Computer Sciences Department

Verifying Concurrent Message-Passing C Programs with Recursive Calls

Sagar Chaki
Edmund Clarke
Nicholas Kidd
Thomas Reps
Tayssir Touili

Technical Report #1532

September 2005

UNIVERSITY OF WISCONSIN MADISON

Verifying Concurrent Message-Passing C Programs with Recursive Calls *

Sagar Chaki Edmund Clarke
Carnegie Mellon University, Pittsburgh, USA

Nicholas Kidd Thomas Reps
University of Wisconsin, Madison, USA

Tayssir Touili
LIAFA, CNRS & University of Paris 7, Paris, France

Abstract

We consider the model-checking problem for C programs with (1) data ranging over very large domains, (2) (recursive) procedure calls, and (3) concurrent parallel components that communicate via synchronizing actions. We model such programs using *communicating pushdown systems*, and reduce the reachability problem for this model to deciding the emptiness of the intersection of two context-free languages L_1 and L_2. We tackle this undecidable problem using a CounterExample Guided Abstraction Refinement (CEGAR) scheme based on (1) computing over-approximations A_1 and A_2 of L_1 and L_2, (2) checking if the intersection of A_1 and A_2 is non-empty, and, if the non-empty intersection represents an infeasible trace, (3) refining these over-approximations A_1 and A_2. Furthermore, we present new *fully automatic* predicate-abstraction refinement techniques to obtain communicating pushdown systems from C source code. We have implemented our techniques in the model-checker MAGIC. We report our experimental results on some non-trivial benchmarks.

1. Introduction

Analysis of concurrent software represents a major challenge in the model-checking community. Indeed, concurrent programs include various complex features such as: (1) the manipulation of data ranging over unbounded domains, such as integers and reals (or very large domains, such as 32-bit ints and floats), (2) the presence of recursive procedure calls, which can lead to an unbounded number of calls, (3) concurrency and the existence of synchronization statements. Unfortunately, checking whether a given control point is reachable is undecidable, even if the program includes only recursive procedures and synchronisation statements [Ram00]. Consequently, any method for solving the reachability problem for these systems is incomplete, and all we can hope for is either an approximate technique, or a semi-decision procedure whose termination is not guaranteed.

During the last few years, several authors have addressed this issue. *Pushown systems* have been proposed as an adequate formalism to describe *pure sequential recursive programs* [EK99, ES01]. This allows to represent the potentially infinite configurations of recursive programs in a symbolic manner using regular languages [BEM97, FWW97]. Recently, compositions of *pushown systems*, called *communicating pushown systems*, have been used to model *concurrent recursive programs* [BET03a, BET03b]. However, in these cases, all data were assumed to have a *small* finite domain.

On the other hand, abstract-interpretation techniques [CC77] have been used to deal with data ranging over unbounded (or very large) domains. More recently, automated *predicate-abstraction* techniques [GS97] have been proposed to deal with this issue. The idea of predicate abstraction is to abstract the infinite data domain into a finite one defined by a given set of predicates. The precision of the abstraction and the model-checking algorithm depend on the number and the form of the predicates, because the size of the model increases with the number of predicates. The central problem in predicate abstraction is then the discovery of a *small* set of predicates sufficient to prove the desired property. To do so, *CounterExample Guided Abstaction Refinement* (CEGAR) techniques [Kur94, CGJ+00] have been used to find such a small set. The idea is to (1) start with an empty set of predicates, (2) perform the verification procedure on the obtained model. If the property is satisfied by the model, we conclude that it is also satisfied by the real program because the program has fewer behaviors than the model. Otherwise, we obtain a counterexample. (3) If the counterexample corresponds to an execution of the program, we conclude that the program does not satisfy the property. (4) Otherwise, we compute a new set of predicates that eliminate future exploration of the spurious trace, and go back to step (2).

This schema has been successfully applied to handle both pure *non-concurrent* (sequential) recursive programs in the tool SLAM [BR01], and concurrent *nonrecursive* programs in the tools BLAST [HJMS02] and MAGIC [CCG+03].

In this work, we go one step further, and combine CEGAR predicate-abstraction techniques with pushdown-system modeling to handle *concurrency, recursion,* **and** *very large data domains* at the same time. Our approach consists of using communicating pushdown systems (CPDSs) to model concurrent programs. To do this, we (1) define CEGAR predicate-abstraction techniques to obtain successively more precise CPDSs from the C source code of a parallel program, and (2) define model-checking algorithms for CPDSs. The main contributions of this paper are:

1. Defining new *automatic* CEGAR predicate-abstraction techniques that can create a CPDS from the C source code of a concurrent (recursive) C program that manipulates variables that

* This research was sponsored by the Office of Naval Research (ONR) and the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ONR, NRL, the U.S. Government or any other entity.

range over very large domains, and that can refine CPDS abstractions to eliminate a given counterexample. Our techniques are defined *component-wise*, which makes them *compositional* and *scalable to large programs*.

2. Defining new model-checking techniques for CPDSs. We restrict ourselves in this work to solving reachability queries. We reduce the reachability problem for CPDSs to the undecidable problem of checking the emptiness of the intersection of two context-free languages L_1 and L_2. To tackle this problem, we apply a second CEGAR scheme that consists of (1) computing over-approximations A_1 and A_2 of L_1 and L_2. (2) If $A_1 \cap A_2 = \emptyset$, we conclude that $L_1 \cap L_2 = \emptyset$. (3) Otherwise, we check whether the intersection $A_1 \cap A_2$ is spurious. In this case, we refine the over-approximations A_1 and A_2, and return to step (2). This semi-decision procedure is guaranteed to terminate if the intersection $L_1 \cap L_2$ is not empty.
3. Implementing our technique in the model-checker MAGIC, and carrying out a number of non-trivial experiments. Our implementation was able to handle a non-trivial example (a Bluetooth driver in Windows NT) that could not be handled with the previous version of MAGIC. Moreover, the implementation provides improved performance for non-recursive examples that the previous version of MAGIC was able to handle *only via inlining*. This shows that our technique represents an advance for recursive *as well as* non-recursive concurrent programs.

One of the novel features of this work is that it applies the CEGAR scheme at two levels: (1) at the predicate-abstraction level to deal with unbounded domain variables, and (2) at the model-checking level to solve reachability queries in CPDSs: the CPDS model checker uses a second CEGAR scheme in its semi-decision procedure for testing emptiness of the intersection of two context-free languages. As far as we know, this is the first time that CEGAR is used in the model-checker itself. Indeed, it is usually used to compute successively more precise abstractions of a system.

Related Work. In [BET03a, BET03b], the reachability problem for CPDSs has also been reduced to computing over-approximations of context-free languages. However, no CEGAR techniques were presented there. More precisely, those works compute over-approximations A_1 and A_2 of two given context-free languages L_1 and L_2, and if $A_1 \cap A_2 = \emptyset$, one concludes that $L_1 \cap L_2 = \emptyset$. However, with the approach of [BET03a, BET03b], no conclusion can be made automatically if $A_1 \cap A_2 \neq \emptyset$. In particular, using [BET03a, BET03b], one can never conclude that $L_1 \cap L_2 \neq \emptyset$. In contrast, our CEGAR-based semi-decision procedure is guaranteed to terminate in this case, with the correct answer.

CEGAR-based predicate-abstraction techniques are used in several C-programs model-checking tools, such as SLAM [BR01], BLAST [HJMS02], ZING [QRR04], and KISS [QW04]. However, as mentioned previously, SLAM cannot deal with concurrency, BLAST cannot handle recursion, and KISS cannot discover errors that appear after a number of interleavings between the parallel components greater than three. ZING is an extension of SLAM to concurrent programs. SLAM and ZING are based on procedure summarization; hence, ZING might not terminate in cases where our technique will. Indeed, in the concurrent case, one needs to keep track of the calling stack, which can be unbounded in the presence of recursive calls. The contents of the stack are explicitely represented in ZING. In contrast, in our framework, they are symbolically represented with regular languages, because we use pushdown system modeling. On the other hand, SLAM and ZING use predicate-abstraction techniques to extract a *Boolean program* from a C program with recursion. Schwoon [Sch02] has implemented in his pushdown-systems-analysis tool MOPED a translation from Boolean programs to pushdown systems. However, MOPED cannot handle concurrent programs. Our CPDS predicate-abstraction-refinement techniques are done component-wise, and amount to performing successive sequential PDS predicate-abstractions and refinements. One can argue that these successive steps can be done using SLAM and then MOPED. However, in this paper, we propose predicate-abstraction techniques that produce *directly* and *more efficently* a pushdown system from C source code of a sequential component without going through a Boolean program. We give in Section 3.4 more details about the difference between our predicate-abstraction techniques and the ones used in SLAM and ZING. Readers who are already familiar with those techniques, and who wish to skip our approach to translating C code to PDSs should concentrate on Sections 2, 4, and 6, which focus on the concurrency-related aspects of our work.

Finally, the new techniques presented in [KIG05, QR05] also use multiple pushdown systems to model concurrent recursive programs. However, [KIG05] is restricted to programs that communicate via a finite number of locks, and assumes a certain nesting condition on the locks. As for [QR05], it uses shared-variables for communication between threads, whereas we use synchronizing actions (these two models can simulate each other). The technique presented in [QR05] sidesteps the undecidability of reachability of multiple pushdown systems by putting a bound k on the number of interleavings between the different threads, whereas we handle this undecidable problem by computing abstractions of context-free languages (without bounding the number of interleavings between the different threads). In certain cases, our technique can be more powerful than the one presented in [QR05]. Indeed, if the target configurations are reachable, our technique is guaranteed to terminate with the correct answer. The same can be said of the technique of [QR05] if we apply it by incrementing automatically the bound k until the target configurations are found to be reachable. However, in certain circumstances (when we find $A_1 \cap A_2 = \emptyset$) we can infer that the target configurations are not reachable, whereas the technique of [QR05] can never establish such a property. Finally, the technique of [QR05] has not been implemented, and no automatic techniques to translate C code to pushdown systems are provided there. In contrast, our method has been implemented and applied to several non-trivial examples. This effort is reported in Section 6.

The remainder of the paper is organized as follows: In Section 2, we define the CPDS model. Section 3 describes the way we generate a CPDS from a C program using predicate abstraction. In Section 4, we give our semi-decision procedure for model-checking a CPDS. Section 5 presents our predicate-abstraction refinement techniques. Section 6 reports our experimental results.

2. Preliminary definitions

A *pushdown system* (PDS) is a four-tuple $\mathcal{P} = (Q, Act, \Gamma, \Delta)$ where P is a finite set of *states*, Act is a finite set of *actions*, Γ is a finite *stack alphabet*, and Δ is a finite set of *transition rules* of the form $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', w \rangle$, where $p, p' \in P, a \in Act, \gamma \in \Gamma$, and $w \in \Gamma^*$. We assume without loss of generality that all the rules of Δ are such that $|w| \leq 2$. This is not restrictive because any PDS can be transformed into a PDS of this form [Sch02]. Moreover, as we will see in the next subsection, the transition rules obtained from programs are always of this form. A *configuration* of $\mathcal{P}$ is a pair $\langle p, w \rangle$, where $p \in P$ is a state and $w \in \Gamma^*$ is *the content of the stack*. A set C of configurations is *regular* if for every $p \in P$ the language $\{w \in \Gamma^* \mid \langle p, w \rangle \in C\}$ is regular.

For every $a \in Act$, we define a transition relation $\overset{a}{\longrightarrow}$ between the configurations of $\mathcal{P}$ as follows:

$$\langle q, \gamma \rangle \overset{a}{\hookrightarrow} \langle q', w \rangle, \text{ then } \langle q, \gamma v \rangle \overset{a}{\longrightarrow} \langle q', wv \rangle \text{ for every } v \in \Gamma^*$$

For $a_1 \cdots a_n \in Act^*$, the relation $\xrightarrow{a_1 \cdots a_n}$ is defined in the obvious way. Let C be a set of configurations. $Post^*(C)$ is the set of successors of C defined as follows:

$$Post^*(C) = \{c' \mid \exists c \in C, a_1 \cdots a_n \in Act^*, c \xrightarrow{a_1 \cdots a_n} c'\}$$

A *communicating pushdown system* (CPDS) [BET03b] is a tuple $CP = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ of pushdown systems over the same set of actions Act such that $Act = Lab \cup \{\tau\}$, where Lab is the set of synchronization actions, and τ represents internal actions. τ is such that for every $a \in Lab$, $\tau a = a\tau = a$. As we will see later, we need this to reduce the reachability problem for CPDSs to checking the emptiness of the intersection of two context-free languages.

A *global configuration* of CP is a tuple $g = (c_1, \ldots, c_n)$ of configurations of $\mathcal{P}_1, \ldots \mathcal{P}_n$. The relation $\xrightarrow{a}$ is extended to global configurations as follows:

- $(c_1, \ldots, c_n) \xrightarrow{\tau} (c'_1, \ldots, c'_n)$ if there is an index $1 \leq i \leq n$ such that $c_i \xrightarrow{\tau} c'_i$ and $c'_j = c_j$ for every $j \neq i$;
- $(c_1, \ldots, c_n) \xrightarrow{a} (c'_1, \ldots, c'_n)$ if there are two distinct indices $i \neq j$ such that $c_i \xrightarrow{a} c'_i$, $c_j \xrightarrow{a} c'_j$, and $c'_k = c_k$ for every $i \neq k \neq j$.

Given a set G of global configurations, we define the successors of G, $Post^*(G)$, as before.

3. Component-wise Predicate Abstraction

We model concurrent recursive programs using CPDSs. In this section, we show how to use predicate abstraction to extract a CPDS from a parallel program.

Suppose that we are given n concurrent recursive C components. We extract a PDS from each recursive component. The parallel composition of these components is then represented by the CPDS corresponding to the tuple of these PDSs. In what follows, we show how to extract a PDS from a sequential component using predicate abstraction. To do this, we extend the approach originally used in MAGIC [CCG+03], which *automatically* extracts a finite-state automaton from C code, to extract a PDS.

Without loss of generality, we assume that there are only six kinds of statements in programs: assignments, procedure calls, if-then-else branches, gotos, synchronization statements, and returns. In MAGIC, we use the CIL tool [NMW+01] to transform arbitrary C programs into the above format.

Each PDS is defined in terms of a current set of seed predicates. Initially, the set of seed predicates is empty. The predicate set is augmented using our refinement techniques (see Section 5). Each predicate represents a set of assignments of the variables of the program. Let p be a predicate over the sets of variables X and Y, where X (resp. Y) is a set of local (resp. global) variables. Then p^{loc} (resp. p^{glob}) is the "projection" of p over the local variables X (resp. global variables Y). For example, let $p = (x > 0 \ \& \ y < 8)$ be a predicate that represents the set of values $\{x > 0, y < 8\}$. If x is a local variable, and y is a global one; p^{loc} denotes the predicate $(x > 0)$; and p^{glob} the predicate $(y < 8)$. We extend these notations to sets of predicates in the obvious manner.

We first describe how, given a set of seed predicates, we generate a larger set of predicates useful to compute an abstraction of the program (this process is called *predicate inference*), and then describe how to use this new set of predicates to obtain a PDS from a sequential C component. As explained above, the CPDS that corresponds to a parallel program is the tuple of all the PDSs of its different sequential components.

3.1 Predicate inference

The weakest precondition of a set of predicates p is defined as follows. Let s be an assignment of the form $v = e$. Then, the weakest precondition of p with respect to s (denoted by $\mathcal{W}_s(p)$) is obtained from p by replacing every occurrence of v in p by e. Assignments through pointers, i.e., statements of the form $*p = e$, are handled by the approach of Morris [Mor82].

Let $\mathcal{C}$ be a set of seed predicates. In MAGIC, we require that the seed set $\mathcal{C}$ is always a subset of the conditions in the program's if statements[1]. To create a PDS that is an abstraction of a sequential component relative to the predicates in seed set $\mathcal{C}$, we repeatedly compute weakest preconditions. That is, for every control point n, we compute a set of predicates $\mathcal{P}[\mathcal{C}]_n$ as follows:

Initially, $\mathcal{P}[\mathcal{C}]_n = \emptyset$ for every point n. We repeat the following until for every n, $\mathcal{P}[\mathcal{C}]_n$ is no longer modified. Let s_n be the statement that corresponds to control point n:

1. if s_n is an assignment that has n' as successor, then add $\mathcal{W}_{s_n}(\mathcal{P}[\mathcal{C}]_{n'})$ to $\mathcal{P}[\mathcal{C}]_n$.
2. if s_n is an if statement and n' is its then or else successor, then add $\mathcal{P}[\mathcal{C}]_{n'}$ to $\mathcal{P}[\mathcal{C}]_n$. Moreover, if c is the corresponding condition of s_n such that $c \in \mathcal{C}$, then add c to $\mathcal{P}[\mathcal{C}]_n$.
3. if s_n is a goto or a synchronisation statement that has n' as successor, then add $\mathcal{P}[\mathcal{C}]_{n'}$ to $\mathcal{P}[\mathcal{C}]_n$.
4. if s_n is a call to a procedure π, where s_n has n' as successor, and if e_π is the initial control point of procedure π, then add $\mathcal{P}[\mathcal{C}]^{loc}_{n'}$ and $\mathcal{P}[\mathcal{C}]^{glob}_{e_\pi}$ to $\mathcal{P}[\mathcal{C}]_n$.

Note that this procedure might not terminate in the presence of loops and recursive procedure calls. In this case, we impose termination by bounding the number of predicates in $\mathcal{P}[\mathcal{C}]_n$, for every control point n.

Let us explain the intuition behind the items above. $\mathcal{P}[\mathcal{C}]_n$ is meant to be the set of predicates needed to characterize the values of the variables when point n is active (with respect to the predicates in $\mathcal{C}$). Let s_n be an assignment that has n' as successor. The first item adds $\mathcal{W}_{s_n}(\mathcal{P}[\mathcal{C}]_{n'})$ to $\mathcal{P}[\mathcal{C}]_n$. This is because if φ is true at n', then $\mathcal{W}_{s_n}(\varphi)$ is true at n. Thus, if we know that φ holds at n', then to avoid a loss of precision, we need to know that $\mathcal{W}_{s_n}(\varphi)$ holds at n.

Now, consider the fourth item (the others are easy to understand). Let π' be the procedure that contains control point n. Because procedure π is called at n, the global variables have the same values at n and e_π. This motivates the inclusion of $\mathcal{P}[\mathcal{C}]^{glob}_{e_\pi}$ in $\mathcal{P}[\mathcal{C}]_n$. Moreover, when the procedure π terminates, and control goes back to point n' in procedure π', the values of the local variables of the procedure π' in n' are the same as those at point n (since these values did not change during the call to π). This is why we add $\mathcal{P}[\mathcal{C}]^{loc}_{n'}$ to $\mathcal{P}[\mathcal{C}]_n$.

Note that in our procedure we reason in a backward manner to compute the $\mathcal{P}[\mathcal{C}]_n$'s. An equivalent approach would have been to use forward reasoning. In this case, we would need to compute strongest postconditions instead of weakest preconditions.

Finally, let $\mathcal{P}[\mathcal{C}] = \cup \mathcal{P}[\mathcal{C}]_n$, where the union is taken over all the control points n of the sequential component, be the set of all the generated predicates.

3.2 Predicate valuation

Recall that our goal is to compute a PDS abstraction of a sequential component. As described in the next section, the states of this PDS correspond to the different valuations of the global predicates, and a symbol of its stack will be a pair that consists of a control point

[1] A query q at a given point p can be emulated by introducing an if statement at p whose branch condition is q.

n, and a valuation of the local predicates at point n. We associate with each location n two sets of formulas $\mathcal{V}[\mathcal{C}]_n^{glob}$ and $\mathcal{V}[\mathcal{C}]_n^{loc}$, respectively, called global and local *valuations* as follows: For x in $\{glob, loc\}$, $\mathcal{V}[\mathcal{C}]_n^x$ is the set of formulas $\{(p_1^x = v_1) \wedge \cdots \wedge (p_{k_x}^x = v_{k_x}) \mid \mathcal{P}[\mathcal{C}]_n^x = \{p_1^x, \ldots, p_{k_x}^x\}, v_i \in \{\text{true,false}\}, i = 1, \ldots, k_x\}$. Moreover, if $\mathcal{P}[\mathcal{C}]_n^x = \emptyset$, then $\mathcal{V}[\mathcal{C}]_n^x = \{empty\}$.

Let V be a valuation of the form $(p_1 = v_1) \wedge \cdots \wedge (p_k = v_k)$. We denote by $\Gamma(V)$ its corresponding predicate $p_1' \wedge \cdots \wedge p_k'$, where $p_i' = p_i$ if $v_i = \text{true}$, and $p_i' = \neg p_i$ if $v_i = \text{false}$. Moreover, we let $\Gamma(empty) = \{\text{true}\}$.

3.3 Creating a PDS that corresponds to a sequential component

We are now ready to describe how to create a PDS that corresponds to a sequential component. Let $\mathcal{C}$ be a given set of seed predicates. We assign to a sequential (possibly recursive) component the PDS $\mathcal{P} = (Q, Act, \Gamma, \Delta)$ defined as follows. Q is the set of valuations that correspond to the global variables, i.e., $Q = \cup\mathcal{V}[\mathcal{C}]_n^{glob}$, where the union is taken over all the control points n of the sequential component. Act contains the action τ as well as the other synchronization actions of the program. Γ is the set of all pairs (n, v), where n is a control point of the sequential component, and v is a valuation in $\mathcal{V}[\mathcal{C}]_n^{loc}$ that corresponds to a set of valuations of the local variables at location n.

To define the rules of Δ, we need one more notion. Consider a goto statement from point n_1 to n_2. Intuitively, we would like to represent this statement with rules of the form

$$\langle glob, (n_1, loc)\rangle \xhookrightarrow{\tau} \langle glob', (n_2, loc')\rangle$$

where $glob \in \mathcal{V}[\mathcal{C}]_{n_1}^{glob}$, $glob' \in \mathcal{V}[\mathcal{C}]_{n_2}^{glob}$, $loc \in \mathcal{V}[\mathcal{C}]_{n_1}^{loc}$, $loc' \in \mathcal{V}[\mathcal{C}]_{n_2}^{loc}$, and the formulas $\big(\Gamma(glob) \wedge \Gamma(glob')\big)$ and $\big(\Gamma(loc) \wedge \Gamma(loc')\big)$ are satisfiable.

This means that if the program is at point n_1 and its variables satisfy the global and local valuations $glob$ and loc, then after performing the goto statement, it goes to point n_2 and its variables can satisfy all the valuations $glob' \in \mathcal{V}[\mathcal{C}]_{n_2}^{glob}$ and $loc' \in \mathcal{V}[\mathcal{C}]_{n_2}^{loc}$ such that $\big(\Gamma(glob) \wedge \Gamma(glob')\big)$ and $\big(\Gamma(loc) \wedge \Gamma(loc')\big)$ are satisfiable. This condition ensures that the PDS we are creating has more behaviors than the concrete program.

However, determining whether $(p_1 \wedge p_2)$ is satisfiable is in general undecidable when p_1 and p_2 are first-order formulas over the integers. To sidestep this problem, we use a sound validity checker [Nel80] that always terminates and answers TRUE, FALSE, or UNKOWN to the question whether a given formula $\neg(p_1 \wedge p_2)$ is valid. We use $\mathcal{A}(p_1, p_2)$ to denote that the answer provided by the validity checker to the question "*Is* $\neg(p_1 \wedge p_2)$ *valid?*" is FALSE or UNKOWN. Then, to ensure that the PDS we are creating is a safe abstraction, we add the PDS-transition above if $\mathcal{A}(p_1, p_2)$ holds.

We are now ready to define the set of rules Δ as follows: Let s be a statement, and n_1 be its corresponding control point:

- If s is a goto statement, it is represented by rules of the form:

  $$\langle glob, (n_1, loc)\rangle \xhookrightarrow{\tau} \langle glob', (n_2, loc')\rangle$$

  where n_2 is the unique successor of n_1, $glob \in \mathcal{V}[\mathcal{C}]_{n_1}^{glob}$, $glob' \in \mathcal{V}[\mathcal{C}]_{n_2}^{glob}$, $loc \in \mathcal{V}[\mathcal{C}]_{n_1}^{loc}$, $loc' \in \mathcal{V}[\mathcal{C}]_{n_2}^{loc}$, $\mathcal{A}\big(\Gamma(glob), \Gamma(glob')\big)$, and $\mathcal{A}\big(\Gamma(loc), \Gamma(loc')\big)$.

- If s is a synchronizing statement labeled with action a, it is represented by rules of the form:

  $$\langle glob, (n_1, loc)\rangle \xhookrightarrow{a} \langle glob', (n_2, loc')\rangle$$

  where n_2 is the unique successor of n_1, $glob \in \mathcal{V}[\mathcal{C}]_{n_1}^{glob}$, $glob' \in \mathcal{V}[\mathcal{C}]_{n_2}^{glob}$, $loc \in \mathcal{V}[\mathcal{C}]_{n_1}^{loc}$, $loc' \in \mathcal{V}[\mathcal{C}]_{n_2}^{loc}$, $\mathcal{A}\big(\Gamma(glob), \Gamma(glob')\big)$, and $\mathcal{A}\big(\Gamma(loc), \Gamma(loc')\big)$.

- If s is an *assignment*, then it is translated into a set of rules of the form

  $$\langle glob, (n_1, loc)\rangle \xhookrightarrow{\tau} \langle glob', (n_2, loc')\rangle.$$

  where n_2 is the unique successor of n_1, $glob \in \mathcal{V}[\mathcal{C}]_{n_1}^{glob}$, $glob' \in \mathcal{V}[\mathcal{C}]_{n_2}^{glob}$, $loc \in \mathcal{V}[\mathcal{C}]_{n_1}^{loc}$, $loc' \in \mathcal{V}[\mathcal{C}]_{n_2}^{loc}$, $\mathcal{A}\Big(\mathcal{W}_s\big(\Gamma(glob')\big), \Gamma(glob)\Big)$, and $\mathcal{A}\Big(\mathcal{W}_s\big(\Gamma(loc')\big), \Gamma(loc)\Big)$.

  In other words, $glob$ and $glob'$ (loc and loc') are valuations that correspond to the values of the global (local) variables before and after the assignment.

  For example, if we have $\mathcal{P}[\mathcal{C}]_{n_1} = \{(y = 7), (x > -2)\}$ and $\mathcal{P}[\mathcal{C}]_{n_2} = \{(y = 7), (x > 1)\}$, where y is a global variable and x is a local one; if s is the assignment $x := x + 3$; then we have the following rule, where T stands for true: $\langle\big((y = 7) = T\big), \Big(n_1, \big((x > -2) = T\big)\Big)\rangle \xhookrightarrow{\tau} \langle\big((y = 7) = T\big), \Big(n_2, \big((x > 1) = T\big)\Big)\rangle$.

- If s is an if statement, it is represented by rules of the form:

  $$\langle glob, (n_1, loc)\rangle \xhookrightarrow{\tau} \langle glob', (n_2, loc')\rangle.$$

  where n_2 is the control point of the corresponding then (resp. else) statement if $glob$ and loc satisfy the if condition (resp. do not satisfy the if condition); and where $glob \in \mathcal{V}[\mathcal{C}]_{n_1}^{glob}$, $glob' \in \mathcal{V}[\mathcal{C}]_{n_2}^{glob}$, $loc \in \mathcal{V}[\mathcal{C}]_{n_1}^{loc}$, $loc' \in \mathcal{V}[\mathcal{C}]_{n_2}^{loc}$, $\mathcal{A}\big(\Gamma(glob), \Gamma(glob')\big)$, and $\mathcal{A}\big(\Gamma(loc), \Gamma(loc')\big)$.

- If s is a call to a procedure π, then it is represented by rules of the form:

  $$\langle glob, (n_1, loc)\rangle \xhookrightarrow{\tau} \langle glob', (e_\pi, loc')\,(n_2, loc'')\rangle$$

  where n_2 is the unique successor of n_1, e_π is the initial control point of the procedure π, $glob \in \mathcal{V}[\mathcal{C}]_{n_1}^{glob}$, $loc \in \mathcal{V}[\mathcal{C}]_{n_1}^{loc}$, $glob' \in \mathcal{V}[\mathcal{C}]_{e_\pi}^{glob}$, $loc' \in \mathcal{V}[\mathcal{C}]_{e_\pi}^{loc}$, $loc'' \in \mathcal{V}[\mathcal{C}]_{n_2}^{loc}$, $\mathcal{A}\big(\Gamma(glob), \Gamma(glob')\big)$, and $\mathcal{A}\big(\Gamma(loc), \Gamma(loc'')\big)$.

  We need to have $\mathcal{A}\big(\Gamma(loc), \Gamma(loc'')\big)$ because the role of loc'' is to save the values of the local variables of the caller procedure.

- Finally, a return statement is translated into rules of the following form, where $glob \in \mathcal{V}[\mathcal{C}]_{n_1}^{glob}$, and $loc \in \mathcal{V}[\mathcal{C}]_{n_1}^{loc}$:

  $$\langle glob, (n_1, loc)\rangle \xhookrightarrow{\tau} \langle glob, \varepsilon\rangle$$

REMARK 3.1. *Note that the predicate abstraction techniques described above are sound only for programs in which there are no assignments through pointers that can hold addresses of local variables of callers. It would not be difficult to extend these techniques with an interprocedural modification-analysis algorithm [CK88] to detect and account for such cases.*

NOTE 3.1. *Observe that all the internal actions are represented by "τ". This is needed to reduce the reachability problem for CPDSs to computing abstractions of path languages for pushdown systems, as will be discussed in Section 4.*

3.4 Comparision with the predicate-abstraction technique of SLAM

As mentioned in Section 1, the SLAM tool uses predicate-abstraction techniques to extract a *Boolean program* from C source code. Then, one can use Schwoon's translation [Sch02] to obtain a

PDS from a Boolean program. Compared with the techniques used in SLAM, the techniques described in Section 3.3 exhibit two main differences:

1. Our translation is more efficient because it produces *directly, in one step*, a PDS from C code without going through an intermediate Boolean program.
2. Technically, the approach described in Section 3.3 is different from SLAM's approach. In our method, we close a given set of seed predicates $\mathcal{C}$ by computing weakest preconditions along the different possible paths of the program, and thus obtain a larger set of predicates that we use to compute the abstract model. In contrast, SLAM uses the seed set of predicates $\mathcal{C}$ as is, without computing its closure by weakest precondition. Instead, it computes largest disjunctions of predicates in $\mathcal{C}$ that imply the weakest preconditions. Consequently, the abstract model we obtain is more precise than SLAM's because it uses more predicates.

3.5 Example

Consider the following two sequential components D_1 and D_2 running in parallel, where a is a synchronization action:

D_1:
main() {
n_0: int x=10;
n_1: proc();
n_2: return;}

void proc(){
n_3: if $(x < 10)$
n_4: {a;}
n_5: else {proc();}
n_6: return;}

D_2:
main(){
m_0: a;
m_1: return;}

Case #1: The set of seed predicates $\mathcal{C}$ is empty: Let us model first the component D_1 by a PDS $\mathcal{P}_1$. There are no local variables, so the stack alphabet is the set of the control points. Moreover, because the set of seed predicates $\mathcal{C}$ is empty, let p be the unique state of $\mathcal{P}_1$ (p corresponds to the valuation *empty*). $\mathcal{P}_1$ contains the following rules:
$r_1 : \langle p, n_0\rangle \overset{\tau}{\hookrightarrow} \langle p, n_1\rangle$; $r_2 : \langle p, n_1\rangle \overset{\tau}{\hookrightarrow} \langle p, n_3 n_2\rangle$; $r_3 : \langle p, n_2\rangle \overset{\tau}{\hookrightarrow} \langle p, \epsilon\rangle$; $r_4 : \langle p, n_3\rangle \overset{\tau}{\hookrightarrow} \langle p, n_4\rangle$; $r_5 : \langle p, n_3\rangle \overset{\tau}{\hookrightarrow} \langle p, n_5\rangle$; $r_6 : \langle p, n_4\rangle \overset{a}{\hookrightarrow} \langle p, n_6\rangle$; $r_7 : \langle p, n_5\rangle \overset{\tau}{\hookrightarrow} \langle p, n_3 n_6\rangle$; $r_8 : \langle p, n_6\rangle \overset{\tau}{\hookrightarrow} \langle p, \epsilon\rangle$.

Similarly, we represent the second component by a PDS $\mathcal{P}_2$ that has a unique state q, and the following rules:
$r'_1 : \langle q, m_0\rangle \overset{a}{\hookrightarrow} \langle q, m_1\rangle$; and $r'_2 : \langle q, m_1\rangle \overset{\tau}{\hookrightarrow} \langle q, \epsilon\rangle$.

Case #2: We have $\mathcal{C} = \{(x < 10)\}$: We model the component D_1 by the following PDS $\mathcal{P}'_1$. We have: $P_{n_1} = P_{n_3} = P_{n_5} = \{x < 10\}$, and $P_n = \emptyset$ for the other points (while computing P_{n_0}, we find the predicate $10 < 10$. Because we ignore predicates that are trivially true or false, we keep $P_{n_0} = \emptyset$). The states of $\mathcal{P}'_1$ are: $p_1 : (x < 10) = false$, $p_2 : (x < 10) = true$, and $p_3 : empty$. $\mathcal{P}'_1$ contains the following rules:
$\langle p_3, n_0\rangle \overset{\tau}{\hookrightarrow} \langle p_1, n_1\rangle$; $\langle p_1, n_1\rangle \overset{\tau}{\hookrightarrow} \langle p_1, n_3 n_2\rangle$; $\langle p_3, n_2\rangle \overset{\tau}{\hookrightarrow} \langle p_3, \epsilon\rangle$; $\langle p_2, n_3\rangle \overset{\tau}{\hookrightarrow} \langle p_3, n_4\rangle$; $\langle p_1, n_3\rangle \overset{\tau}{\hookrightarrow} \langle p_1, n_5\rangle$; $\langle p_3, n_4\rangle \overset{a}{\hookrightarrow} \langle p_3, n_6\rangle$; $\langle p_1, n_5\rangle \overset{\tau}{\hookrightarrow} \langle p_1, n_3 n_6\rangle$; $\langle p_3, n_6\rangle \overset{\tau}{\hookrightarrow} \langle p_3, \epsilon\rangle$.

4. Reachability Analysis of CPDSs

Suppose that the program consists of n sequential components. In MAGIC, we usually ask the following query: "Suppose that the system starts from a configuration where each component i is at its initial control point n_0^i, for $i = 1, \ldots, n$; can one component reach an error point?" We show in this section how to tackle the reachability analysis of these systems. In the remainder of this paper, we restrict ourselves to systems that consist of two parallel sequential components. The technique can be extended in a straightforward manner to the general case (see [BET03b] for more details); the implementation reported in Section 6 supports an arbitrary number of components.

We reduce the reachability problem for CPDSs to deciding the emptiness question for the intersection of two context-free languages as follows: Let $(\mathcal{P}_1, \mathcal{P}_2)$ be a CPDS, and let $C_1 \times C_2$ and $C'_1 \times C'_2$ be two sets of global configurations of the system. Because all the internal actions are represented by "τ" (which is a neutral element for concatenation), $C'_1 \times C'_2$ is reachable from $C_1 \times C_2$ if and only if there exists at least one sequence of synchronization actions that simultaneously leads $\mathcal{P}_1$ from a configuration in C_1 to a configuration in C'_1 and $\mathcal{P}_2$ from a configuration in C_2 to a configuration in C'_2. This holds iff $L(C_1, C'_1) \cap L(C_2, C'_2) \neq \emptyset$, where $L(C_i, C'_i)$ is the context-free language consisting of all the sequences of actions (or, equivalently, of synchronization actions because the internal actions are represented by τ) that lead $\mathcal{P}_i$ from C_i to C'_i.

Because deciding the emptiness of two context-free languages is undecidable, we propose a semi-decision procedure that, in case of termination, answers *exactly* whether the intersection is empty or not. Moreover, if $L(C_1, C'_1) \cap L(C_2, C'_2) \neq \emptyset$, the semi-decision procedure is *guaranteed to terminate* and return a sequence in the intersection.

The semi-decision procedure is based on a CounterExample Guided Abstraction Refinement (CEGAR) scheme as follows:

1. **Abstraction:** We compute an over-approximation A_i of the path language $L(C_i, C'_i)$.
2. **Verification:** We check if $A_1 \cap A_2 = \emptyset$, and, if so, we conclude that $L(C_1, C'_1) \cap L(C_2, C'_2) = \emptyset$, i.e., that $C'_1 \times C'_2$ is unreachable from $C_1 \times C_2$. Otherwise, we compute the "counterexample" $I = A_1 \cap A_2$.
3. **Counterexample Validation:** We check whether I contains a sequence x that is in $L(C_1, C'_1) \cap L(C_2, C'_2)$. In this case I is not spurious, and we conclude that $L(C_1, C'_1) \cap L(C_2, C'_2) \neq \emptyset$, i.e., that $C'_1 \times C'_2$ is reachable from $C_1 \times C_2$. Otherwise, we proceed to the next step.
4. **Refinement:** If I is spurious, we refine the over-approximations A_1 and A_2, i.e., we compute other over-approximations A'_1 and A'_2 such that $L(C_i, C'_i) \subseteq A'_i \subseteq A_i$. We then continue from step 2.

In the remainder of this section, we discuss these steps in detail. We fix two sets of global configurations $C_1 \times C_2$ and $C'_1 \times C'_2$. For the sake of simplicity, we denote $L(C_1, C'_1)$ by L_1, and $L(C_2, C'_2)$ by L_2.

4.1 Computing over-approximations of path languages

To compute over-approximations of pushdown-system path languages, our technique is based on the approach presented in [BET03b]. We summarize this approach in what follows.

Consider an abstract lattice $(D, \leq, \sqcap, \sqcup, \bot, \top)$ associated with an idempotent semiring $(D, \oplus, \odot, \bar{0}, \bar{1})$ such that $\oplus = \sqcup$ is an associative, commutative, and idempotent ($a \oplus a = a$) operation; $\odot$ is an associative operation; $\bar{0} = \bot$; $\bar{0}$ and $\bar{1}$ are neutral elements for

$\oplus$ and $\odot$, respectively; $\bar{0}$ is an annihilator for $\odot$ ($a \odot \bar{0} = \bar{0} \odot a = \bar{0}$); and $\odot$ distributes over $\oplus$. Finally, $\leq$ is such that $x \leq x \oplus a$.

D is related to the concrete domain 2^{Lab^*} as follows:

- It contains an element v_a for every letter $a \in Lab$,
- It is associated with an abstraction function $\alpha : 2^{Lab^*} \rightarrow D$ and a concretization function $\gamma : D \rightarrow 2^{Lab^*}$ defined as follows:

$$\alpha(L) = \bigoplus_{a_1 \cdots a_n \in L} v_{a_1} \odot \cdots \odot v_{a_n}$$

and

$$\gamma(x) = \{a_1 \cdots a_n \in Lab^* \mid v_{a_1} \odot \cdots \odot v_{a_n} \leq x\}$$

It is easy to see that for every language $L \subseteq Lab^*$; $\alpha(L) \in D$, and $\gamma\big(\alpha(L)\big) \supseteq L$. In other words, $\gamma\big(\alpha(L)\big)$ is an over-approximation of L that is finitely represented in the abstract domain D by the element $\alpha(L)$. Intuitively, the abstract operations $\odot$ and $\oplus$ correspond to concatenation and union, respectively; $\leq$ and $\sqcap$ correspond to inclusion and intersection, respectively; and the abstract elements $\bar{0}$ and $\bar{1}$ correspond to the empty language and $\{\epsilon\}$, respectively.

Therefore, to compute the over-approximations $\gamma\big(\alpha(L_i)\big)$, we need to compute its representative $\alpha(L_i)$ in the abstract domain D. Let a *finite-chain abstraction* be an abstraction such that D does not contain an infinite ascending chain, and let h be the maximal height of a chain in D. Then we have:

THEOREM 4.1. *[BET03b, RSJ03] Let $\mathcal{P} = (Q, Act, \Gamma, \Delta)$ be a PDS and C, C' be two regular sets of configurations of $\mathcal{P}$, and let α be a finite-chain abstraction defined on the abstract domain D. Then $\alpha\big(L(C, C')\big)$ can be effectively computed in D in $\mathcal{O}(h|\Delta||Q|^2)$ time.*

There are two different algorithms that provide the basis of this theorem, one described in [BET03a, BET03b], and one described in [RSJ03, RSJM]. The latter has been implemented in a tool called WPDS++ [KRML]. We use this tool to compute abstractions of path languages.

To check the emptiness of the intersection of the over-approximations $\gamma\big(\alpha(L_1)\big)$ and $\gamma\big(\alpha(L_2)\big)$, it suffices to check whether $\alpha(L_1) \sqcap \alpha(L_2) = \bot$. Indeed, using the fact that $\alpha(\emptyset) = \bot$ and $\gamma(\bot) = \emptyset$, we can show that

$$\forall L_1, L_2 \in Lab^*, \alpha(L_1) \sqcap \alpha(L_2) = \bot \Leftrightarrow \gamma\big(\alpha(L_1)\big) \cap \gamma\big(\alpha(L_2)\big) = \emptyset$$

4.2 Defining refinable finite-chain abstractions

To be able to apply our CEGAR scheme, we need to define refinable finite-chain abstractions, i.e., a series $(\alpha_i)_{i \geq 1}$ such that α_i is more precise than α_j if $i > j$; i.e., for every language $L \subseteq Lab^*$, if $i > j$ then

$$L \subseteq \gamma_i\big(\alpha_i(L)\big) \subseteq \gamma_j\big(\alpha_j(L)\big)$$

For this we define the *i^{th}-prefix abstraction* as follows:

- Let W_i be the set of words of Lab^* of length less than or equal to i. The abstract lattice D_i is equal to 2^{W_i};
- for every $a \in Lab$, $v_a = a$;
- $\oplus = \cup$;
- $\sqcap = \cap$;
- $U \odot V = \{(uv)_i \mid u \in U, v \in V\}$, where $(w)_i$ is the prefix of w of length i;
- $\bar{0} = \emptyset$;
- $\bar{1} = \{\epsilon\}$;
- $\leq = \subseteq$.

Let α_i and γ_i be the abstraction and the concretization functions associated with this domain. It is easy to see that $\alpha_i(L)$ is the set of words of L of length less than i, union the set of prefixes of length i of L, i.e., $\alpha_i(L) = \{w \mid |w| < i \text{ and } w \in L, \text{ or } |w| = i \text{ and } \exists v \in Lab^* \text{ s.t. } wv \in L\}$. Therefore, $\gamma_i\big(\alpha_i(L)\big) = \{w \in \alpha_i(L) \mid |w| < i\} \cup \{wv \mid w \in \alpha_i(L), |w| = i, v \in Lab^*\}$.

Observe that it is possible to decide whether $\alpha_i(L_1) \cap \alpha_i(L_2) = \emptyset$ because for every $L \subseteq Lab^*$, $\alpha_i(L)$ is a finite set of words.

It is easy to see that if $i > j$, then α_i is more precise than α_j. Indeed, we have

$$L \subseteq \gamma_i\big(\alpha_i(L)\big) \subseteq \gamma_j\big(\alpha_j(L)\big).$$

We have thus defined a series of refinable finite-chain abstractions $\alpha_1, \alpha_2, \alpha_3, \ldots$.

REMARK 4.1. *The i^{th}-prefix* abstraction *is only one abstraction that can be used to instantiate the framework. Others are possible, such as the i^{th}-suffix or the i^{th}-subword abstractions (defined in an analogous way).*

4.3 Checking whether the counterexample is spurious

It remains to check whether $I = \gamma_i\big(\alpha_i(L_1)\big) \cap \gamma_i\big(\alpha_i(L_2)\big)$ contains an element x such that $x \in L_1 \cap L_2$. This amounts to deciding whether $I \cap L_1 \cap L_2 = \emptyset$. Unfortunately, this problem is undecidable because I is a regular language (because for $L \subseteq Lab^*$, $\gamma_i\big(\alpha_i(L)\big)$ is regular). To sidestep this problem, we check instead whether L_1 and L_2 have a common word of length at most i. This amounts to checking whether

$$\big(\alpha_i(L_1) \cap L_1\big) \cap \big(\alpha_i(L_2) \cap L_2\big) = \emptyset$$

This is decidable because $\alpha_i(L)$ is a finite set.

4.4 The semi-decision procedure

Summarizing the previous discussion, we obtain the following semi-decision procedure:

1. Initially, $i = 1$;
2. Compute the common words of length less than i, and the common prefixes of length i of $L(C_1, C_1')$ and $L(C_2, C_2')$: $I' = \alpha_i\big(L(C_1, C_1')\big) \cap \alpha_i\big(L(C_2, C_2')\big)$.
3. If $I' = \emptyset$, conclude that $L(C_1, C_1') \cap L(C_2, C_2') = \emptyset$, and that $C_1' \times C_2'$ is unreachable from $C_1 \times C_2$. Otherwise, determine whether or not I' is spurious: Check whether $I' \cap L(C_1, C_1') \cap L(C_2, C_2') \neq \emptyset$. If this holds, conclude that $L(C_1, C_1')$ and $L(C_2, C_2')$ have a common word of length less than or equal to i, and therefore, that $L(C_1, C_1') \cap L(C_2, C_2') \neq \emptyset$, and $C_1' \times C_2'$ is reachable from $C_1 \times C_2$.
4. Otherwise, increment i and proceed from step 2.

It is easy to see that:

THEOREM 4.2. *If $L(C_1, C_1') \cap L(C_2, C_2') \neq \emptyset$, then the above semi-decision procedure terminates with the exact solution.*

Proof: Let $x \in L(C_1, C_1') \cap L(C_2, C_2')$, and let k be the length of x. Then $x \in \alpha_k\big(L(C_1, C_1')\big) \cap \alpha_k\big(L(C_2, C_2')\big)$. □

REMARK 4.2. *It follows from Theorem 4.1 that at each step i, computing $\alpha_i(L)$ necessitates $\mathcal{O}(2^{|Lab|^i}|\Delta||Q|^2)$ time since there are at most $|Lab|^i$ words of length i, and therefore at most $2^{|Lab|^i}$ elements in D_i. This is the worst case complexity of our algorithm. However, in practice, our tool behaves well as described in Section 6.*

4.5 Example

Let $\mathcal{P}_1$ be the PDS having the following rules:
$r_1 : \langle p, n_0\rangle \overset{a}{\hookrightarrow} \langle p, n_1\rangle$; $r_2 : \langle p, n_1\rangle \overset{\tau}{\hookrightarrow} \langle p, n_0 n_2\rangle$; $r_3 : \langle p, n_2\rangle \overset{b}{\hookrightarrow} \langle p, \epsilon\rangle$; $r_4 : \langle p, n_0\rangle \overset{b}{\hookrightarrow} \langle p, \epsilon\rangle$.
And let $\mathcal{P}_2$ be the PDS having the following rules:
$r'_1 : \langle q, m_0\rangle \overset{a}{\hookrightarrow} \langle q, m_1\rangle$; $r'_2 : \langle q, m_1\rangle \overset{b}{\hookrightarrow} \langle q, m_2\rangle$; $r'_3 : \langle q, m_2\rangle \overset{\tau}{\hookrightarrow} \langle q, m_0 m_3\rangle$; $r'_4 : \langle q, m_3\rangle \overset{b}{\hookrightarrow} \langle q, \epsilon\rangle$; and $r'_5 : \langle q, m_0\rangle \overset{d}{\hookrightarrow} \langle q, \epsilon\rangle$.

It is easy to see that in $\mathcal{P}_1$, $L_1 = L\big(\langle p, n_0\rangle, \langle p, \epsilon\rangle\big) = \{a^k b b^k \mid k \geq 0\}$; and that in $\mathcal{P}_2$, $L_2 = L\big(\langle q, m_0\rangle, \langle q, \epsilon\rangle\big) = \{(ab)^k d b^k \mid k \geq 0\}$, and therefore that $L_1 \cap L_2 = \emptyset$. We use this straightforward example to illustrate our approach:

- $\alpha_1(L_1) \cap \alpha_1(L_2) = \{a\} \neq \emptyset$;
- $a \notin L_1$, therefore, we refine the abstraction and go to α_2;
- $\alpha_2(L_1) \cap \alpha_2(L_2) = \{ab\} \neq \emptyset$;
- $ab \notin L_2$, therefore, we refine the abstraction and go to α_3;
- $\alpha_3(L_1) \cap \alpha_3(L_2) = \emptyset$. Therefore, we infer that $L_1 \cap L_2 = \emptyset$.

5. Component-wise Refinement

The construction of the CPDS model from the C program involves predicate abstraction. It is parametrized by a set of predicates. The main issue in predicate abstraction is to find a small set of predicates that allows to prove a property of interest. In our case, the property in question is whether the system can reach an error from the initial configuration, where component i (where, e.g., $i = 1, 2$) is in $\langle glob_0^i, (n_0^i, loc_0^i)\rangle$ such that n_0^i is the initial control point of the component i, and $glob_0^i, loc_0^i$ are initial valuations of the global and local variables, respectively. Similarly, an error is a configuration where at least one component i is in a configuration of the form $\langle glob, (n_e^i, loc)\rangle$, where n_e^i correponds to an error point, and $glob$ and loc are arbitrary valuations of the variables. MAGIC finds this minimal set of predicates by applying a CEGAR approach as follows:

We start with a model involving an empty set of seed predicates, and perform the model-checking step described in Section 4. If the model checker answers that the error state is unreachable in the CPDS model, we are sure that this is also the case for the concrete program because the program has fewer behaviors than the model. Otherwise, if the model checker finds that the CPDS can reach an error state by performing a sequence of synchronization actions $a_1 \cdots a_n$ ($a_1 \cdots a_n \in I' \cap L(C_1, C'_1) \cap L(C_2, C'_2)$), we need to verify whether this behavior corresponds to any real executions of the program (in which case, we have shown that the program is not correct), or whether the erroneous-looking behavior has been introduced by abstraction. If this is the case, we need to refine the CPDS model. More precisely, the model checker returns two sequences of rules $r_1^1, \ldots, r_{m_1}^1$ and $r_1^2, \ldots, r_{m_2}^2$ such that the CPDS $(\mathcal{P}_1, \mathcal{P}_2)$ reaches the error state if $\mathcal{P}_i$ performs the sequence $r_1^i, \ldots, r_{m_i}^i$ (in this case, $a_1 \cdots a_n$ is the sequence of synchronization actions corresponding to these sequences of rules). We say that the sequence $r_1^i, \ldots, r_{m_i}^i$ is a counterexample for component i. To check whether this counterexample is spurious, we need to check whether component i can perform the sequence of statements that correspond to the sequence of rules $r_1^i, \ldots, r_{m_i}^i$. If either component fails to perform its corresponding sequence, we refine its corresponding PDS to eliminate the spurious sequence of rules. Observe that all these steps are done *component-wise*, which makes the technique compositional and scalable to large programs.

In this section, we first show how to check whether a sequence of rules returned by the model checker is spurious. We show then how to add new seed predicates to create a more precise CPDS model that eliminates a spurious trace.

5.1 Counterexample validation

We present in this subsection an algorithm that takes as input a counterexample given by a sequence of rules $r_1, \ldots, r_n$ of a PDS that models a sequential component, and answers whether it is spurious. Let $s_1, \ldots, s_n$ be the sequence of statements that correspond to $r_1, \ldots, r_n$. Intuitively, the algorithm simulates the different steps to determine whether the concrete component could possibly perform them. The algorithm starts from the initial point n_0, and the valuations $glob_0$ and loc_0 of the variables. Then, it applies successively the different statements $s_i, i = 1, \ldots, n$, updates the values of the variables, and checks whether the if-then-else conditions are satisfied in this sequence of instructions. More precisely, the algorithm works as follows:

- Initially $\varphi = glob_0 \wedge loc_0$,
- For $i = 1$ to n do
  - if s_i is an assignment, compute the strongest postcondition of φ with respect to s_i. For example, if s_i is the assignment $x := x + 5$, and φ is the valuation $(1 < x < 4) =$ true; the updated valuation φ is $(6 < x < 9) =$ true.
  - if s_i is an if statement with condition c, then if s_{i+1} corresponds to its then successor, $\varphi := \varphi \wedge c$. Otherwise, if s_{i+1} corresponds to its else successor, $\varphi := \varphi \wedge \neg c$.
- If φ is satisfiable, then the program can execute the sequence of statements, and the counterexample is valid; otherwise, the counterexample is spurious.

5.2 Eliminating the counterexample

If the counterexample is spurious for component i, we need to refine the PDS model $\mathcal{P}_i$ corresponding to this component by adding new seed predicates. The predicates that we add are subsets of the set of conditions of the if-then-else branches of the program. Intuitively, it works as follows: In most cases, the counterexample is spurious because in the abstract model we have not modeled an if condition with sufficient precision, and we have allowed both of its branches to be followed (at some "moment" during an abstract execution), whereas in any concrete execution run only one branch can be followed; the counterexample corresponds to a trace that takes the "wrong" branch. So, to eliminate this trace, we need to add the condition c of this if statement as a seed predicate. More precisely, let $X = \{c_1, \ldots, c_k\}$ be the set of conditions of the if statements of the program, and let $\mathcal{C}$ be the current set of seed predicates, i.e., such that $\mathcal{P}_i$ is computed as described in Section 3 using the set of predicates $\mathcal{P}[\mathcal{C}]$. We proceed as follows:

1. $i := 1$,
2. if $c_i \in \mathcal{C}$, then increment i and go to step 2,
3. $\mathcal{C}' := \mathcal{C} \cup \{c_i\}$,
4. Create the PDS $\mathcal{P}'_i$ that corresponds to the predicates $\mathcal{P}[\mathcal{C}']$ as described in Section 3.3. If the new model eliminates the counterexample, then let the new seed set be $\mathcal{C} := \mathcal{C}'$. Otherwise increment i and go to step 2.

If none of the predicates $c_1, \ldots, c_k$ succeeds in eliminating the counterexample, we try to add two predicates at each step. If we try all the possibilities, and the counterexample is still not eliminated, we try to add three predicates at each step, etc.

5.3 Example

Let us consider the parallel program given in Section 3.5. Consider this query: Can D_2 reach the point m_1 if the system starts

from (n_0, m_0)? Obviously, this is not the case, because the second component can go to m_1 only if it synchronizes with D_1 using the action a, whereas the first component can never perform a, because at n_3 we do not have $x < 10$. If we model the concurrent program using no seed predicates, i.e., if we consider the model $(\mathcal{P}_1, \mathcal{P}_2)$ described in Section 3.5, the model checker answers that $(n_6 n_2, m_1)$ is reachable with the following sequences: $r_1 r_2 r_4 r_6$ for $\mathcal{P}_1$, and r'_1 for $\mathcal{P}_2$. Using our method, we can check that $r_1 r_2 r_4 r_6$ is spurious because $\varphi = (x = 10) \wedge (x < 10)$ is not satisfiable. Therefore, we refine PDS $\mathcal{P}_1$. If we consider $\mathcal{C} = (x < 10)$, we obtain the PDS $\mathcal{P}'_1$ described in Section 3.5. Then it is easy to see that in the CPDS $(\mathcal{P}'_1, \mathcal{P}_2)$, $\mathcal{P}_2$ cannot reach m_1.

6. Experimental Results

We implemented the method described in the paper in ComFoRT [CISW05], a model checker built on top of MAGIC [CCG+03], and experimented with a set of non-trivial benchmarks. Our implementation supports two kinds of abstractions described in Section 4.2: the i^{th}-prefix and the i^{th}-suffix abstractions.

6.1 Application to concurrent recursive programs: a Windows NT Bluetooth driver

We applied our technique to a nontrivial recursive concurrent program that could not be handled with the original (non-recursive) version of MAGIC: a Windows NT Bluetooth driver. Our tool found a bug in this program (that was reported in [QW04]). The bug could be detected with the i^{th}-prefix as well as the i^{th}-suffix abstractions. Our experiments were performed on a Linux 2.4.21-27.0.1 SMP P4 3.00 Ghz machine with 2 GB memory. The results are summarized in Figure 1.

Abstraction	*Execution time(seconds)*	*Memory used (MB)*
i^{th}-prefix	84.6	667
i^{th}-suffix	36.7	375

Figure 1. Results for the Bluetooth driver

Note that the suffix abstraction is more efficient in this case. This is due to the fact that in this example, there are less possible backward paths from the erroneous configurations than forward paths from the initial configurations.

We describe in what follows the program (the source code can be found in [QW04]), its corresponding CPDS model, and we show how to apply our technique to find the error.

The driver consists of a certain number of processes running in parallel: a process STOP-D, a process COUNTER, a process STOPPING-FLAG, a process STOPPING-EVENT, and an arbitrary number of processes REQUEST (one per each request for the driver). The role of the process COUNTER is to count the number of requests that the driver receives. This number is set to 1 initially, is incremented when the driver receives a new request, and is decremented when a request exits the driver. The process STOP-D may issue a request to stop the driver at any time. It then has to wait until all the other requests have finished their work before stopping the driver. The process STOPPING-FLAG has two control points: F-S-F (FALSE-STOP-FLAG) and T-S-F (TRUE-STOP-FLAG), depending on whether the process STOP-D is trying to stop the driver or not. It is initially in state F-S-F, and moves to state T-S-F if it receives a message from STOP-D. No new thread can enter the driver if this process is in T-S-F. The process STOPPING-EVENT also has two control points: F-S-E (FALSE-STOP-EVENT) and T-S-E (TRUE-STOP-EVENT). It enters state T-S-E if the driver stops, i.e. when the number of running REQUESTs reaches 0. Finally, when a new REQUEST enters the driver, it has to increment the number stored in COUNTER, perform the work asked by the request, and then decrement the number stored in COUNTER before exiting the driver. This is done by two functions: *Increment* and *Decrement*.

Each of these processes can be modeled by a PDS as described below (all the techniques described here were *automatically* performed by our tool).

The process COUNTER: It has no global variables, so let p_0 be its unique state. The number of threads is represented in a stack. The stack alphabet is $\{0, 1\}$. Initially, the stack contains the word "10" meaning that the number of requests is zero. It can then contain any word in 1^*0. The number of 1's in the stack corresponds to the number of running requests minus 1. The increment and decrement operations are invoked by *incr* and *decr* messages from the REQUEST processes.

COUNTER is represented by the following PDS rules:

- $\langle p_0, 1\rangle \xrightarrow{incr} \langle p_0, 11\rangle$ and $\langle p_0, 0\rangle \xrightarrow{incr} \langle p_0, 10\rangle$. These rules increment the counter.
- $\langle p_0, 1\rangle \xrightarrow{decr} \langle p_0, \epsilon\rangle$. This rule decrements the counter.
- $\langle p_0, 1\rangle \xrightarrow{not-zero} \langle p_0, 1\rangle$ and $\langle p_0, 0\rangle \xrightarrow{is-zero} \langle p_0, 0\rangle$. These rules test whether the counter is 0.

The process STOPPING-FLAG: It has no global variables, so let p_1 be its unique state.

- $\langle p_1, \text{F-S-F}\rangle \xrightarrow{stop} \langle p_1, \text{T-S-F}\rangle$: The process receives a "*stop*" request from STOP-D.
- $\langle p_1, \text{T-S-F}\rangle \xrightarrow{stop'} \langle p_1, \text{T-S-F}\rangle$: The process communicates with a REQUEST process via a "*stop'*" message.
- $\langle p_1, \text{F-S-F}\rangle \xrightarrow{non-stop} \langle p_1, \text{F-S-F}\rangle$: It sends a "non-stop" request to the incoming REQUESTs.

The process STOPPING-EVENT: As with the two previous processes, this process has also no global variables, so let p_2 be its unique state.

- $\langle p_2, \text{F-S-E}\rangle \xrightarrow{stop-driver} \langle p_2, \text{T-S-E}\rangle$ and $\langle p_2, \text{T-S-E}\rangle \xrightarrow{is-stopped} \langle p_2, \text{T-S-E}\rangle$: The process uses the messages "is-stopped" and a "stop-driver" to indicate that the driver is stopped.
- $\langle p_2, \text{F-S-E}\rangle \xrightarrow{non-stopped} \langle p_2, \text{F-S-E}\rangle$: It sends a "non-stopped" message to indicate that the driver is still running.

The process STOP-D: Again, this process has no global variables, so let p_3 be its unique state.

- $\langle p_3, n_0\rangle \xrightarrow{stop} \langle p_3, e_{Decrement} \cdot n_1\rangle$: STOP-D sends a "stop" request to STOPPING-FLAG, and calls *Decrement*.
- $\langle p_3, n_1\rangle \xrightarrow{is-stopped} \langle p_3, \text{RELEASE}\rangle$. If the driver is stopped, the allocated resources are released.

This process has a copy of the function *Decrement* that will be described below.

The process REQUEST: This process has a global variable g that can be 0, 1, or -1. It is set initially to 1. Let g_0, g_1, and g_{-1} be the global states of the PDS that correspond to the cases where g is equal to 0, 1, and -1, respectively. The process REQUEST does the following:

1. It starts by calling a function *Increment*. This function will set g to -1 if the STOPPING-FLAG is set to TRUE, otherwise, it will increment the counter, and set g to 0.

2. If *Increment* returns 0, REQUEST performs the work it has to do, and then asserts that STOPPING-EVENT is in state F-S-E (i.e., that the driver is still running).
3. Afterwards, it calls a function *Decrement* that decrements the counter. If this counter has reached 0, it sends a message to inform STOPPING-EVENT that the driver is stopped because there are no more requests running.

The process REQUEST can be modeled by the following PDS rules, where $x \in \{1, 0, -1\}$:

- $\langle g_x, e\rangle \xrightarrow{\tau} \langle g_x, e_{Inc} \cdot n\rangle$. First, *Increment* is called;
- $\langle g_0, n\rangle \xrightarrow{\tau} \langle g_0, n_{Work}\rangle$. If *Increment* returns 0, REQUEST performs some work.
- $\langle g_0, n_{Work}\rangle \xrightarrow{\tau} \langle g_0, n_{End-Work}\rangle$. The work ends.
- $\langle g_0, n_{End-Work}\rangle \xrightarrow{non-stopped} \langle g_0, e_{Decrement}\rangle$. After the work is finished, we have to make sure that the driver is still running, i.e., that STOPPING-EVENT is in F-S-E.
- $\langle g_0, n_{End-Work}\rangle \xrightarrow{is-stopped} \langle g_0, ABORT\rangle$. If this is not the case, we have reached an error, and the program ABORTs.
- $\langle g_x, n\rangle \xrightarrow{\tau} \langle g_x, e_{Decrement}\rangle$. After, *Decrement* is called.

The function *Increment* is represented as follows:

- $\langle g_x, e_{Inc}\rangle \xrightarrow{stop'} \langle g_{-1}, \epsilon\rangle$. If STOPPING-FLAG is in T-S-F, the function returns -1;
- $\langle g_x, e_{Inc}\rangle \xrightarrow{non-stop} \langle g_x, n'\rangle$. Otherwise, it increments the counter, and returns:
- $\langle g_x, n'\rangle \xrightarrow{incr} \langle g_0, \epsilon\rangle$

The function *Decrement* is represented as follows, where $g \in \{g_x, p_3\}$ (this function is shared by REQUEST and STOP-D):

- $\langle g, e_{Decrement}\rangle \xrightarrow{dec} \langle g, n''\rangle$. The counter is decremented.
- $\langle g, n''\rangle \xrightarrow{not-zero} \langle g, \epsilon\rangle$. If it has not reached 0, the function terminates;
- $\langle g, n''\rangle \xrightarrow{is-zero} \langle g, n'''\rangle$. Otherwise, it communicates with STOPPING-EVENT using a message "stop-driver":
- $\langle g, n'''\rangle \xrightarrow{stop-driver} \langle g, \epsilon\rangle$

The erroneous trace: The error arises even if we have only one running request, i.e., if we have the processes STOP-D, COUNTER, STOPPING-FLAG, STOPPING-EVENT, and an instance of REQUEST running in parallel. We show that the program can reach the bad point ABORT. A configuration will be represented by a 5-tuple $(a_1, a_2, a_3, a_4, a_5)$, where a_1, a_2, a_3, a_4, and a_5 represent the configurations of the processes STOP-D, COUNTER, STOPPING-FLAG, STOPPING-EVENT, and REQUEST, respectively. The initial configuration is

$$\Big(\langle p_0, 10\rangle, \langle p_1, \text{F-S-F}\rangle, \langle p_2, \text{F-S-E}\rangle, \langle p_3, n_0\rangle, \langle g_1, e\rangle\Big)$$

With a τ action, this configuration can move to:

$$\Big(\langle p_0, 10\rangle, \langle p_1, \text{F-S-F}\rangle, \langle p_2, \text{F-S-E}\rangle, \langle p_3, n_0\rangle, \langle g_1, e_{Inc} \cdot n\rangle\Big)$$

and then by exchanging a "non-stop" message between REQUEST and STOPPING-FLAG to:

$$\Big(\langle p_0, 10\rangle, \langle p_1, \text{F-S-F}\rangle, \langle p_2, \text{F-S-E}\rangle, \langle p_3, n_0\rangle, \langle g_1, n' \cdot n\rangle\Big)$$

Now, STOP-D can send a "stop" request to STOPPING-FLAG:

$$\Big(\langle p_0, 10\rangle, \langle p_1, \text{T-S-F}\rangle, \langle p_2, \text{F-S-E}\rangle, \langle p_3, e_{Decrement} \cdot n_1\rangle, \langle g_1, n' \cdot n\rangle\Big)$$

The counter is decremented:

$$\Big(\langle p_0, 0\rangle, \langle p_1, \text{T-S-F}\rangle, \langle p_2, \text{F-S-E}\rangle, \langle p_3, n'' \cdot n_1\rangle, \langle g_1, n' \cdot n\rangle\Big)$$

The counter is tested as to whether it is 0 by exchanging the message "is-zero":

$$\Big(\langle p_0, 0\rangle, \langle p_1, \text{T-S-F}\rangle, \langle p_2, \text{F-S-E}\rangle, \langle p_3, n''' \cdot n_1\rangle, \langle g_1, n' \cdot n\rangle\Big)$$

Therefore, *Decrements* confirms that the driver is stopped by sending the message "stop-driver":

$$\Big(\langle p_0, 0\rangle, \langle p_1, \text{T-S-F}\rangle, \langle p_2, \text{T-S-E}\rangle, \langle p_3, n_1\rangle, \langle g_1, n' \cdot n\rangle\Big)$$

Now, the resources are released:

$$\Big(\langle p_0, 0\rangle, \langle p_1, \text{T-S-F}\rangle, \langle p_2, \text{T-S-E}\rangle, \langle p_3, \text{RELEASE}\rangle, \langle g_1, n' \cdot n\rangle\Big)$$

At this point, the REQUEST at point n' decides to resume its execution, and it increments the counter:

$$\Big(\langle p_0, 10\rangle, \langle p_1, \text{T-S-F}\rangle, \langle p_2, \text{T-S-E}\rangle, \langle p_3, \text{RELEASE}\rangle, \langle g_0, n\rangle\Big)$$

Now the request executes its work:

$$\Big(\langle p_0, 10\rangle, \langle p_1, \text{T-S-F}\rangle, \langle p_2, \text{T-S-E}\rangle, \langle p_3, \text{RELEASE}\rangle, \langle g_0, n_{Work}\rangle\Big)$$

The work finishes:

$$\Big(\langle p_0, 10\rangle, \langle p_1, \text{T-S-F}\rangle, \langle p_2, \text{T-S-E}\rangle, \langle p_3, \text{RELEASE}\rangle, \langle g_0, n_{End-Work}\rangle\Big)$$

Now, REQUEST realises that the driver was stopped by communicating with STOPPING-EVENT using the message "is-stopped"and it aborts!

$$\Big(\langle p_0, 10\rangle, \langle p_1, \text{T-S-F}\rangle, \langle p_2, \text{T-S-E}\rangle, \langle p_3, \text{RELEASE}\rangle, \langle g_0, ABORT\rangle\Big)$$

It is easy to see that this trace will be found by our technique when using prefixes of length 8 (α_8) because it contains 8 synchronisation actions. The same can be achieved using the suffix abstraction.

6.2 Application to non-recursive examples

We also applied our implementation to several examples without recursion to which MAGIC had already been applied. The previous version of MAGIC handles *non-recursive* procedure calls by in-line expansion. The goal of these experiments was to determine whether the method described in the paper would improve MAGIC's performance. Indeed, inlining produces huge finite-state models. Consider for example a procedure having k control points and m calls to a procedure having n control points. This system can be modeled using a finite automaton with $O(k + mn)$ states, or a PDS that has only $O(k + n)$ states.

As described next, our results are encouraging, and we believe that they can be improved even further via a more efficient implementation. This shows that in addition to its utility in the verification of concurrent *recursive* programs, our technique represents an advance for recursive *as well as* non-recursive concurrent programs.

The experiments were carried out on a dual-Athlon-XP 1800+ machine with 3 GB of RAM running RedHat 9.0. We used the i^{th}-prefix abstraction for those experiments. The results are summarized in Figure 2. The columns are explained below the table. Each row corresponds to a different benchmark. More precisely, the *srvr* series is a single OpenSSL server (1444 lines of code); the *clnt* series is a single OpenSSL client (1386 lines of code); *ssl* series is a server and a client that execute concurrently (2830 lines of code). Each element of a series corresponds to a different property being verified on the same source code. The *ucos* benchmark (6263 lines

of code) refers to a single process running under the Micro-C operating system; *ucos-2* (12526 lines of code) and *ucos-3* (18789 lines of code) refer to two and three Micro-C processes, respectively. Finally, *casting* (54992 lines of code) refers to an actual controller deployed in a metal-casting plant. In all cases, the properties we checked were found to be valid.

Program	Previous version			Our technique		
	Abs	Verif	Mem	Abs	Verif	Mem
srvr-1	25.5	0.001	24.3	25.5	1.2	31.3
srvr-2	25.8	0.001	22.2	25.7	1.3	31.3
srvr-3	25.7	0.003	23.3	25.6	1.2	31.3
srvr-4	25.5	0.025	24.3	25.6	1.2	31.3
srvr-5	25.4	0.034	25.4	25.7	2.2	34.4
srvr-6	25.7	0.038	22.3	25.7	2.3	34.1
srvr-7	25.5	0.024	24.3	25.9	2.1	34.0
srvr-8	25.4	0.035	25.4	25.8	2.1	34.0
clnt-1	18.9	0.001	16.1	19.3	0.881	22.1
clnt-2	19.2	0.001	14.1	19.0	0.950	24.9
clnt-3	18.9	0.002	16.1	19.2	0.856	23.2
clnt-4	19.1	0.001	14.6	18.9	0.880	24.9
clnt-5	18.7	0.026	18.7	19.1	1.65	27.2
clnt-6	18.9	0.027	16.1	19.3	1.78	27.2
clnt-7	19.2	0.027	14.1	19.1	1.71	27.2
clnt-8	19.2	0.027	14.1	19.3	1.68	27.2
ssl-1	46.2	**16.2**	56.3	46.8	**2.82**	58.0
ssl-2	46.2	**16.1**	56.3	46.4	**3.83**	68.7
ssl-3	46.8	**14.0**	56.2	46.8	**19.2**	450
ssl-4	46.7	**14.2**	56.2	46.2	**2.76**	57.1
ssl-5	46.7	**14.0**	56.2	46.8	**3.02**	58.3
ssl-6	46.1	**14.0**	53.5	46.8	**2.93**	58.3
ssl-7	46.3	**15.0**	56.3	46.2	**3.34**	58.3
ucos	**29.1**	**0.044**	**293**	**6.8**	**0.702**	**110**
ucos-2	**84.8**	**578**	**639**	**16.5**	**1.324**	**161**
ucos-3	**168**	*	*	**29.2**	**2.144**	**213**
casting	45.7	0.257	196.1	40.3	38.2	2145

Figure 2. Abs = predicate-abstraction time (sec); Verif = model-checking time (sec); Mem = memory consumtion (MB); * = exceeded 2 GB memory limit.

We observe that predicate-abstraction times are comparable in the first three series of benchmarks because there was essentially no inlining to be done. In the last four cases, the CPDS approach leads to faster predicate abstraction, e.g., by a factor of over 5.7 for *ucos-3*. In some cases, this also dramatically reduces overall memory consumption, e.g., by over an order of magnitude in the case of *ucos-3*. In terms of overall time, the CPDS approach is much better for the *ssl* series, *ucos-2* and *ucos-3* (in this case by over two orders of magnitude).

7. Conclusion and Future Work

The paper described how we extended the tool MAGIC to handle concurrent programs *with recursive procedure calls*. We model such programs using Communicating Pushdown Systems, and extend the CEGAR scheme implemented in MAGIC so that it can manipulate CPDSs. For that, we define new predicate-abstraction-refinement techniques to compute and refine CPDSs from C source code. Moreover, we propose a semi-decision procedure that solves reachability queries for CPDSs. Finally, we report on several encouraging non-trivial experimental results.

The semi-decision procedure is based on computing a series of refinable finite-chain abstractions of pushdown-system path languages. We use in this work i^{th}-prefix and i^{th}-suffix abstractions. It would be interesting to try other kinds of finite-chain abstractions, such as i^{th}-subword abstraction, or the i^{th}-occurrence ordering abstraction that considers the order between the i^{th} first occurences of each letter in the alpahbet, etc.

For the time being, MAGIC can only handle reachability properties for programs that contain both concurrency and recursion. In the future, we plan to extend it so that it can handle more general properties. To do so, we need to define (approximate) techniques for model checking CPDSs against LTL and CTL formulas.

Acknowledgment. We would like to thank Mihaela Sighireanu for helpful discussions about the Bluetooth driver program.

References

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCUR'97*. LNCS 1243, 1997.

[BET03a] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proceedings of the 30th ACM SIPGPLAN-SIGACT on Principles of Programming Languages, POPL'03*, 2003.

[BET03b] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *International Journal of Foundations of Computer Science*, 2003.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057, 2001.

[CC77] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP Conf. on Formal Description of Programming Concepts*. North-Holland Pub., 1977.

[CCG+03] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verifi cation of software components in C. In *International Conference on Software Engineering (ICSE)*, pages 385–395, 2003.

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refi nement. In *Computer Aided Verifi cation*, pages 154–169, 2000.

[CISW05] S. Chaki, J. Ivers, N. Sharygina, and K. Wallnau. The comfort reasoning framework. In *Computer Aided Verifi cation*, 2005.

[CK88] K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, pages 57–66, 1988.

[EK99] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *FOSSACS'99*, volume LNCS 1578, 1999.

[ES01] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *In Proc. of CAV'01, number 2102 in Lecture Notes in Computer Science, pages 324-336. Springer-Verlag*, 2001.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A Direct Symbolic Approach to Model Checking Pushdown Systems. In *Infi nity'97*, 1997.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verifi cation (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[KIG05] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *Computer Aided Verifi cation*, 2005.

[KRML] N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems.

http://www.cs.wisc.edu/wpis/wpds++/.

[Kur94] R. P. Kurshan. Computer-aided verifi cation of coordinating processes: the automata-theoretic approach. In *Princeton University Press*, 1994.

[Mor82] J.M. Morris. Assignment and linked data structures. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 35–41. D. Reidel Publishing Co., Boston, MA, 1982.

[Nel80] G. Nelson. Techniques for program verifi cation. Phd thesis, Stanford University, 1980.

[NMW+01] G. Necula, S. McPeak, W. Weimer, B. Liblit, R. To, and A. Bhargava. C intermediate language. *http://manju.cs.berkeley.edu/cil*, 2001.

[QR05] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.

[QRR04] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL 04: ACM Principles of Programming Languages*, pages 245–255, 2004.

[QW04] S. Qadeer and D. Wu. Kiss: Keep it simple and sequential. In *PLDI 04: Programming Language Design and Implementation*, pages 14–24, 2004.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22:416–430, 2000.

[RSJ03] T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SAS*, pages 189–213, 2003.

[RSJM] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. of Comp. Prog.* To appear.

[Sch02] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.