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 use 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 using *systems that are monotonic w.r.t. a well-quasi ordering* $\preceq$ [2, 24]. This framework provides a scheme for proving termination of backward reachability analysis, which has already been used for the design of verification algorithms of various infinite-state systems (e.g., Petri nets, lossy channel systems) [7, 21, 23]. 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. $\preceq$) 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* [5, 3, 6] in order to *force* monotonicity. Given a preorder $\preceq$ on configurations, monotonic abstraction 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 [3, 5]. 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 includes 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 [5]. 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 "Safety Zone" is done using the *interpolation* technique [30, 28]. 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 Appendix. 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:	$\left[w \to t : \neg lock = 0 \wedge lock'\right]$
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 [3, 5], 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 [3, 5]. In the CEGAR algorithm, we use $\preceq$ as the initial preorder.

3.2 Refinement of the Abstraction

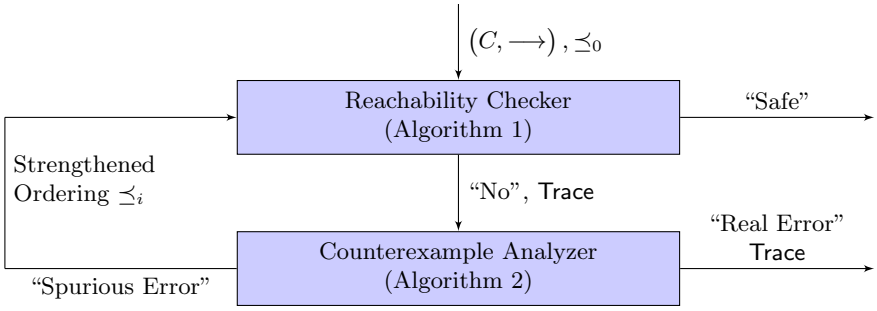

Fig. 2. An overview of the CEGAR algorithm (Algorithm 3).

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

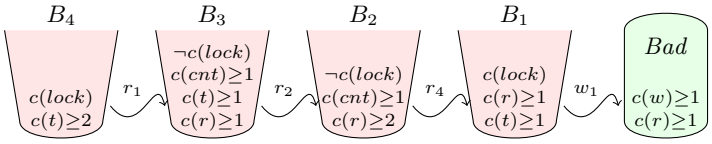

Fig. 3. The counterexample produced by backward reachability analysis on the readers and writers protocol. Notice that in the counterexample, $Init \cap B_4 \neq \emptyset$.

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 $B_4 \cap Init \neq \emptyset$.

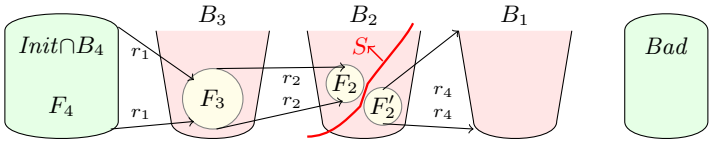

Fig. 4. Simulating the counterexample on the concrete system. Here $F_4 = Init \cap B_4 = \{c \mid c(t) \geq 2 \wedge c(w) = c(r) = c(cnt) = 0 \wedge c(lock)\}$, $F_3 = \{c \mid c(t) \geq 1 \wedge c(w) = 0 \wedge c(r) = c(cnt) = 1 \wedge \neg c(lock)\}$, $F_2 = \{c \mid c(cnt) = c(r) = 2 \wedge c(w)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

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 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

[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

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 [30, 28] 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 [30] and CLP-prover [31]. In particular, since $\phi_A, \phi_B \in \Phi$, if we use the "split solver" algorithm equipped with theory of difference bound [28] 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: either (1) "Safe" or (2) "No" with a *counterexample* $\phi_1; t_1; \ldots; \phi_n; t_n; \phi_{Bad}$

1 $\mathsf{Next} := \{(\phi_{Bad}, \phi_{Bad})\}$;
2 **while** Next *is not empty* **do**
3 Pick and remove a pair $(\phi_{Cur}, \mathsf{Trace})$ from Next;
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 **if** $\neg\exists(\phi, \bullet) \in \mathsf{Next}.\phi_{Pre} \sqsubseteq \phi$ **then** Add $(\phi_{Pre}, \phi_{Pre}; t; \mathsf{Trace})$ to Next;
8 **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

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"*;

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".

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

Here 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. *[20] Both Φ and Ψ are closed under projection (existential quantification) and the projection functions are computable.*

Lemma 2. *[20] The satisfiability problem of formulae in Φ and Ψ is decidable.*

Below we explain how to preform these constraint operations. 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 $t = \left[q \to r : stmt \right] \in T$ can be described by the formula $\theta^t := stmt \wedge q^{\#'} = q^{\#} - 1 \wedge r^{\#'} = r^{\#} + 1$, 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 [20].

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 $[\![\phi_{Init}]\!] \cap [\![\phi_B]\!] \neq \emptyset$ iff $\phi_{Init} \wedge \phi_B$ 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 [28] is used to compute 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 [19] 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 $\left[\cdot \to q : stmt\right]$, (2) dynamic deletion of processes $\left[q \to \cdot : stmt\right]$, and (3) synchronous movement $\left[q_1, q_2, \ldots, q_n \to r_1, r_2, \ldots, r_n : stmt\right]$. 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 readers/writers protocol, sleeping barbers problem, the missionaries/cannibals problem [11], the swimming pool protocol [11, 25], 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 [28]; the other one is the CLP-prover [31], 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 [26], and FASTer [11]. Three representative examples from our case studies are used for the comparison, namely, the refined readers/writers protocol, the missionaries/cannibals problem and the swimming pool protocol. We model these examples in the modeling languages of the tools. The results are summarized in Table 2.

Model	Interpolant	Pass	Time	#ref	#cons
readers/writers	DBM	$\surd$	0.04 sec	1	90
	CLP	$\surd$	0.08 sec	1	90
refined readers/writers	DBM	$\surd$	3.9 sec	2	3037
priority to readers	CLP	X	-	-	-
refined readers/writers	DBM	$\surd$	3.5 sec	1	2996
priority to writers	CLP	$\surd$	68 sec	4	39191
sleeping	DBM	$\surd$	3.9 sec	1	1518
barbers	CLP	$\surd$	4.1 sec	1	1518
reference	DBM	$\surd$	0.02 sec	1	19
counting	CLP	$\surd$	0.05 sec	1	19
pmap reference	DBM	$\surd$	0.1 sec	1	249
counting	CLP	$\surd$	0.1 sec	1	249
missionary and	DBM	X	-	-	-
cannibals	CLP	$\surd$	0.1 sec	3	86
swimming	DBM	$\surd$	0.2 sec	2	59
pool v2	CLP	$\surd$	0.2 sec	2	55

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. All case studies are described in details in appendix.

9 Related and Future Work

We have presented a method for refining monotonic abstraction in the context of verification of safety properties 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 [27, 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, 26], 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.

Model	Tool	Pass	Result	Model	Tool	Pass	Result
missionary cannibals	cma	$\surd$	0.1 sec	pmap ref. counting	cma	$\surd$	0.1 sec
	FASTer	X	out of memory		FASTer	$\surd$	85 sec
	Interproc	X	timeout		Interproc	$\surd$	2.5 sec
	ALV	X	can not verify		ALV	X	can not verify
Model	**Tool**	**Pass**	**Result**	**Model**	**Tool**	**Pass**	**Result**
swimming pool proc. v2	cma	$\surd$	0.2 sec	readers writers priority readers	cma	$\surd$	3.9 sec
	FASTer	X	out of memory		FASTer	$\surd$	186 sec
	Interproc	X	timeout		Interproc	X	timeout
	ALV	X	can not verify		ALV	X	can not verify

Table 2. Summary of tool comparisons. The memory limit of each execution is 8GB. For cma, we list the best result obtained from DBM and CLP. The computation time of the Interproc Analyzer is limited to 1 minute by the tool itself. ALV outputs "unable to verify" for all test cases. FASTer fails to verify the swimming tool protocol and the missionaries/cannibals model within a memory limit of 8GB. For FASTer, we tested our examples with libraries MONA, LASH and OMEGA, and applied the forward and the backward search strategies in turn. For the other tools, we just used the default settings.

Also, the refinement method proposed in the present paper allows us to automatically verify new case studies (e.g. reference counting schemes) that cannot handled by regular model checking [29, 17, 8, 12, 32, 13], monotonic abstractions [5, 3, 6] (they give false positives), environment abstraction [15], and invisible invariants [9]. 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 [4] 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. P. A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziz, and A. Rezine. Monotonic abstraction for programs with dynamic memory heaps. In *Proc.* 20^{th} *Int. Conf. on Computer Aided Verification*, 2008.
2. P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. LICS '96,* 11^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.
3. P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *Proc.* 19^{th} *Int. Conf. on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 145–157, 2007.
4. P. A. Abdulla, G. Delzanno, and A. Rezine. Approximated context-sensitive analysis for parameterized verification. In *FMOODS '09/FORTE '09: Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems*, pages 41–56, Berlin, Heidelberg, 2009. Springer-Verlag.
5. P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *Proc. TACAS '07,* 13^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 721–736. Springer Verlag, 2007.

6. P. A. Abdulla, N. B. Henda, G. Delzanno, and A. Rezine. Handling parameterized systems with non-atomic global conditions. In *Proc. VMCAI '08,* 9^{th} *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, 2008.
7. P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.
8. P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model checking made simple and efficient. In *Proc. CONCUR 2002,* 13^{th} *Int. Conf. on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130, 2002.
9. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In Berry, Comon, and Finkel, editors, *Proc.* 13^{th} *Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 221–234, 2001.
10. D. Bacon and V. Rajan. Concurrent cycle collection in reference counted systems. *ECOOP 2001Object-Oriented Programming*, pages 207–235.
11. S. Bardin, J. Leroux, and G. Point. FAST extended release. In *Computer Aided Verification*, pages 63–66. Springer.
12. B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In *Proc.* 15^{th} *Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 223–235, 2003.
13. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *CAV04*, Lecture Notes in Computer Science, pages 372–386, Boston, July 2004. Springer Verlag.
14. T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 382, Washington, DC, USA, 2001. IEEE Computer Society.
15. E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *Proc. VMCAI '06,* 7^{th} *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141, 2006.
16. P. Cousot, P. Ganty, and J.-F. Raskin. Fixpoint-guided abstraction refinements. Lecture Notes in Computer Science. Springer Verlag, 2007.
17. D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, 2001.
18. R. David and H. Alla. *Petri nets and Grafcet*. Prentice Hall, 1992.
19. L. E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *Amer. J. Math.*, 35:413–422, 1913.
20. D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite-State Systems*, volume 407 of *Lecture Notes in Computer Science*, 1989.
21. E. Emerson and K. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Proc. LICS '98,* 13^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 70–80, 1998.
22. M. Emmi, R. Jhala, E. Kohler, and R. Majumdar. Verifying reference counting implementations. In *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 352–367, Berlin, Heidelberg, 2009. Springer-Verlag.
23. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. LICS '99,* 14^{th} *IEEE Int. Symp. on Logic in Computer Science*, 1999.
24. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.
25. L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. *CONCUR'97: Concurrency Theory*, pages 213–227.
26. M. A. Gal Lalire and B. Jeannet. A web interface to the interproc analyzer. http://pop-art.inrialpes.fr/interproc/interprocweb.cgi.
27. G. Geeraerts, J.-F. Raskin, and L. V. Begin. Expand, enlarge and check... made efficient. In *Proc.* 16^{th} *Int. Conf. on Computer Aided Verification*, Lecture Notes in Computer Science. Springer Verlag, 2005.
28. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *Proc. TACAS '06,* 12^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, 2006.
29. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256:93–112, 2001.
30. K. L. McMillan. An interpolating theorem prover. In *Proc. TACAS '04,* 10^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, 2004.
31. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *Verification, Model Checking, and Abstract Interpretation*, pages 346–362. Springer.
32. T. Touili. Regular Model Checking using Widening Techniques. *Electronic Notes in Theoretical Computer Science*, 50(4), 2001. Proc. Workshop on Verification of Parametrized Systems (VEPAS'01), Crete, July, 2001.

```
int readcount = 0;
semaphore wsem = 1, x = 1;

void reader(){
    while(1){
        wait(x);
        readcount++;                          void writer(){
        if (readcount==1) wait(wsem);         while(1){
        signal(x);                                wait(wsem)
        doReading();                              doWriting();
        wait(x);                                  signal(wsem)
        readcount--;                          }
        if (readcount==0)signal(wsem);
        signal(x);
    }
}
```

Fig. 5. Reader/writers with priority to readers.

A Description of case-studies

A.1 Readers and Writers: Priority to Readers

The RW problem is a classic problem for which design of synchronization and concurrency mechanisms can be tested. The problem is defined as follows. There is a resource that is shared among a number of processes. Any number of readers may simultaneously read to the data area. Only one writer at a time may write to the data area. If a writer is writing to the data area, no reader may read it. If there is at least one reader reading the data area, no writer may write to it. Readers only read and writers only write. A process that reads and writes to a data area must be considered a writer.

One possible strategy to control the access to the shared resource is to give priority to readers as in the pseudo-code of Fig. 5. In this solution, readers wait that no writers are inside the critical section and then get the lock on semaphore *wsem*. This blocks writers until there are incoming readers. The last reader unlock *wsem* and let waiting writers to enter critical section. We model this concurrent program using the parameterized system in Fig. 6. Every processes is modelled with a set of transitions with constraints on a set of shared variables. We use $lockR$ and $lockW$ to model resp. mutex x and *wsem* of Fig. 5. Furthermore, we use *count* to model variable *readcount*. When $lockR = 1$ then mutex is unlocked, locked otherwise. Location $test_1$ is used to simulate the first if-then-else on *count* in the reader's code of Fig. 5 (in the enter section). Location $test_2$ is used to simulate the if-then-else in the exit section. In this example we consider the mutual exclusion property for readers and writers. Monotonic

Shared : $lockR, lockW \in 0, 1, count : int$

Reader :
$think \longrightarrow test_1 : lockR = 1, lockR' = 0, count' = count + 1$
$test_1 \longrightarrow read : count = 1, lockW = 1, lockW' = 0, lockR' = 1$
$test_1 \longrightarrow read : count \geq 2, lockR' = 1$
$read \longrightarrow test_2 : count' = count - 1, lockR = 1, lockR' = 0$
$test_2 \longrightarrow think : count = 0, lockW' = lockR' = 1$
$test_2 \longrightarrow think : count \geq 1, lockR' = 1$

Writer :
$think \longrightarrow write : lockW = 1, lockW' = 0$
$write \longrightarrow think : lockW' = 1$

Initial state : $lockW = 1, count = 0 \bullet think, think, \ldots$
Bad states : $\phi = (read, write)$

Fig. 6. Parameterized model for readers writers with priority to readers.

abstraction cannot verify this property and returns a spurious counterexample. As shown in Table 1, our CEGAR method automatically checks the property for any number of readers and writers after one step of refinement for a total execution time of 3.9 seconds.

A.2 Readers and Writers: Priority to Writers

Another possible strategy for solving the reader/writer problem is to give priority to writers as in the pseudo-code of Fig. 7. The idea here is to delay readers that requires access when there are waiting writers (i.e. if a reader is in critical section, writers are waiting, incoming readers cannot jump into the critical section but have to wait for the writers). This is achieved by introducing an additional semaphore $rsem$ needed here to create an additional barrier for readers and an additional counter *writecount* to keep track of the number of waiting writers.

We model this concurrent program using the parameterized system in Fig. 8. We use $lockZ$, $lockR$ and $lockW$ to model resp. mutex z, $rsem$, and $wsem$ (we directly lock x using atomic local transitions) Furthermore, we use $countR$ and $countW$ to model variables *readcount* and *writecount*, resp.

Location $test_1$ is used to simulate the first if-then-else on *count* in the reader's code of Fig. 5 (in the enter section). In this example we consider the mutual exclusion property for readers and writers. Monotonic abstraction cannot verify this property and returns a spurious counterexample. As shown in Table 1, with DBM interpolant solver, our CEGAR method automatically checks the property for any number of readers and writers after 1 step of refinement for a total execution time less than 4 seconds.

A.3 Sleeping Barber

The sleeping barber problem is a classic inter-process communication and synchronization problem between multiple operating system processes. The prob-

```
int readcount, writecount = 0;
semaphore rsem, wsem, x,y,z = 1; //

void reader(){
   while(1){
     wait(z);
      wait(rsem);
       wait(x);
        readcount++;
        if (readcount==1) wait(wsem);
       signal(x);
      signal(rsem);
     signal(z);
     doReading();
     wait(x);
        readcount--;
        if (readcount==0) signal(wsem);
     signal(x);
   }
}
```

```
void writer(){
  while(1){
      wait(y);
        writecount++;
        if (writecount==1) wait(rsem);
     signal(y);
     wait(wsem);
     doWriting();
     signal(wsem);
     wait(y);
        writecount--;
        if (writecount==0) signal(rsem);
     signal(y);
  }
}
```

Fig. 7. Reader/writers with priority to writers.

lem is analogous to that of keeping a barber working when there are customers, resting when there are none and doing so in an orderly manner. The barber and his customers represent aforementioned processes. As shown in Fig. 13, the most common solution involves using three semaphores: one for any waiting customers, one for the barber (to see if he is idle), and the third ensures mutual exclusion. When a customer arrives, he attempts to acquire the mutex, and waits until he has succeeded. The customer then checks to see if there is an empty chair for him (either one in the waiting room or the barber chair), and if none of these are empty, leaves. Otherwise the customer takes a seat thus reducing the number available (a critical section). The customer then signals the barber to awaken through his semaphore, and the mutex is released to allow other customers (or the barber) the ability to acquire it. If the barber is not free, the customer then waits. The barber sits in a perpetual waiting loop, being awakened by any waiting customers. Once he is awoken, he signals the waiting customers through their semaphore, allowing them to get their hair cut one at a time. We model this concurrent program using the parameterized system in Fig. 12. We use shared variables *cust* and *barber* to model the corresponding generic semaphore. Furthermore, we use variable *mutex* to model the mutex with the same name. Furthermore, to emphasize the similarities with the other example, we use a shared counter *avail* to model the current number of available chairs (i.e. $avail = N - waiting$). We also introduce a counter *chair* that

Shared : $lockZ, lockR, lockW \in 0, 1, countR, countW : int$

Reader :
$think \longrightarrow waitR_1 : lockZ = 1, lockZ' = 0$
$waitR_1 \longrightarrow waitR_2 : lockR = 1, lockR' = 0$
$waitR_2 \longrightarrow read : countR = 0, lockW = 1,$
$\qquad countR' = 1, lockW' = 0, lockZ' = lockR' = 1$
$waitR_2 \longrightarrow read : countR >= 1, lockZ' = 1, lockR' = 1$
$read \longrightarrow think : countR = 1, countR' = 0, lockW' = 1$
$read \longrightarrow think : countR >= 2, countR' = countR - 1$

Writer :
$think \longrightarrow waitW : countW = 0, lockR = 1, lockR' = 0$
$think \longrightarrow waitW : countW >= 1$
$waitW \longrightarrow write : lockW = 1, lockW' = 0, countW' = countW + 1$
$write \longrightarrow releaseW : lockW' = 1$
$releaseW \longrightarrow think : countW = 1, countW' = 0, lockR' = 1$
$releaseW \longrightarrow think : countW >= 2, countW' = countW - 1$

Initial state : $countW = countR = 0, lockZ = lockW = lockR = 1 \bullet\ think, think, \ldots$
Bad states : $read, write$

Fig. 8. Readers writers with priority to writers.

represent the corresponding resources (they are not used for synchronization but just to keep track of the physical presence of chairs). N is here a parameter (i.e. a shared variable that is never updated). Location *skip* indicates the failure of the test $avail = 0$ in the body of the code of a customer. If location *skip* the customer releases the mutex and goes back to the initial state.

In this example we would like to verify that the counter *avail* is coherent with the current number of available resources (*chair*), i.e., that it never happens that a customer is in location *skip* while $chair \geq 1$. Monotonic abstraction cannot verify this property and returns a spurious counterexample. As shown in Table 1, our CEGAR method automatically checks the property for any number of customers and for any value assigned to N. The test requires 1 step of refinement for a total execution time of about 4 seconds.

A.4 Reference Counting

Other interesting case-studies come from the applications that use reference counting to maintain consistent information of data shared among different processes/processors [22]. The scheme is based on the following idea. For each shared resource, the resource manager keeps track of the current number of references by using a counter. Critical operations are executed only when the reference counter is zero. Instances of this scheme can be found in several file systems (e.g. reference counter associated to Unix inodes), virtual memory manager (e.g. to keep track of uses of physical pages), and multiprocessor systems In this section we present as a case study the analysis of the virtual memory manager

```
const int CHAIRS = 5;
semaphore customers, barbers=0,
mutex mutex = 1;
int waiting = 0;

void barber(){
    while(1) {
        wait(customers);
        wait(mutex);
        waiting = waiting - 1;
        signal(barbers);
        signal(mutex);
        cut_hair();
    }
}

void customer(){
 while(1) {
    wait(mutex);
    if (waiting < CHAIRS) {
        waiting++;
        signal(customers);
        signal(mutex);
        wait(barbers);
        get_haircut();
     }
    else signal(mutex);
 }}
```

Fig. 9. Sleeping barber.

described in [22]. The manager uses a table of counter to associate the number of references to each physical page. Process environments maintain a local page table in which virtual pages are associated to physical pages. Environments are created and deallocated dynamically. They can be linked together to form a virtual address space. They can request to map a physical page to a given virtual page, to unmap a virtual page, to map a physical page to another environment. Critical operations on physical pages are performed only if the corresponding counter is zero.

To model this application, we consider as our starting point the pmap.c program manually enriched with assertions (with skolem constants) described in [22]. The assertions can be used here to extract a parameterized model of the virtual memory manager in which the number of environments and the value of counters is not fixed a priori. More specifically, we fix a given physical page P (referenced object). We consider then two types of environments: env_0 if P is not mapped in its virtual address, env_1 if P is mapped in its virtual address. We then keep a reference counter rc for keeping track of the number of environments that have a reference to P (for each environment we count the presence of P in its page table). As in the pmap.c program in [22] we consider a main program that non-deterministically invoke the functionalities of the manager, i.e., (de)allocation of a new environment, allocation of a virtual address, (un)mapping of a virtual address, and a special state used to check consistency of the reference counting scheme.

The main loop is modelled with the help of a monitor process that has states $loop, env\_alloc, \ldots$. Processes can synchronize using rules of the form

Parameter : $N \geq 1$
Shared : $mutex \in \{0,1\}, barb, cust, avail, chair : int$

Barber :
$b0 \longrightarrow b1 : cust >= 1, cust' = cust - 1, mutex = 1, mutex' = 0$
$b1 \longrightarrow b2 : avail < N, avail' = avail + 1, chair' = chair + 1$
$b2 \longrightarrow b3 : barb' = barb + 1, mutex' = 1$
$b3 \longrightarrow b0$

Customer :
$c0 \longrightarrow c1 : mutex = 1, mutex' = 0$
$c1 \longrightarrow skip : avail = 0$
$skip \longrightarrow c0 : mutex' = 1$
$c1 \longrightarrow c2 : avail \geq 1, chair \geq 1,$
$\qquad chair' = chair - 1, avail' = avail - 1, cust' = cust + 1, mutex' = 1$
$c2 \longrightarrow c3 : barb \geq 1, barb' = barb - 1$
$c3 \longrightarrow c0$

Initial state : $cust = barb = 0, avail = chair = N, mutex = 1 \bullet b0, c0, \ldots, c0$
Bad states : $chair \geq 1 \bullet skip$

Fig. 10. Sleeping Barber.

$p_1, \ldots, p_n \longrightarrow p'_1, \ldots, p'_n : \varphi$, meaning that process in state p_i moves to p'_i for $i : 1, \ldots, n$ provided φ is satisfied. Creation and deletion is modelled by producing or consuming a state. For instance, creation of environments is modelled with a synchronization rule in which the monitor moves back to *loop* and a new process with state env_0 is created. Deallocation of an environment of type env_0 simply removes it from the current configuration. For environments of type env_1 we also have to decrement rc. The other operations is modelled from the perspective of a generic environment of type env_0/env_1, of page P and of its reference counter rc. For instance, the allocation of a physical page pp to a virtual page vp gives rise to several cases: vp can be already mapped to P, unmapped or mapped to another page. pp can be either P or another physical page. If $pp = P$ an environment moves to state env_1. In the first rule we assume that P does not occur in its vm so we have to increment rc. In the second rule we assume that P is already present. In the third rule we assume that $pp \neq P$ but P is already in the vm. Thus, we have to decrement rc and move to env_0. The other operations are modelled in the same spirit.

In this example we would like to verify that the counter rc is coherent with the current number of environments that have references to P, i.e., it is not possible to fire last transition and reach a configuration with at least one occurrence of process *bad*. As in the other example, monotonic abstraction cannot verify this property. Indeed, it returns a counterexample due to the loss of synchrony between rc and the number of env_1 processes in the abstract transition relation with unconstrained relation. However, as shown in Table 1, our CEGAR method automatically checks the property for any number of customers and for any value

```
typedef struct env {
    int env_mypp;
    int env_pgdir[NVPAGES];
    ...
} env_t;

int pages[NPPAGES];
...
int page_alloc(env_t *env, int vp) {
  int pp = page_getfree();
  if (pp < 0) return -1;
  if (env->env_pgdir[vp] >= 0) pages[env->env_pgdir[vp]]--;
  env->env_pgdir[vp] = pp;
  pages[pp]++;
  return 0;
}
...
int page_unmap(env_t *env, int vp) {
  if (env->env_pgdir[vp] >= 0) {
     pages[env->env_pgdir[vp]]--;
     env->env_pgdir[vp] = -1;
  }
}
...
```

Fig. 11. Fragment of pmap.c example.

assigned to N. The test requires 1 step of refinement for a total execution time of 0.1 second.

A.5 Reference-counting Garbage Collection

Garbage Collection is the automatic reclamation of computer storage [10]. In many systems, programmers must explicitly reclaim heap memory at some point in the program, by using a "free" statement. Systems with a garbage collector free the programmer from this burden. The garbage collector is used to find data objects that are no longer in use and make their space available for reuse by the running program. An object is considered *garbage* (and subject to reclamation) if it is not reachable by the running program via any path of pointer traversals. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never traverse a dangling pointer into a deallocated object.

There are essentially two types of garbage collectors: tracing garbage collectors and reference-counting garbage collectors. For that latter, each object keeps a count of the number of references to it. An object's reference count is incremented when a reference to it is created, and decremented when a reference

Let $L = \{env\_alloc, env\_free, page\_alloc, page\_map, page\_unmap, check\}$ **in**
Shared : $rc : int$

Main :
$loop \longrightarrow loc : true \ for \ each \ loc \in L$

Env_alloc :
$env\_alloc \longrightarrow loop, env_0 : true$

Env_free :
$env\_free, env_0 \longrightarrow loop : true$
$env\_free, env_1 \longrightarrow loop : rc >= 1, rc' = rc - 1$

Page_alloc :
$page\_alloc, env_0 \longrightarrow loop, env_1 : rc' = rc + 1$
$page\_alloc, env_1 \longrightarrow loop, env_1 : true$
$page\_alloc, env_1 \longrightarrow loop, env_0 : rc >= 1, rc' = rc - 1$
$page\_alloc, env_0 \longrightarrow loop, env_0 : true$

Page_map :
$page\_map, env_0, env_0 \longrightarrow loop, env_0, env_0 : true$
$page\_map, env_0, env_1 \longrightarrow loop, env_1, env_1 : rc' = rc + 1$
$page\_map, env_0, env_1 \longrightarrow loop, env_1, env_0 : rc >= 1$
$page\_map, env_0, env_1 \longrightarrow loop, env_0, env_0 : rc >= 1, rc' = rc - 1$
$page\_map, env_0, env_1 \longrightarrow loop, env_0, env_1 : true$
$page\_map, env_1, env_1 \longrightarrow loop, env_1, env_1 : true$

Page_unmap :
$page\_unmap, env_0 \longrightarrow loop, env_0 : true$
$page\_unmap, env_1 \longrightarrow loop, env_0 : rc >= 1, rc = rc - 1$
$page\_unmap, env_1 \longrightarrow loop, env_1 : true$

Check consistency :
$check, env_1 \longrightarrow loop, env_1, bad : rc = 0$

Initial state : $loop, env_0, \ldots, env_0, \ldots$
Bad states : $bad \geq 1$

Fig. 12. A Parameterized Model for the Pmap Example in [22].

is destroyed. The object should be garbage (and therefore reclaimed) when the reference count reaches zero.

A reference counting garbage collector must devise a special algorithm to reclaim cyclic garbage. We verified the algorithm of Bacon et al. [10], without the cyclic garbage collector. This garbage collector is a concurrent system (ie. it runs concurrently with other threads on other processors, as opposed to a *stop-the-world* garbage collector), but not parallel (ie. it is a single-threaded application). Objects are allocated with a reference count 1. Each thread running on some processor performs updates to objects on the heap and depending on its type, the update is enqueued as an increment or a decrement into respective buffers. When appropriate, a garbage collection is triggered, and the *collector* thread is scheduled on the first processor. It simply collects the increments and decrements that were differed in the buffers by the threads running on that

```
Trigger(){
    if (! TestAndSet(GCLock))
        Schedule(NextEpoch, 1)
        return true
    else
        return false
}

NextEpoch(){
    if (CPU.Id < NCPU)
        ProcessEpoch()
        SynchronizeMemory() // to see the buffer updates
        Schedule(NextEpoch,CPU.Id+1)
    else
        Collect()
}

Collect(){
    ProcessIncrements()
    ProcessDecrements()
    //CollectCycles()
    Notify(CPU.Epoch)
    FetchAndStore(GCLock, 0)
    CPU.Epoch = CPU.Epoch + 1
}
```

Fig. 13. Reference-Counting Garbage Collector.

processor. It then prepares the processor for the next garbage collection (such as allocating new buffers and restarting the interrupted thread) and schedules itself on the next processor. When it arrives at the last processor, it performs the collected increments and decrements on the reference counts. It is the only thread to update the reference counts.

This garbage collector is a producer/consumer system: the running threads produce operations on reference counts, which are placed into buffers and periodically turned over to the collector. We model this concurrent system using the parameterized system in Fig. 14. Any thread can perform update to heap objects but the increments and decrements are buffered and only processed when the garbage collection is triggered. There is a lock to prevent two garbage collections to be triggered at the same time, but we do not need to model it in our model. We distinguish 3 modes: a normal mode (n) in the case the garbage collection is not triggered and two phases for the garbage collection. In order to avoid data races, increments are performed first (mode $g1$), and then decrements are performed (mode $g2$).

We keep $userV$ and rcV as the number of references and the reference count respectively for a given object V. We would like to verify that the count rcV

Shared : $n, g1, g2, userV, rcV, incV, decV : int$
Normal mode :
$Enqueue\ Increment : n \longrightarrow n : userV' = userV + 1, incV' = incV + 1$
$Enqueue\ Decrement : n \longrightarrow n : userV' = userV - 1, decV' = decV + 1$
$Trigger\ GC : n \longrightarrow g1$
Garbage mode :
$Process\ Increment : g1 \longrightarrow g1 : incV' = incV - 1, rcV' = rcV + 1$
$Start\ G2 : g1 \longrightarrow g2 : incV = 0$
$Process\ Decrement : g2 \longrightarrow g2 : decV' = decV - 1, rcV' = rcV - 1$
$End\ GC : g2 \longrightarrow n : decV = 0$
Initial state : $n = 1, g1 = g2 = 0, incV = decV = 0$
Bad states : $g2 = 1, userV \geq 1 \bullet rcV = 0$

Fig. 14. Model for Reference-Counting Garbage Collector.

never reaches zero while there are references to the object ($userV$), ie. that an object has been reclaimed while threads are still using it. Note that while the collector thread is in mode $g1$ or $g2$, the threads from other processors are on mode n and can enqueue increments and decrements, but they would be enqueued into other buffers.

A.6 Missionaries and Cannibals

The missionaries and cannibals problem is to decide whether it is possible for three missionaries and three cannibals to cross a river using a boat, under the constraints that 1) the boat can carry at most three people, and 2) for both banks and in the boat, there cannot be more cannibals than missionaries. An example is described in Fig. 15. For the missionaries and cannibals, we use shared variables ml, mr, cl, cr to record their numbers on the banks, and use mb, cb to record their numbers in the boat. $maxb$ is a constant upper-bounding the number of people in the boat. We use a state variable to represent the current position of the boat, which is either the left bank or the right bank. Given that the constraints are met, the boat can choose one of the following two *atomic* actions in each step: loading one person, or crossing the river and then unloading all passengers. In the example, the initial values of ml and cl are parameterized by $M = 3$ and $C \geq 3$, respectively. [2] This means that the total number of cannibals is $C \geq 3$, and the total number of missionaries is $M = 3$. Now the problem reduces to verify that if the configuration $\{mr = M, cr = C\}$ is reachable from the initial configuration $\{ml = M, cl = C\}$. Monotonic abstraction cannot verify this property and returns a spurious counterexample. On the other hand, our CEGAR method automatically checks the property for instances ($M = 1, C \geq$

[2] Note that the parameterization here does not mean a generalization. Indeed, instances ($M \geq m, C \geq c$) and ($M \geq m, C = c$) always have solutions, while instance ($M = m, C \geq c$) has a solution if and only if ($M = m, C = c$), the original problem, has a solution.

Parameter : $M = 3, C \geq 3$
Shared : $left, right, cl, cr, cb, ml, mr, mb, maxb : int$

$left \longrightarrow left : cl > 0, cb + 1 < mb, cb + mb < maxb, cb' = cb + 1, cl' = cl - 1$
$left \longrightarrow left : ml > 0, cb < mb + 1, cb + mb < maxb, mb' = mb + 1, ml' = ml - 1$
$left \longrightarrow right : cr + cb <= mr + mb, cl <= ml, cr' = cr + cb, mr' = mr + mb, cb' = 0, mb' = 0$
$right \longrightarrow right : cr > 0, cb + 1 < mb, cb + mb < maxb, cb' = cb + 1, cr' = cr - 1$
$right \longrightarrow right : mr > 0, cb < mb + 1, cb + mb < maxb, mb' = mb + 1, mr' = mr - 1$
$right \longrightarrow left : cl + cb <= ml + mb, cr <= mr, cl' = cl + cb, ml' = ml + mb, cb' = 0, mb' = 0$

Initial state : $ml = M, cl = C, mb = 0, cb = 0, mr = 0, cr = 0, maxb = 3$
Accepting state : $mr = M, cr = C$

Fig. 15. A Model for the Missionaries/Cannibals Problem for $M = 3, C \geq 3$.

1), $(M \geq 1, C = 1)$, $(M = 3, C \geq 3)$ and $(M \geq 3, C = 3)$ in a reasonable time, as shown in Table 2 and Table 3. It is observed that the refining procedure takes more time as parameters m, c increase. We tested the four instances on the ALV tool [14], Interproc [26] and FASTer [11]. [3] These tools gave outputs similar to those stated in Table 2 and none of the instances were verified.

Model	Time	#ref	#cons
$M = 1, C \geq 1$	0.1 sec	3	86
$M \geq 1, C = 1$	0.2 sec	3	46
$M = 3, C \geq 3$	45 min 59 sec	12	3871
$M \geq 3, C = 3$	77 min 8 sec	14	18094

Table 3. Running Times of the Missionaries/Cannibals Model, with different m and c. Each result is obtained on an Intel Xeon 2.66GHz processor with 8GB memory, using the CLP-prover [31] as the interpolant solver. *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. It can be observed that verification is more difficult for our approach as m and c get larger.

A.7 The Swimming Pool Protocol

The swimming pool protocol is a Petri net with 6 transitions and 6 variables studied in [25]. It is proved by hand in [18] and verified automatically in [11] that, for two non-negative parameters Q_1, Q_2 and initial values $x_1 = x_2 = x_3 =$

[3] For FASTer, we tested our examples with libraries Mona, Lash and Omega, and applied forward and backward search strategies in turn. For the other tools, we just used the default settings.

Parameter : $Q_1 \geq 0, Q_2 \geq 0$
Shared : $left, right, cl, cr, cb, ml, mr, mb, maxb, k : int$

$init \longrightarrow init : x_6 > 0, x_1' = x_1 + 1, x_6' = x_6 - 1, k' = k - 1$
$init \longrightarrow init : x_1 > 0, x_7 > 0, x_1' = x_1 - 1, x_7' = x_7 - 1, x_2' = x_2 + 1, k' = k - 1$
$init \longrightarrow init : x_2 > 0, x_2' = x_2 - 1, x_3' = x_3 + 1, x_6' = x_6 + 1, k' = k - 1$
$init \longrightarrow init : x_3 > 0, x_6 > 0, x_3' = x_3 - 1, x_6' = x_6 - 1, x_4' = x_4 + 1, k' = k - 1$
$init \longrightarrow init : x_4 > 0, x_4' = x_4 - 1, x_5' = x_5 + 1, x_7' = x_7 + 1, k' = k - 1$
$init \longrightarrow init : x_5 > 0, x_5' = x_5 - 1, x_6' = x_6 + 1, k' = k - 1$

Initial state : $x_1 = 0, x_2 = C, x_3 = 0, x_4 = 0, x_5 = 0, x_6 = Q_1, x_7 = Q_2$
Bad state : $x_2 = x_4 = x_5 = x_6 = x_7 = 0$

Fig. 16. Modeling a variant of the swimming pool protocol.

$x_4 = x_5 = 0$, $x_6 = Q_1$, $x_7 = Q_2$, the protocol has a deadlock regardless of the values of Q_1 and Q_2. We take a benchmark from [11] that models this protocol, which can be represented using the parameterized system in Fig. 16. We verify a weaker property that there *exist* some $Q_1 \geq 1, Q_2 \geq 1$ such that a *certain* deadlock $x_2 = x_4 = x_5 = x_6 = x_7 = 0$ occurs. Monotonic abstraction cannot verify this property and returns a spurious counterexample. On the other hand, our method automatically checks the property in a reasonable time. As shown in Table 1, the CEGAR method automatically checks the property for $Q_1 \geq 1$ and $Q_2 \geq 1$ after 2 steps of refinement for a total execution time of 0.2 seconds. Table 4 shows that the time needed for verification grow very fast as the values of Q_1 and Q_2 increase. We tested this model for $Q_1 = 1, Q_2 = 1$ on the ALV tool [14], Interproc [26] and FASTer [11]. These tools failed to verify the property and gave outputs as stated in Table 2.

Model	**Time**	**#ref**	**#cons**
$Q_1 \geq 1, Q_2 \geq 1$	0.2 sec	2	55
$Q_1 \geq 2, Q_2 \geq 2$	12 min 49 sec	14	35220
$Q_1 \geq 3, Q_2 \geq 3$	$>$ 100 min	-	-

Table 4. Running Times of the Variant Swimming Pool Protocols, with different Q_1 and Q_2. Each result is obtained on an Intel Xeon 2.66GHz processor with 8GB memory. *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. It can be observed that verification is more difficult for our approach as Q_1 and Q_2 get larger.

B Proof of Lemmas

Proof (Lemma 3). For any infinite sequence $X = \langle x_1, x_2, \cdots \rangle$, let S_X be a subsequence of X obtained by eliminating elements not in S. If $|S_X| = \infty$, note that $\preceq_S$ is a WQO on S_X, due to the fact that for $x, y \in S$, $y \preceq_S x$ iff $y \preceq x$, and the fact that $\preceq$ is a WQO on S_X. If $|S_X| < \infty$, note that $\preceq_S$ is a WQO on X/S_X, due to the fact that for $x, y \notin S$, $y \preceq_S x$ iff $y \preceq x$, and the fact that $\preceq$ is a WQO on X/S_X. In both cases, there exist $i < j$ and $x_i, x_j \in X$ such that $x_i \preceq x_j$. Therefore $\preceq_S$ is a WQO on C.