Verifying Concurrent Message-Passing C Programs with Recursive Calls*

S. Chaki[1], E. Clarke[1], N. Kidd[2], T. Reps[2], and T. Touili[3]

[1] Carnegie Mellon University, Pittsburgh, USA
[2] University of Wisconsin, Madison, USA
[3] 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. We implemented our technique in the model checker MAGIC and found a previously unknown bug in a version of a Windows NT Bluetooth driver.

1 Introduction

Analysis of concurrent software represents a major challenge in the model-checking community. Concurrent programs include various complex features such as: (1) the manipulation of data ranging over unbounded domains, e.g., integers and reals (or very large domains like 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 synchronization statements [1]. 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 for which termination is not guaranteed. This work uses the latter approach to sidestep the undecidability issue. Though not guaranteed to terminate, such an approach can still be useful; for instance, our tool found a previously unknown bug in a version of a Windows NT Bluetooth driver.

During the last few years, several authors have addressed related issues. *Pushown systems* have been proposed as an adequate formalism to describe *pure sequential recursive programs* [2, 3]. They are able to represent the potentially infinite configurations of recursive programs in a symbolic manner using regular languages [4, 5]. Recently, compositions of *pushown systems*, called *communicating pushown systems*, have been used to model *concurrent recursive programs* [6, 7]. However, in these cases, all data were assumed to have a *small* finite domain.

On the other hand, abstract-interpretation techniques [8] have been used to deal with data ranging over unbounded (or very large) domains. More recently, automated *predicate-abstraction* techniques [9] 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. The size of

* Supported by ONR under contracts N00014-01-1-{0796,0708}.

the model increases with the number of predicates, which increases the cost of model checking. Hence, a central problem in predicate abstraction is the discovery of a *small* set of predicates sufficient to prove the desired property. *CounterExample Guided Abstaction Refinement* (CEGAR) techniques [10, 11] have been used to find such a small set. The idea is: (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 [12], and concurrent *non-recursive* programs in the tools BLAST [13] and MAGIC [14].

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 source code of a concurrent (recursive) C program that manipulates variables that range over very large domains, and that can refine CPDS abstractions to eliminate a given counterexample. Our techniques are defined *componentwise*, which makes them *compositional* and *scalable to large programs* (e.g., one experiment on an 18 KLOC program ran in less than 2.2 seconds).
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 (CFLs) 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 two non-trivial examples (a Windows NT Bluetooth driver and an algorithm for concurrent insertions in a binary search tree) that could not be handled with the previous version of MAGIC. In addition, it discovered a previously unknown bug in a second version of the Windows NT Bluetooth driver. Moreover, the implementation provides improved performance for non-recursive examples that the previous version of MAGIC was able to handle only via in-lining. This shows that our technique represents an advance for *non-recursive* as well as recursive concurrent programs.

One of the novel features of this work is that it applies the CEGAR scheme at two levels: (1) at the model-checking level to solve reachability queries in CPDSs: the CPDS model checker uses a CEGAR scheme in its semi-decision procedure for testing emptiness of the intersection of two CFLs (see §4), and (2) at the predicate-abstraction level to deal with unbounded domain variables (see §5). As far as we know, this is the first time that CEGAR is used in the model-checker itself.

The remainder of the paper is organized as follows: §2 defines the CPDS model; §3 describes how to generate a CPDS from a C program using predicate abstraction; §4 presents the semi-decision procedure for model-checking a CPDS; §5 presents our predicate-abstraction-refinement techniques; §6 reports experimental results; §7 discusses related work.

2 Preliminary definitions

A *pushdown system* (PDS) is a four-tuple $\mathcal{P} = (Q, Act, \Gamma, \Delta)$ where Q 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 $\langle q,\gamma\rangle \xhookrightarrow{a} \langle q',w\rangle$, where $q,q' \in Q, a \in Act$, $\gamma \in \Gamma$, and $w \in \Gamma^*$. Without loss of generality, we assume that for all rules of Δ, $|w| \leq 2$. This is not restrictive because any PDS can be transformed into a PDS of this form [15]; moreover, the transition rules obtained from a program have this form. A *configuration* of $\mathcal{P}$ is a pair $\langle q,w\rangle$, where $q \in Q$ and $w \in \Gamma^*$ is *the contents of the stack*. A set of configurations C is *regular* if for each $q \in Q$ the language $\{w \in \Gamma^* \mid \langle q,w\rangle \in C\}$ is regular.

For every $a \in Act$, we define a transition relation $\xrightarrow{a}$ between the configurations of $\mathcal{P}$ as follows: if $\langle q,\gamma\rangle \xhookrightarrow{a} \langle q',w\rangle \in \Delta$, then $\langle q,\gamma v\rangle \xrightarrow{a} \langle q',wv\rangle$ 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) [7] is a tuple $CP = (\mathcal{P}_1,\ldots,\mathcal{P}_n)$ of PDSs 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 has the property 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 CFLs.

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

Given a set of global configurations G, the successors of G (denoted by $Post^*(G)$) are defined as before.

3 Componentwise Predicate Abstraction

We model concurrent recursive programs using CPDSs. This section describes how to extract a CPDS from a parallel program. (A more in-depth discussion is given in [16].)

Suppose that we are given n concurrent recursive C components. For each component i, we extract a PDS $\mathcal{P}_i$. The parallel composition of the C components is represented by the CPDS corresponding to the tuple ($\mathcal{P}_1$,...,$\mathcal{P}_n$). To extract each $\mathcal{P}_i$, we extend the approach originally used in MAGIC [14], which *automatically* extracts a finite-state automaton from C code, to extract a PDS. Without loss of generality, we assume there are only six kinds of statements in programs: assignments, procedure calls, if-then-else branches, gotos, synchronization statements, and returns. We use CIL [17] to transform arbitrary C programs into this form.

Each PDS is defined in terms of a current set of seed predicates (which is initially empty). Each predicate represents a set of assignments over 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.

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 [18].

Let $\mathcal{C}$ be a set of seed predicates. 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 $P[\mathcal{C}]_n$ as follows:

Initially, $P[\mathcal{C}]_n = \emptyset$ for every point n. We repeat the following until for every n, $P[\mathcal{C}]_n$ 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 $\mathcal{W}_{s_n}\big(P[\mathcal{C}]_{n'}\big)$ to $P[\mathcal{C}]_n$.
2. if s_n is an if statement and n' is its then or else successor, then add $P[\mathcal{C}]_{n'}$ to $P[\mathcal{C}]_n$. Moreover, if c is the corresponding condition of s_n such that $c \in \mathcal{C}$, then add c to $P[\mathcal{C}]_n$.
3. if s_n is a goto or a synchronisation statement that has n' as successor, then add $P[\mathcal{C}]_{n'}$ to $P[\mathcal{C}]_n$.
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 $P[\mathcal{C}]^{loc}_{n'}$ and $P[\mathcal{C}]^{glob}_{e_\pi}$ to $P[\mathcal{C}]_n$.

(This method 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 item 1. Predicate set $P[\mathcal{C}]_n$ is only capable of making a certain set of distinctions among the concrete states that can arise at execution time at point n. Let s_n be an assignment that has n' as successor. Item 1 adds $\mathcal{W}_{s_n}\big(P[\mathcal{C}]_{n'}\big)$ to $P[\mathcal{C}]_n$ because if $\mathcal{W}_{s_n}(\varphi)$ is true at n, then φ must be true at n'. We wish to minimize the loss of precision in characterizing the states at n': to be able to determine whether φ holds at n', we need to know whether $\mathcal{W}_{s_n}(\varphi)$ holds at n.

Finally, let $P[\mathcal{C}] = \cup P[\mathcal{C}]_n$, 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 PDS Extraction

Using $\mathcal{C}$, 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 on $P[\mathcal{C}]^{glob}$; Act contains the action τ, as well as the other synchronization actions of the program; Γ is the set of all pairs (n, loc), where n is a control point of the sequential component, and loc is a valuation of $P[\mathcal{C}]_n^{loc}$; Δ is defined using the sequential component's control flow graph. For example, if s is a non-synchronizing assignment statement at control location n_1 with successor n_2, then Δ contains all the PDS rules $\langle glob, (n_1, loc)\rangle \xhookrightarrow{\tau} \langle glob', (n_2, loc')\rangle$, where $glob \in P[\mathcal{C}]_{n_1}^{glob}$, $glob' \in P[\mathcal{C}]_{n_2}^{glob'}$ (resp. $loc \in P[\mathcal{C}]_{n_1}^{loc}$, $loc' \in P[\mathcal{C}]_{n_2}^{loc'}$), such that they potentially satisfy $(\mathcal{W}_s(glob') \wedge glob)$ (resp. $(\mathcal{W}_s(loc') \wedge loc)$).[4] These formulas ensure that the generated PDS has more behaviors than the concrete program.

If instead s is a synchronizing statement with action a, then Δ contains all the PDS rules $\langle glob, (n_1, loc)\rangle \xhookrightarrow{a} \langle glob', (n_2, loc')\rangle$, where again $glob$ and $glob'$ (resp. loc and loc') potentially satisfy the conditions stated above. Further details about converting the various types of C statements to their corresponding PDS rules are given in [16].

3.3 Comparision with the predicate-abstraction technique of SLAM

The SLAM tool [12] uses predicate-abstraction techniques to extract a Boolean program from C source code. One can then use Schwoon's translation [15] to obtain a PDS from a Boolean program. Compared with the techniques used in SLAM, the approach sketched in §3.2 has 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. We close a given set of seed predicates $\mathcal{C}$ by computing weakest preconditions along the different possible paths of the program. 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.

4 Reachability Analysis of CPDSs

Given a program that consists of n sequential components, we usually ask the following query: "Suppose that the system starts from a configuration where each component i, for $i = 1, \ldots, n$, is at its initial control point n_0^i. Can one of the components reach an error point?" Our technique answers this kind of question by modeling the program as the CPDS $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ with initial configurations $C_1 \times \ldots \times C_n$ and error configurations $C'_1 \times \ldots \times C'_n$ (where the states of configurations in some C'_i correspond to error points and the states of configurations in $C'_1, \ldots, C'_{i-1}, C'_{i+1}, \ldots, C'_n$ are unconstrained). If the error configurations are reachable from the initial configurations, our algorithm returns a sequence of synchronization actions that yield a failing program run. We show in this

[4] 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 [19] that always terminates and answers TRUE, FALSE, or UNKNOWN to the question whether a given formula $\neg(p_1 \wedge p_2)$ is valid. If the validity checker returns FALSE or UNKNOWN to the question "Is $\neg(p_1 \wedge p_2)$ valid?", then $(p_1 \wedge p_2)$ is potentially satisfiable.

section how to tackle the reachability analysis of these systems. In the remainder of the paper, we restrict ourselves to systems that consist of two components. The technique can be extended in a straightforward manner to the general case (see [7] for more details); the implementation discussed in §6 supports an arbitrary number of components.

We reduce the reachability problem for CPDSs to deciding the emptiness question for the intersection of two CFLs 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 CFL 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 the intersection of two CFLs 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 witness 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 each 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 brevity, 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 PDS path languages, our technique is based on the approach presented by Bouajjani et al. [7], which is summarized below.

Consider an abstract lattice $(D, \leq, \sqcap, \sqcup, \perp, \top)$ 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; $\bar{0} = \perp$; $\bar{0}$ 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, $\forall a, b \in D, a \leq b \iff a \oplus b = a$.

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

– D contains an element v_a for every letter $a \in Lab$,

– There is an abstraction function $\alpha : 2^{Lab^*} \to D$ and a concretization function $\gamma : D \to 2^{Lab^*}$ defined as follows:
$\alpha(L) = \bigoplus_{a_1 \cdots a_n \in L} v_{a_1} \odot \cdots \odot v_{a_n}$ and $\gamma(x) = \{a_1 \cdots a_n \in Lab^* \mid v_{a_1} \odot \cdots \odot v_{a_n} \leq x\}$,
such that $\gamma(\perp) = \emptyset$.

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 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 $\{\varepsilon\}$, respectively.

Therefore, to compute the over-approximation $\gamma\big(\alpha(L_i)\big)$, 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 1. *[7, 20] Let $\mathcal{P} = (Q, Act, \Gamma, \Delta)$ be a PDS; let 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 time $O(h|\Delta||Q|^2)$.*

Two different algorithms provide the basis of this theorem, one due to Bouajjani et al. [6, 7], the other to Reps et al. [20, 21]. The latter has been implemented in a tool called WPDS++ [22]. 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 $\alpha(L_1) \sqcap \alpha(L_2) = \perp$. Indeed, using the fact that $\gamma(\perp) = \emptyset$, we can show that
$\forall L_1, L_2 \in Lab^*, \alpha(L_1) \sqcap \alpha(L_2) = \perp \Leftrightarrow \gamma\big(\alpha(L_1)\big) \cap \gamma\big(\alpha(L_2)\big) = \emptyset$.

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 at least as precise as 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 $a \in Lab$, $v_a = a$; $\oplus = \cup$; $\sqcap = \cap$; $U \odot V = \big\{(uv)_i \mid u \in U, v \in V\big\}$, where $(w)_i$ is the prefix of w of length i; $\bar{0} = \emptyset$; $\bar{1} = \{\varepsilon\}$; $\leq = \subseteq$.

Let α_i and γ_i be the abstraction and 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, $\gamma_i\big(\alpha_i(L)\big) = \{w \in \alpha_i(L) \mid |w| < i\} \cup \{wv \mid w \in \alpha_i(L), |w| = i, v \in Lab^*\}$.

Note 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 also easy to see that if $i > j$, then α_i is at least as precise as 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 refinable series of finite-chain abstractions $\alpha_1, \alpha_2, \alpha_3, \ldots$.

Remark 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 (based on the i^{th}*-prefix* abstraction) for the reachability problem for CPDSs:

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 continue from step 2.

Theorem 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 2. It follows from Theorem 1 that at each step i, computing $\alpha_i(L)$ necessitates $O(2^{|Lab|^i}|\Delta||Q|^2)$ 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 the algorithm. However, in practice, our implementation behaves well, as discussed in §6.

4.5 Example

Let $\mathcal{P}_1$ be the PDS that has the following rules:
$r_1 : \langle p, n_0 \rangle \xhookrightarrow{a} \langle p, n_1 \rangle$; $r_2 : \langle p, n_1 \rangle \xhookrightarrow{\tau} \langle p, n_0 n_2 \rangle$; $r_3 : \langle p, n_2 \rangle \xhookrightarrow{b} \langle p, \varepsilon \rangle$; $r_4 : \langle p, n_0 \rangle \xhookrightarrow{b} \langle p, \varepsilon \rangle$.
Let $\mathcal{P}_2$ be the PDS that has the following rules:
$r_1' : \langle q, m_0 \rangle \xhookrightarrow{a} \langle q, m_1 \rangle$; $r_2' : \langle q, m_1 \rangle \xhookrightarrow{b} \langle q, m_2 \rangle$; $r_3' : \langle q, m_2 \rangle \xhookrightarrow{\tau} \langle q, m_0 m_3 \rangle$;
$r_4' : \langle q, m_3 \rangle \xhookrightarrow{b} \langle q, \varepsilon \rangle$; and $r_5' : \langle q, m_0 \rangle \xhookrightarrow{d} \langle q, \varepsilon \rangle$.

For $\mathcal{P}_1$, let L_1 be $L\big(\langle p, n_0 \rangle, \langle p, \varepsilon \rangle\big) = \{a^k b b^k \mid k \geq 0\}$. For $\mathcal{P}_2$, let L_2 be $L\big(\langle q, m_0 \rangle, \langle q, \varepsilon \rangle\big) = \{(ab)^k d b^k \mid k \geq 0\}$. Note 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 conclude that $L_1 \cap L_2 = \emptyset$.

5 Componentwise Refinement

The construction of the CPDS model from the C program involves predicate abstraction. It is parametrized by a set of predicates. A central issue in predicate abstraction is how to find a small set of predicates that allows a property of interest to be established. In our case, the property in question is whether the system can reach an error configuration from the initial configuration, where component i (where, e.g., $i = 1,2$) starts in configuration $\langle glob_0^i, (n_0^i, loc_0^i)\rangle$, n_0^i is the initial control point of component i, and $glob_0^i, loc_0^i$ are initial valuations of the global and local variables, respectively. Similarly, an error configuration 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 an appropriate set of predicates by applying a CEGAR approach, as described below.

We start with a model involving an empty set of seed predicates, and perform the model-checking step described in §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 a real execution of the program (in which case, we have shown that the program is not correct), or whether the apparently-erroneous behavior has been introduced by abstraction. If the latter 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 rule sequence $r_1^i, \ldots, r_{m_i}^i$. If either component fails to perform its corresponding sequence, we refine its corresponding PDS to eliminate the spurious rule sequence. Note that all of these steps are done *componentwise*, which makes the technique compositional and scalable to large programs.

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 corresponds 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 §3 using the set of predicates $P[\mathcal{C}]$. 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 $P[\mathcal{C}']$ as described in §3.2. 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 An example illustrating the CEGAR predicate-abstraction technique

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

```
D1:
main() {
      n0: int x=10;
      n1: proc();
      n2: return;
}

void proc() {
      n3: if (x < 10)
      n4:        a;
      n5: else proc();
      n6: return;
}

D2:
main() {
      m0: a;
      m1: return;
}
```

The CPDS model.

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:
$r_1 : \langle p, n_0\rangle \xhookrightarrow{\tau} \langle p, n_1\rangle$; $r_2 : \langle p, n_1\rangle \xhookrightarrow{\tau} \langle p, n_3n_2\rangle$; $r_3 : \langle p, n_2\rangle \xhookrightarrow{\tau} \langle p, \varepsilon\rangle$; $r_4 : \langle p, n_3\rangle \xhookrightarrow{\tau} \langle p, n_4\rangle$;
$r_5 : \langle p, n_3\rangle \xhookrightarrow{\tau} \langle p, n_5\rangle$; $r_6 : \langle p, n_4\rangle \xhookrightarrow{a} \langle p, n_6\rangle$; $r_7 : \langle p, n_5\rangle \xhookrightarrow{\tau} \langle p, n_3n_6\rangle$; $r_8 : \langle p, n_6\rangle \xhookrightarrow{\tau} \langle p, \varepsilon\rangle$.

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

Case #2: We have $\mathcal{C} = \{(x < 10)\}$: We model the component D_1 by the following PDS $\mathcal{P}'_1$. We have: $P[\mathcal{C}]_{n_1} = P[\mathcal{C}]_{n_3} = P[\mathcal{C}]_{n_5} = \{x < 10\}$, and $P[\mathcal{C}]_n = \emptyset$ for the other points (while computing $P[\mathcal{C}]_{n_0}$, we find the predicate $10 < 10$. Because we ignore predicates that are trivially true or false, we keep $P[\mathcal{C}]_{n_0} = \emptyset$). 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:
$\langle p_3, n_0\rangle \xhookrightarrow{\tau} \langle p_1, n_1\rangle$; $\langle p_1, n_1\rangle \xhookrightarrow{\tau} \langle p_1, n_3n_2\rangle$; $\langle p_3, n_2\rangle \xhookrightarrow{\tau} \langle p_3, \varepsilon\rangle$; $\langle p_2, n_3\rangle \xhookrightarrow{\tau} \langle p_3, n_4\rangle$;
$\langle p_1, n_3\rangle \xhookrightarrow{\tau} \langle p_1, n_5\rangle$; $\langle p_3, n_4\rangle \xhookrightarrow{a} \langle p_3, n_6\rangle$; $\langle p_1, n_5\rangle \xhookrightarrow{\tau} \langle p_1, n_3n_6\rangle$; $\langle p_3, n_6\rangle \xhookrightarrow{\tau} \langle p_3, \varepsilon\rangle$.

Refinement. Consider the 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 $(\mathcal{P}_1, \mathcal{P}_2)$, the model checker answers that (n_6n_2, m_1) is reachable with the following sequences: $r_1r_2r_4r_6$ for $\mathcal{P}_1$, and r'_1 for $\mathcal{P}_2$. Using our method, we can check that $r_1r_2r_4r_6$ is spurious because $\varphi = (x = 10) \wedge (x < 10)$ is not satisfiable. Therefore, we refine PDS $\mathcal{P}_1$ using $\mathcal{C} = \{(x < 10)\}$ to obtain the PDS $\mathcal{P}'_1$. 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 our method in ComFoRT [23], a model checker built on top of MAGIC [14], and experimented with a set of non-trivial benchmarks. The implementation supports two kinds of abstractions described in §4.2: the i^{th}-prefix and the i^{th}-suffix abstractions.

version	*# procs.*	*abstraction*	*len.*	*time(secs.)*	*mem.*
BT_1	1	i^{th}-prefix	8	8	358
BT_1	1	i^{th}-suffix	8	5	334
BT_2	2	i^{th}-prefix	14	67	490
BT_2	2	i^{th}-suffix	14	20	391
BT_3	1	i^{th}-suffix	6	2	304
BT_3	2	i^{th}-suffix	7	25	441

Table 1. Performance for the Bluetooth driver (len. = counterexample length, except for BT_3, where it indicates the abstraction length;mem. = memory usage (MB)).

6.1 Application to concurrent recursive programs

We applied the technique to two non-trivial recursive concurrent programs that could not be handled with the original (non-recursive) version of MAGIC: a Windows NT Bluetooth driver, and an algorithm for concurrent insertions in a binary search tree.

The experiments were performed on a 3.0 GHz P4 SMP with 2 GB memory, running Linux 2.4.21-27.0.1.

A new bug in a Windows NT Bluetooth driver. The tool found bugs in two versions of this program (BT_1 and BT_2) and verified the correctness for a two-process instantiation of a third version (BT_3). BT_1 was the version for which KISS had previously found a bug [24], and our tool identified the same bug. In contrast to KISS (as well as the work reported in [25]), our approach can also verify correctness by determining that all error configurations are unreachable. The authors of [24] sent us BT_2 to see if correctness could be verified. Instead, we found a bug in BT_2 that can arise when two concurrent processes are running. Both bugs could be detected with the i^{th}*-prefix* abstraction as well as the i^{th}*-suffix* abstraction. Using the counterexample found by our tool, we modified BT_2 to create BT_3, and analyzed BT_3 for a two-process configuration. The tool reported that the error state is unreachable in BT_3.

Tab. 1 shows the running times and memory consumption for these experiments. The i^{th}*-suffix* abstraction is more efficient because we use it to compute Pre^* from the error states. Therefore, the language will stop growing once Pre^* has traversed i actions from the error state.

Note that the Bluetooth driver is not recursive; however, we use a recursive process to model a counter. In the real program, the counter is an integer (which is a global variable). Because we needed to represent global variables by means of synchronization actions, we had to represent the counter as a process. We modeled the counter process as a PDS with stack alphabet $\{1\}$. The number of 1's on the stack corresponds to the value of the counter. Then, incrementing the counter amounts to pushing a 1 onto the stack, and decrementing it amounts to popping a 1 off the stack.

An algorithm for concurrent insertions in a binary search tree. We also considered an algorithm that handles a finite number of concurrent insertions in a binary search tree [26]. The algorithm can be applied to handle simultaneous insertions into a database (by several users), or to reduce the time necessary for a single insertion. The algorithm was modified so that one process does not adhere to the required lock and unlock semantics, and we then applied our tool (using the i^{th}-prefix abstraction) to the modified version. The times needed to detect the bug (as a function of number of processes) are shown in Tab. 2.

# procs.	*len.*	*time (secs.)*
2	1	0.8
3	1	0.8
4	1	0.8
5	1	1.1
6	1	2.7
7	1	12.9

Table 2. Times needed to detect the bug in the concurrent-insertions algorithm.

6.2 Application to non-recursive examples

We 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 purpose of the non-recursive experiments was to test whether our technique was better than inlining.

We tested sequential programs to determine whether the implementations were of comparable speed (without the complication of concurrency). They were not: the times for the *srvr-i* and *clnt-i* examples show that the overhead introduced by our technique is substantial (cf. the times in the two columns of Tab. 3 labeled "Verif"). The reason for this difference is that MAGIC performs a reachability query over an FSM, whereas we use the full CPDS machinery (which includes the inner CEGAR loop).

Sequential Experiments

Program	MAGIC			CPDS			
	Abs	Verif	Mem	Abs	Verif	Mem	Len
srvr-1	25.5	0.001	24.3	25.5	1.2	31.3	2
srvr-2	25.8	0.001	22.2	25.7	1.3	31.3	2
srvr-3	25.7	0.003	23.3	25.6	1.2	31.3	2
srvr-4	25.5	0.025	24.3	25.6	1.2	31.3	2
srvr-5	25.4	0.034	25.4	25.7	2.2	34.4	2
srvr-6	25.7	0.038	22.3	25.7	2.3	34.1	2
srvr-7	25.5	0.024	24.3	25.9	2.1	34.0	2
srvr-8	25.4	0.035	25.4	25.8	2.1	34.0	2
clnt-1	18.9	0.001	16.1	19.3	0.881	22.1	2
clnt-2	19.2	0.001	14.1	19.0	0.950	24.9	2
clnt-3	18.9	0.002	16.1	19.2	0.856	23.2	2
clnt-4	19.1	0.001	14.6	18.9	0.880	24.9	2
clnt-5	18.7	0.026	18.7	19.1	1.65	27.2	2
clnt-6	18.9	0.027	16.1	19.3	1.78	27.2	2
clnt-7	19.2	0.027	14.1	19.1	1.71	27.2	2
clnt-8	19.2	0.027	14.1	19.3	1.68	27.2	2

Concurrent Experiments

Program	MAGIC			CPDS			
	Abs	Verif	Mem	Abs	Verif	Mem	Len
ssl-1	46.2	**16.2**	56.3	46.8	**2.82**	58.0	2
ssl-2	46.2	**16.1**	56.3	46.4	**3.83**	68.7	2
ssl-3	46.8	**14.0**	56.2	46.8	**19.2**	450	4
ssl-4	46.7	**14.2**	56.2	46.2	**2.76**	57.1	2
ssl-5	46.7	**14.0**	56.2	46.8	**3.02**	58.3	2
ssl-6	46.1	**14.0**	53.5	46.8	**2.93**	58.3	2
ssl-7	46.3	**15.0**	56.3	46.2	**3.34**	58.3	2
ucos	**29.1**	**0.044**	**293**	**6.8**	**0.702**	**110**	2
ucos-2	**84.8**	**578**	**639**	**16.5**	**1.324**	**161**	2
ucos-3	**168**	*	*	**29.2**	**2.144**	**213**	2
casting	45.7	0.257	196.1	40.3	38.2	2145	3

Table 3. Abs = predicate-abstraction time (sec); Verif = model-checking time (sec); Mem = memory usage (MB); * = exceeded 2 GB memory limit; Len = abstraction length.

Despite this handicap, when model checking concurrent programs, our technique was almost always better than the in-lining technique of the base MAGIC system (see the bold entries in the right-hand table of Tab. 3). The new technique outperforms MAGIC in these cases because it avoids the state-space explosion that can occur because of in-lining. The cost of the technique depends heavily on the length of the synchronization sequences examined by the model checker. This can be seen by comparing the times for the non-recursive examples and for the Bluetooth example. Each of the non-recursive examples are verifed using strings of only 2–4 synchronization actions. However, BT_1, BT_2, and BT_3 need 8, 14, and 7 actions, respectively, which causes the running times to be much larger. This is an interesting aspect of our technique, namely, the limiting factor is the length of the synchronization sequences considered, not program size. Indeed, the analysis times are encouraging for the programs *ucos-2* and *ucos-3*, which are 12K LOC and 18K LOC, respectively (see Tab. 3).

7 Related Work

Bouajjani et al. also reduced the reachability problem for CPDSs to computing over-approximations of CFLs; however, no CEGAR techniques were presented there [6, 7]. More precisely, their work computes over-approximations A_1 and A_2 of two given CFLs L_1 and L_2, and if $A_1 \cap A_2 = \emptyset$, one concludes that $L_1 \cap L_2 = \emptyset$. However, no conclusion can be made automatically if $A_1 \cap A_2 \neq \emptyset$. In particular, 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-program model-checking tools, such as SLAM [12], BLAST [13], ZING [27], and KISS [24]. 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 explicitly represented in ZING. In contrast, in our PDS modeling framework, they are symbolically represented with regular languages. On the other hand, SLAM and ZING use predicate-abstraction techniques to extract a Boolean program from a C program with recursion. Schwoon has implemented a translation from Boolean programs to PDSs in the MOPED tool [15]. However, MOPED cannot handle concurrent programs. Our CPDS predicate-abstraction-refinement techniques are performed componentwise, and amount to performing successive sequential PDS predicate-abstractions and refinements. These successive steps could be performed using SLAM and then MOPED; however, in this paper, we present predicate-abstraction techniques that create a PDS from C source code of a sequential component directly and more efficently (i.e., without going through an intermediate Boolean program).

Finally, the techniques presented in [28, 25] also use multiple PDSs to model concurrent recursive programs. However, [28] is restricted to programs that communicate via a finite number of locks, and assumes a certain nesting condition on the locks. As for [25], it uses shared-variables for communication between threads, whereas we use synchronizing actions (these two models can simulate each other). The technique presented in [25] sidesteps the undecidability of the reachability problem for multiple PDSs by putting a bound k on the number of interleavings between different threads, whereas we sidestep undecidability by computing abstractions of CFLs (without bounding the number of interleavings). In certain cases, our technique can be more powerful than the one presented in [25]. Namely, when we find $A_1 \cap A_2 = \emptyset$, we can infer that the target configurations are not reachable, whereas the technique of [25] can never establish such a property because it computes an underapproximation. Indeed, after correcting BT_2 to create BT_3, our tool verified that BT_3 is correct for two processes. Finally, the technique of [25] has not been implemented, and no automatic techniques to translate C code to PDS are presented there.

Acknowledgments. We thank M. Sighireanu for helpful discussions about the Bluetooth driver program, S. Qadeer for providing us with BT_2, and A. Lal for his helpful insights.

References

1. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. TOPLAS **22** (2000) 416–430
2. Esparza, J., Knoop, J.: An automata-theoretic approach to interprocedural data-flow analysis. In: FOSSACS. (1999)
3. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: CAV. (2001)

4. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model checking. In: CONCUR. (1997)
5. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. In: Infinity. (1997)
6. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL. (2003)
7. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. Int. J. Found. of Comp. Sci. (2003)
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: POPL. (1977)
9. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: CAV. (1997)
10. Kurshan, R.P.: Computer-aided verification of coordinating processes: The automata-theoretic approach. In: Princeton University Press. (1994)
11. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. (2000)
12. Ball, T., Rajamani, S.: Automatically validating temporal safety properties of interfaces. In: SPIN. (2001)
13. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. (2002)
14. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. In: ICSE. (2003)
15. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, TUM (2002)
16. Chaki, S., Clarke, E., Kidd, N., Reps, T., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. Tech. Rep. 1532, Univ. of Wisconsin (2005)
17. Necula, G., McPeak, S., Weimer, W., Liblit, B., To, R., Bhargava, A.: C intermediate lang. (2001) http://manju.cs.berkeley.edu/cil.
18. Morris, J.: Assignment and linked data structures. In: Theoretical Foundations of Programming Methodology. D. Reidel Publishing Co. (1982)
19. Nelson, G.: Techniques for Program Verification. PhD thesis, Stanford University (1980)
20. Reps, T., Schwoon, S., Jha, S.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: SAS. (2003)
21. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. SCP **58** (2005)
22. Kidd, N., Reps, T., Melski, D., Lal, A.: WPDS++: A C++ library for weighted pushdown systems (2004) http://www.cs.wisc.edu/wpis/wpds++/.
23. Chaki, S., Ivers, J., Sharygina, N., Wallnau, K.: The ComFoRT reasoning framework. In: CAV. (2005)
24. Qadeer, S., Wu, D.: KISS: Keep it simple and sequential. In: PLDI. (2004)
25. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS. (2005)
26. Kung, H., Lehman, P.: Concurrent manipulation of binary search trees. TODS **5** (1980)
27. Qadeer, S., Rajamani, S., Rehof, J.: Summarizing procedures in concurrent programs. In: POPL. (2004)
28. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: CAV. (2005)