no constraint in Next that is larger than ϕ_{Pre} (w.r.t. $\sqsubseteq$), ϕ_{Pre} (together with a trace to ϕ_{Bad}) is added to Next (Line 7).

4.3 The Counterexample Analyzer

Given a counterexample $\phi_1; t_1; \ldots; \phi_n; t_n; \phi_{n+1}$, Algorithm 2 checks whether it is spurious or not. If spurious, it returns a constraint ϕ_S that describes a "Safety Zone"that will be used to strengthen the preorder.

As we explained in Section 3, we simulate the counterexample forwardly (Line 1-6). The algorithm begins with the constraint $\phi_1 \wedge \phi_{Init}$. If the counterexample is spurious, we will find a constraint ϕ in the i-th loop iteration for some i : $1 \leq i \leq n$ such that none of the configurations in $[\![\phi]\!]$ has transition $\xrightarrow{t_i}$ to $[\![\phi_{i+1}]\!]$ (Line 3). For this case, it computes the constraint ϕ' characterizing the set of configurations with transitions $\xrightarrow{t_i}$ to $[\![\phi_{i+1}]\!]$ (Line 4) and then computes a constraint characterizing a "Safety Zone".

Algorithm 2. The counterexample analyzer.

input : A counterexample $\phi_1; t_1; \ldots; \phi_n; t_n; \phi_{n+1}$
output: "Real Error" or "Spurious Error" with a constraint ϕ_S

1 $\phi = \phi_1 \wedge \phi_{Init}$;
2 **for** $i = 1$ *to* n **do**
3 **if** $[\![\mathsf{Post}_{t_i}(\phi)]\!] = \emptyset$ **then**
4 $\phi' = \mathsf{Pre}_{t_i}(\phi_{i+1})$;
5 **return** *"Spurious Error",* $\mathsf{ITP}(\phi, \phi')$;
6 $\phi = \mathsf{Post}_{t_i}(\phi) \wedge \phi_{i+1}$;
7 **return** *"Real Error"*;

As we explained in Section 3, a "Safety Zone"is a set S of configurations that satisfies (1) $[\![\phi]\!] \subseteq S$ and (2) $S \cap [\![\phi']\!] = \emptyset$. Therefore, the constraint ϕ_S characterizing the "Safety Zone"should satisfy (1) $\phi \implies \phi_S$ and (2) $\phi_S \wedge \phi'$ is not satisfiable. The *interpolant* of (ϕ, ϕ') is a natural choice of ϕ_S that satisfies the aforesaid two conditions. Hence, in this case the algorithm returns $\mathsf{ITP}(\phi, \phi')$ (Line 5). If the above case does not happen, the algorithm computes a constraint characterizing the next set of forward reachable configurations in the counterexample (Line 6) and proceeds to the next loop iteration. It returns "Real Error" (Line 7) if the above case does not happen during the forward simulation.

4.4 The CEGAR Algorithm of Monotonic Abstraction

Algorithm 3. A CEGAR algorithm for monotonic abstraction

input : An initial preorder $\preceq_0$ over configurations, constraints ϕ_{Init} and ϕ_{Bad}
output: "Safe" or "Real Error" with a *counterexample* $\phi_1; t_1; \ldots; \phi_n; t_n; \phi_{Bad}$

1 $i = 0$;
2 **while** *true* **do**
3 $result = \mathsf{ReachabilityChecker}(\preceq_i, \phi_{Init}, \phi_{Bad})$;
4 **if** *result="No",* Trace **then**
5 $type = \mathsf{CounterexampleAnalyzer}(\mathsf{Trace})$;
6 **if** *type="Spurious Error",* ϕ_S **then** $i = i + 1, \preceq_i := \mathsf{Str}(\preceq_{i-1}, \phi_S)$;
7 **else return** *"Real Error",* Trace
8 **else return** *"Safe"*

In Algorithm 3, we describe the CEGAR approach for monotonic abstraction with the initial preorder $\preceq_0$. As described in Section 3, the algorithm works iteratively. In the i-th iteration, in Line 3, we invoke the *reachability checker* (Algorithm 1) using a preorder $\preceq_{i-1}$. When a counterexample is found, the *counterexample analyzer* (Algorithm 2) is invoked to figure out if the counterexample is real (Line 8) or spurious. In the latter case, the *counterexample analyzer* generates a constraint characterizing a "Safety Zone"and from which Algorithm 3 computes a strengthened preorder $\preceq_i$ (Line 6 and 7). The function $\mathsf{Str}(\preceq_{i-1}, \phi_S)$ in Line 8 strengthens the preorder $\preceq_{i-1}$ by the set of configurations $[\![\phi_S]\!]$.

5 Constraint Operations

In this section we explain how to compute all the constraint operations used in the algorithms in Section 4. Recall that Φ denotes the set of formulae $\{\phi_N \wedge \phi_B \mid \phi_N \in \mathcal{N}(Q^{\#} \cup X_N), \phi_B \in \mathcal{B}(X_B)\}$, where each formula in Φ is a constraint representing a set of configurations. We define $\Psi := \{\phi_N \wedge \phi_B \mid \phi_N \in \mathcal{N}(Q^{\#} \cup Q^{\#'} \cup X_N \cup X'_N), \phi_B \in \mathcal{B}(X_B \cup X'_B)\}$, where each formula in Ψ defines a relation between sets of configurations. Observe that formulae in Φ and in Ψ are closed under the Boolean connectives and substitution.

Lemma 1. *[19] Both Φ and Ψ are closed under projection (existential quantification) and the projection functions are computable.*

Lemma 2. *[19] The satisfiability problem of formulae in Φ and Ψ is decidable.*

Below we explain how to preform the constraint operations used in the algorithms in Section 4. For notational simplicity, we define $\mathsf{V} := Q^{\#} \cup X_N \cup X_B$ and $\mathsf{V}' := Q^{\#'} \cup X'_N \cup X'_B$. Let ϕ be a formula in Φ (respectively, Ψ) and X a set of variables in V (respectively, $\mathsf{V} \cup \mathsf{V}'$), we use $\exists X.\ \phi$ to denote some formula ϕ' in Φ (respectively, Ψ) obtained by the quantifier elimination algorithm (Lemma 1).

Pre and Post. The transition relation $\xrightarrow{t}$ for $t = \left[\, q \to r : \mathit{stmt} \,\right] \in T$ can be described by the formula $\theta^t := \mathit{stmt} \wedge q^{\#'} = q^{\#} - 1 \wedge r^{\#'} = r^{\#} + 1$, which is in Ψ. For a constraint ϕ, $\mathsf{Pre}_t(\phi) = \exists \mathsf{V}'.\ (\theta^t \wedge \phi[\{x \leftarrow x' \mid x \in \mathsf{V}\}]) \in \Phi$ and $\mathsf{Post}_t(\phi) = (\exists \mathsf{V}.\ (\theta^t \wedge \phi))[\{x' \leftarrow x \mid x \in \mathsf{V}\}] \in \Phi$. Both functions are computable.

Entailment. Given two constraints ϕ_1 and ϕ_2, we have $\phi_1 \sqsubseteq \phi_2$ iff $\phi_1 \wedge \neg\phi_2$ is unsatisfiable, which can be automatically checked. In practice, constraints can be easily translated into disjunctions of difference bound matrices (DBM) and hence a *sufficient* condition for entailment can be checked by standard DBM operations [19].

Intersection with Initial States. Let ϕ_{Init} be a constraint characterizing the initial configurations and ϕ_B be a constraint characterizing a set of configurations. We have $[\![\phi_{\mathit{Init}}]\!] \cap [\![\phi_B]\!] \neq \emptyset$ iff $\phi_{\mathit{Init}} \wedge \phi_B$ is satisfiable.

Strengthening. Here we explain how to strengthen an ordering $\preceq$ w.r.t a constraint $\phi_S \in \Phi$, providing that $\preceq$ is expressed as a formula $\phi_{\preceq} \in \Psi$. The strengthened order can be expressed as the formula $\phi_{\preceq_S} := \phi_{\preceq} \wedge (\phi_S \vee \neg\phi_S[\{x \leftarrow x' \mid x \in \mathsf{V}\}])$. Intuitively, for two configurations c_1 and c_2, the formula says that $c_1 \preceq_S c_2$ iff $c_1 \preceq c_2$ and either c_1 is in the "Safety Zone"or c_2 is not in the "Safety Zone".

Remark 1. The initial preorder $\preceq_0$ of our algorithm can be expressed as the formula $\bigwedge_{x \in Q^{\#} \cup X_N,\ x' \in Q^{\#'} \cup X'_N} .\ x \leq x' \wedge \bigwedge_{b \in X_B,\ b' \in X'_B} .\ (b \wedge b') \vee (\neg b \wedge \neg b')$, which is in Ψ. The constraint extracted from each spurious counterexample is in Φ if the algorithm in [26] is used to compute the interpolant. Since the initial preorder is a formula in Ψ and the constraint used for strengthening is in Φ, the formula for the strengthened order is always in Ψ and computable.

Upward Closure. We assume that the ordering $\preceq$ is expressed as a formula $\phi_\preceq \in \Psi$ and the constraint $\phi \in \Phi$. The upward closure of ϕ w.r.t. $\preceq$ can be captured as $\mathsf{Up}_\preceq(\phi) := (\exists \mathsf{V}.\ (\phi \wedge \phi_\preceq))[\{x' \leftarrow x \mid x \in \mathsf{V}\}]$, which is in Φ.

6 Termination

In this section, we show that each loop iteration of our CEGAR algorithm terminates. We can show by Dickson's lemma [18] that the initial preorder $\preceq$ is a WQO. An ordering over configurations is a WQO iff for any infinite sequence $c_0, c_1, c_2, \ldots$ of configurations, there are i and j such that $i < j$ and $c_i \preceq c_j$. Moreover, we can show that the strengthening of a preorder also preserves WQO.

Lemma 3. *Let S be a set of configurations. If $\preceq$ is a WQO over configurations then $\preceq_S$ is also a WQO over configurations.*

If a transition system is monotonic w.r.t. a WQO over configurations, backward reachability analysis, which is essentially a fix-point calculation, terminates within a finite number of iterations [2]. The abstract transition system is monotonic. In Section 5, we show that all the constraint operations used in the algorithms are computable. Therefore, in each iteration of the CEGAR algorithm, the termination of the *reachability checker* (Algorithm 1) is guaranteed. Since the length of a counterexample is finite, the termination of the *counterexample analyzer* (Algorithm 2) is also guaranteed. Hence, we have the following lemma.

Lemma 4. *Each loop iteration of the CEGAR algorithm (Algorithm 3) is guaranteed to terminate.*

7 Extension

The model described in Section 2 can be extended to allow some additional features. For example, (1) dynamic creation of processes $\left[\cdot \rightarrow q : \mathit{stmt}\right]$, (2) dynamic deletion of processes $\left[q \rightarrow \cdot : \mathit{stmt}\right]$, and (3) synchronous movement $\left[q_1, q_2, \ldots, q_n \rightarrow r_1, r_2, \ldots, r_n : \mathit{stmt}\right]$. Moreover, the language of the *statement* can be extended to any formula in Presburger arithmetic. For all of the new features, we can use the same constraint operations as in Section 5; the extended transition rule still can be described using a formula in Ψ, Presburger arithmetic is closed under Boolean connectives, substitution, and projection and all the mentioned operations are computable.

8 Case Studies and Experimental Results

We have implemented a prototype and tested it on several case studies of classical synchronization schemes and reference counting schemes, which includes

Table 1. Summary of experiments of case studies. *Interpolant* denotes the kind of interpolant prover we use, where DBM denotes the difference bound matrix based solver, and CLP denotes the CLP-prover. *Pass* indicates whether the refinement procedure can terminate with a specific interpolant prover. *Time* is the execution time of the program, measured by the bash time command. *#ref* is the number of refinements needed to verify the property. *#cons* is the total number of constraints generated by the reachability checker. For each model, we use *#t*, *#l*, *#s* to denote the number of transitions, the number of local variables, and the number of shared variables, respectively. All case studies are described in details in tech. report [3].

model	interpolant	pass	time	#ref	#cons	#t	#l	#s
readers/writers	DBM	√	0.04 sec	1	90	6	5	2
	CLP	√	0.08 sec	1	90			
refined readers/writers priority to readers	DBM	√	3.9 sec	2	3037	8	5	3
	CLP	X	-	-	-			
refined readers/writers priority to writers	DBM	√	3.5 sec	1	2996	12	7	5
	CLP	√	68 sec	4	39191			
sleeping barbers	DBM	√	3.9 sec	1	1518	10	15	1
	CLP	√	4.1 sec	1	1518			
pmap reference counting	DBM	√	0.1 sec	1	249	25	4	7
	CLP	√	0.1 sec	1	249			
reference counting gc	DBM	√	0.02 sec	1	19	7	4	1
	CLP	√	0.05 sec	1	19			
missionary & cannibals	DBM	X	-	-	-	7	7	1
	CLP	√	0.1 sec	3	86			
swimming pool v2	DBM	√	0.2 sec	2	59	6	0	10
	CLP	√	0.2 sec	2	55			

readers/writers protocol, sleeping barbers problem, the missionaries/cannibals problem [11], the swimming pool protocol [11,23], and virtual memory management. These case studies make use of shared counters (in some cases protected by semaphores) to keep track of the number of current references to a given resource. Monotonic abstraction returns spurious counterexamples for all the case studies. In our experiments, we use two interpolating procedures to refine the abstraction. One is a homemade interpolant solver based on difference bound matrices [26]; the other one is the CLP-prover [29], an interpolant solvers based on constraint logic programming. The results, obtained on an Intel Xeon 2.66GHz processor with 8GB memory, are listed in Table 1. It shows that our CEGAR method efficiently verifies many examples in a completely automatic manner.

We compare our approach with three related tools: the ALV tool [14], the Interproc Analyzer [24], and FASTer [11] based on several examples (and their variants) from our case studies. The results are summarized in Table 2. Note that these tools either perform an exact forward analysis where the invariant is exactly represented (FASTer), or try to capture all possible invariants of a certain form (ALV and Interproc Analyzer). In these two approaches, the verification of the property is deduced from the sometimes expensively generated invariants. The main difference between our approach and the other ones is that

Table 2. Summary of tool comparisons. For cma, we selected the best results among the ones obtained from DBM and CLP. For FASTer, we tested our examples with library MONA and the backward search strategy. For the other tools, we just used the default settings. In our experiment, ALV outputted "unable to verify" for the missionaries/cannibals model and failed to verify the other test cases after one day of execution. FASTer failed to verify four of the six test cases within a memory limit of 8GB. Interproc Analyzer gave false positives for examples other than the swimming pool protocol and the missionaries/cannibals model. That is, it proved reachability for models where the bad states were not reachable.

Model	**Tool**	**Pass**	**Result**	**Model**	**Tool**	**Pass**	**Result**
swimming pool protocol v2	cma	$\surd$	0.2 sec	pmap reference counting	cma	$\surd$	0.1 sec
	FASTer	X	oom		FASTer	$\surd$	85 sec
	Interproc	$\surd$	2.7 sec		Interproc	X	false positive
	ALV	X	timeout		ALV	X	timeout
Model	**Tool**	**Pass**	**Result**	**Model**	**Tool**	**Pass**	**Result**
missionary & cannibals	cma	$\surd$	0.1 sec	readers writers pri. readers	cma	$\surd$	3.9 sec
	FASTer	X	oom		FASTer	$\surd$	3 min 44 sec
	Interproc	$\surd$	2 sec		Interproc	X	false positive
	ALV	X	cannot verify		ALV	X	timeout
Model	**Tool**	**Pass**	**Result**	**Model**	**Tool**	**Pass**	**Result**
missionary & cannibals v2	cma	$\surd$	0.2 sec	readers writers pri. readers v2	cma	$\surd$	0.5 sec
	FASTer	X	oom		FASTer	X	oom
	Interproc	X	false positive		Interproc	X	false positive
	ALV	X	timeout		ALV	X	timeout

we concentrate on minimal constraints to track the violation of the property at hand. Using upward closed sets as a symbolic representation efficiently exploits the monotonicity of the abstract system where the analysis is exact yet efficient.

9 Related and Future Work

We have presented a method for refining monotonic abstraction in the context of verification of safety properties for parameterized systems. We have implemented a prototype based on the method and used it to automatically verify parameterized versions of synchronization and reference counting schemes. Our method adopts an iterative counter-example guided abstraction refinement (CEGAR) scheme. Abstraction refinement algorithms for forward/backward analysis of well-structured models have been proposed in [25,16]. Our CEGAR scheme is designed instead for undecidable classes of models. Other tools dealing with the verification of similar parameterized systems can be divided into two categories: exact and approximate. In Section 8, we compare our method to a representative from each category. The results confirm the following. Exact techniques, such as FASTer [11], restrict their computations to under-approximations of the set of reachable states. They rely on computing the exact effect of particular categories of loops, like non-nested loops for instance, and may not terminate in general. On the contrary, our method is guaranteed to terminate at each iteration.On the

other hand, approximate techniques like ALV and the Interproc Analyzer [14,24], rely on widening operators in order to ensure termination. Typically, such operators correspond to extrapolations that come with a loss of precision. It is unclear how to refine the obtained over-approximations when false positives appear in parameterized systems like those we study.

Also, the refinement method proposed in the present paper allows us to automatically verify new case studies (e.g. reference counting schemes) that cannot be handled by regular model checking [27,17,9,12,30,13], monotonic abstractions [6,4,7] (they give false positives), environment abstraction [15], and invisible invariants [10]. It is important to remark that a distinguished feature of our method with respect to methods like invisible invariants and environment abstraction is that we operate on abstract models that are still infinite-state thus trying to reduce the loss of precision in the approximation required to verify a property.

We currently work on extensions of our CEGAR scheme to systems in which processes are linearly ordered. Concerning this point, in [5] we have applied a manually supplied strengthening of the subword ordering to automatically verify a formulation of Szymanski's algorithm (defined for ordered processes) with non-atomic updates.

References

1. Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziz, F., Rezine, A.: Monotonic abstraction for programs with dynamic memory heaps. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 341–354. Springer, Heidelberg (2008)
2. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: LICS 1996, pp. 313–321 (1996)
3. Abdulla, P.A., Chen, Y.-F., Delzanno, G., Haziza, F., Hong, C.-D., Rezine, A.: Constrained monotonic abstraction: a cegar for parameterized verification. Tech. report 2010-015, Uppsala University, Sweden (2010)
4. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with global conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007)
5. Abdulla, P.A., Delzanno, G., Rezine, A.: Approximated context-sensitive analysis for parameterized verification. In: FMOODS 2009/FORTE 2009, pp. 41–56 (2009)
6. Abdulla, P.A., Henda, N.B., Delzanno, G., Rezine, A.: Regular model checking without transducers (on efficient verification of parameterized systems). In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007)
7. Abdulla, P.A., Henda, N.B., Delzanno, G., Rezine, A.: Handling parameterized systems with non-atomic global conditions. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 22–36. Springer, Heidelberg (2008)
8. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Info. Comput. 127(2), 91–101 (1996)
9. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J.: Regular model checking made simple and efficient. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 116–130. Springer, Heidelberg (2002)

10. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.: Parameterized verification with automatically computed inductive assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001)
11. Bardin, S., Leroux, J., Point, G.: FAST extended release. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 63–66. Springer, Heidelberg (2006)
12. Boigelot, B., Legay, A., Wolper, P.: Iterating transducers in the large. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 223–235. Springer, Heidelberg (2003)
13. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004)
14. Bultan, T., Yavuz-Kahveci, T.: Action language verifier. In: ASE 2001, p. 382 (2001)
15. Clarke, E., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2005)
16. Cousot, P., Ganty, P., Raskin, J.-F.: Fixpoint-guided abstraction refinements. LNCS. Springer, Heidelberg (2007)
17. Dams, D., Lakhnech, Y., Steffen, M.: Iterating transducers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 286–297. Springer, Heidelberg (2001)
18. Dickson, L.E.: Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. Amer. J. Math. 35, 413–422 (1913)
19. Dill, D.: Timing assumptions and verification of finite-state concurrent systems. In: AVMFSS 1989, pp. 197–212 (1989)
20. Emerson, E., Namjoshi, K.: On model checking for non-deterministic infinite-state systems. In: LICS 1998, pp. 70–80 (1998)
21. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: LICS 1999 (1999)
22. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! TCS 256(1-2), 63–92 (2001)
23. Fribourg, L., Olsén, H.: Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 213–227. Springer, Heidelberg (1997)
24. Gal Lalire, M.A., Jeannet, B.: A web interface to the interproc analyzer, `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`
25. Geeraerts, G., Raskin, J.-F., Begin, L.V.: Expand, enlarge and check... made efficient. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 394–404. Springer, Heidelberg (2005)
26. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
27. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. TCS 256, 93–112 (2001)
28. McMillan, K.L.: An interpolating theorem prover. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 101–121. Springer, Heidelberg (2004)
29. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint solving for interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)
30. Touili, T.: Regular Model Checking using Widening Techniques. ENTCS 50(4) (2001)