Symbolic Context-Bounded Analysis of Multithreaded Java Programs[*]

Dejvuth Suwimonteerabuth, Javier Esparza, Stefan Schwoon

Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany

Abstract. The reachability problem is undecidable for programs with both recursive procedures and multiple threads with shared memory. Approaches to this problem have been the focus of much recent research. One of these is to use context-bounded reachability, i.e. to consider only those runs in which the active thread changes at most k times, where k is fixed. However, to the best of our knowledge, context-bounded reachability has not been implemented in any tool so far, primarily because its worst-case runtime is prohibitively high, i.e. $O(n^k)$, where n is the size of the shared memory. Moreover, existing algorithms for context-bounded reachability do not admit a meaningful symbolic implementation (e.g., using BDDs) to reduce the run-time in practice. In this paper, we propose an improvement that overcomes this problem. We have implemented our approach in the tool jMoped and report on experiments.

1 Introduction

The analysis of procedural multithreaded programs has been intensively studied in the last years. If both recursion and multithreading are allowed, checking assertions such as reachability of program points is undecidable, even for programs whose variables are all of boolean type (see for instance [1]).

In order to cope with this undecidability result, three approaches have been proposed. First, the reachability of program points (and other, more general problems) has been studied and shown to be decidable for several special cases, like no communication between threads (but unrestricted thread creation) [2]; communication through nested locks [3, 4]; communication following a transactional policy [5]; or communication following a task-based policy [6]. In a second approach, techniques have been developed that compute an overapproximation of the set of reachable states. In [7, 8] it is shown that so called commutative abstractions of the set of reachable states can be effectively computed, while [9, 10] describes a CEGAR loop for another class of abstractions; this loop has been implemented in the MAGIC and Spade systems.

This paper lines with the third approach to the problem, namely the computation of underapproximations of the set of reachable states. In [11], Qadeer and Rehof proposed the first nontrivial technique, working for possibly recursive

[*] Partially supported by the DFG project *Algorithms for Software Model Checking.*

multithreaded programs communicating through global variables. They introduce the notion of *context switch* (transfer of control from one thread to another) and show how to compute, for a fixed k, the set of states reachable by computations with at most k context switches. The algorithm of [11] was extended in [12] to a more general programming model allowing for both global and local variables. In [13] it was also adapted to the analysis of concurrent queue systems.

Given a computation with k context switches, let us define its *trace* as the sequence of valuations of the global variables at which the switches take place. A shortcoming of the algorithms of [11, 12] is that they require to explicitly examine each trace one by one. If the global variables have n possible valuations, the number of traces is $O(n^k)$, which seriously limits the applicability of the approach. Recently, Lal et al. have proposed a new algorithm which avoids this problem at the expense of, loosely speaking, computing the reachability relation for a thread instead of only the set of reachable states [14]. To the best of our knowledge, none of the techniques for computing underapproximations presented in [11, 12, 14] has been implemented yet.

We present an improvement of context-bounded reachability algorithms that no longer requires to consider each trace individually. Our algorithm admits a symbolic implementation, using BDDs, which makes it usable in practice. We have implemented this approach as an extension of the jMoped tool [15]. The implementation can deal with Java code "as is", without replacing non-native libraries by stubs or manually translating the code into the modelling language of the checker. More specifically, the contributions of the paper are as follows:

- A new algorithm for the computation of the states reachable after a bounded number of context switches, based on *lazy splitting*. The algorithm deals symbolically with sets of traces that do not need to be distinguished, and splits them only when necessary. It addresses the same problem as [14], but with a different approach. The techniques of [11, 12] and [14] are compared in some more detail in the conclusions.
- Implementations of the algorithm of [11, 12] and the new algorithm in the jMoped tool.
- Optimizations of the algorithm for dealing with Java programs.
- Experimental evaluation on different versions of the Bluetooth driver considered in [9, 10], and on the java.util.Vector class.

We proceed as follows: Section 2 discusses preliminaries, recalls the details of the context-bounded reachability, and gives an overview of previous work (i.e., [11, 12]) and ours. Section 3 presents the novel elements of our algorithm. Section 4 discusses details of our implementation, and Section 5 provides experimental data. We conclude in Section 6.

2 Context-Bounded Reachability

In this section we define the problem we are working on and briefly discuss previous solutions.

2.1 Pushdown Networks

We will consider systems with n parallel processes, where n is a positive integer. Let $[n] = \{1, \dots, n\}$. A *pushdown network* is a triple $\mathcal{N} = (G, \Gamma, (\Delta_i)_{[n]})$, where:

- G is a finite set of *globals*;
- Γ is a finite *stack alphabet*;
- Δ_i, for each $i \in [n]$, is the finite set of *transition rules* for the i-th process (see below for its precise definition).

A *local configuration* of $\mathcal{N}$ is a pair $(g, \alpha) \in G \times \Gamma^*$, i.e. a global and a word over the stack alphabet. A *global configuration* of $\mathcal{N}$ is a tuple $(g, \alpha_1, \dots, \alpha_n)$, where g is a global and α_1 to α_n are words over the stack alphabet. For better distinction, we will denote local configurations by lowercase letters (e.g., c) and global configurations by uppercase letters (e.g., C). Intuitively, the system consists of n processes, each of which have some local storage (i.e., the local storage of the i-th process is the word α_i), and the processes can communicate by reading and manipulating the global storage represented by g. A *pushdown system* is a pushdown network where $n = 1$.

For each $i \in [n]$, Δ_i contains rules of the form $\langle g, \gamma \rangle \hookrightarrow \langle g', \alpha \rangle$, where g, g' are globals, $\gamma \in \Gamma$, and $\alpha \in \Gamma^*$. We define the local transition relation of the i-th process, written $\rightarrow_i$, as follows: $(g, \gamma\beta) \rightarrow_i (g', \alpha\beta)$ iff $\langle g, \gamma \rangle \hookrightarrow \langle g', \alpha \rangle$ in Δ_i and $\beta \in \Gamma^*$. In other words, each process by itself is a pushdown system; however, its control location is the global store shared by all processes. The transition relation of $\mathcal{N}$, denoted $\rightarrow_{\mathcal{N}}$ or just $\rightarrow$ for short, is defined as follows: $(g, \alpha_1, \dots, \alpha_i, \dots, \alpha_n) \rightarrow_{\mathcal{N}} (g', \alpha_1, \dots, \alpha'_i, \dots, \alpha_n)$ iff $(g, \alpha_i) \rightarrow_i (g', \alpha'_i)$. By $\rightarrow_i^*$, $\rightarrow_{\mathcal{N}}^*$, $\rightarrow^*$, we denote the reflexive and transitive closures of these relations.

2.2 Extensions

The computational model introduced in Section 2.1 is equivalent to the *concurrent pushdown systems* (CPS) originally used by Qadeer and Rehof [11]. In [12] an extension was studied, called APN. There, every thread has an additional local state, and transitions can either be "global" (depending on the global state, the local state of the thread and its stack) or "local" (depending only on the latter two). APN have the same expressive power as CPS, but allow for more refined complexity analysis. Since this aspect plays only a minor role in this paper, we chose to omit local states to simplify the presentation. However, it will be easy to see that our techniques also work for APN, with minor modifications.

Both [11] and [12] also studied extensions of the model with "dynamic" rules, i.e. the ability to fork new threads. We do not present this aspect in any detail here because the context-bounded reachability problem for systems with thread creation can be reduced to the context-bounded reachability problem for systems without (see [11] for details). Our implementation, discussed in Sections 4 and 5, does handle thread creation, and in these cases we denote a "fork" rule by $\langle g, \gamma \rangle \hookrightarrow \langle g', \alpha' \rangle \rhd \alpha''$. In this case, the global changes from g to g', the active process replaces its top-of-stack symbol γ by α', and a new thread with stack contents α'' is generated.

2.3 The reachability problem for pushdown networks

Let $\mathcal{N} = (G, \varGamma, (\varDelta_i)_{i\in[n]})$ be pushdown network. We define the following reachability problems:

- Given $i \in [n]$ and an initial local configuration $c_0 = (g_0, \alpha_0)$, the *local (forward) reachability problem* for the i-th process is to compute the set $post_i^*(c_0) = \{\, c \mid c_0 \rightarrow_i^* c \,\}$, i.e. the set of local configurations reachable by moves of the i-th process alone.
- Given an initial global configuration $C_0 = (g_0, \alpha_1, \ldots, \alpha_n)$, the *(forward) reachability problem* for $\mathcal{N}$ is to compute the set $post^*(C_0) = \{\, C \mid C_0 \rightarrow^* C \,\}$ of globally reachable configurations.

Both problems can be extended to sets of configurations in the usual manner. It is well-known that *local* reachability is closed under regularity, i.e. $post_i^*(c_0)$ is a regular set, and the result still holds when c_0 is replaced by a regular set of configurations. Moreover, the local reachability problem can be solved efficiently, in time proportional to $|G|^2 \cdot |\varDelta_i|$ [16].

In contrast, the (global) reachability problem is undecidable; more precisely, it is in general undecidable whether $C \in post^*(C_0)$, for a given pair C, C_0 [1]. For this reason, one tries to approximate $post^*(C_0)$. One such approximation, introduced in [11], uses the notion of *context-bounded* computations:

A *context* of $\mathcal{N}$ is a sequence of transitions where all moves are made by a single process. In other words, let us define a (global) reachability relation $\rightsquigarrow$ as follows: $(g, \alpha_1, \ldots, \alpha_i, \ldots, \alpha_n) \rightsquigarrow (g', \alpha_1, \ldots, \alpha_i', \ldots, \alpha_n)$ iff $(g, \alpha_i) \rightarrow_i^* (g', \alpha_i')$ for some $i \in [n]$. Then $\rightsquigarrow$ is a relation between global configurations reachable from each other in a single context. Correspondingly, we define $\overline{post}_i^*(C_0) = \{\, C \mid C_0 \rightarrow_i^* C \,\}$, i.e. $\overline{post}_i^*(C_0)$ is the set of global configurations reachable from C_0 by moves of the i-th process. Moreover, we denote by $\rightsquigarrow_j$, where $j \geq 0$, the reachability relation within j contexts: $\rightsquigarrow_0$ is the identity relation on global configurations, and $\rightsquigarrow_{i+1} = \rightsquigarrow_i \circ \rightsquigarrow$. We can now define our central problem:

- Given $k \geq 1$ and an initial global configuration C_0, the (forward) *context-bounded reachability problem* is to compute the set of configurations reachable in at most k contexts, i.e. the set $post_{\leq k}^*(C_0) = \{\, C \mid \exists j \leq k \colon C_0 \rightsquigarrow_j C \,\}$.

The context-bounded reachability problem is decidable, and its solution can be computed in a time that is essentially proportional to $(n \cdot |G|)^k$ [11, 12].

2.4 View tuples

Let us fix a pushdown network $\mathcal{N} = (G, \varGamma, (\varDelta_i)_{i\in[n]})$, a global configuration C_0, and a context bound $k \geq 1$ for the rest of the section.

The principal problem that one faces when solving the context-bounded reachability problem is to find a data structure for representing the set $post_{\leq k}^*(C_0)$. Note that while the global storage can assume only finitely many values, the number of possible stack contents is infinite, thus finding a suitable data structure for representing sets of global configurations is not straightforward. Here, we define a data structure that will be helpful to discuss the algorithms in previous work [11, 12] and in this paper. The main idea is to represent $post^*_{\leq k}(C_0)$ by so-called *view tuples*, which represent subsets of $post^*_{\leq k}(C_0)$.

Definition 1. *Let $c = (g, \alpha_1, \ldots, \alpha_n)$ be a global configuration. For $i \in [n]$, we call the local configuration (g, α_i) the i-*view *of c. A* view tuple *$T = (V_1, \ldots, V_n)$ is a collection where V_i is a* regular *set of local configurations, i.e. a set of i-views for each $i \in [n]$, represented by a finite automaton. T is associated with the following set of configurations:*

$$[\![T]\!] = \{\, (g, \alpha_1, \ldots, \alpha_n) \mid (g, \alpha_i) \in V_i \textit{ for all } i \in [n] \,\}$$

Not every set of global configurations can be represented as a view tuple. As a running example, let us consider a system with two processes, globals g, g', g'' and stack alphabet a, b. Consider the set $\mathcal{C}_0 = \{(g, a, a), (g', b, a), (g'', a, a), (g'', b, b)\}$. Suppose that there is a view tuple $T = (V_1, V_2)$ such that $[\![T]\!] = \mathcal{C}_0$. Then V_1 necessarily contains the pair (g'', a) and V_2 the pair (g'', b). But then, $[\![T]\!]$ also contains (g'', a, b), which is not in $\mathcal{C}_0$.

More importantly, the sets arising in the context-bounded reachability problem are not representable as view tuples. Continuing the example, suppose that $\Delta_1 = \{\langle g, a\rangle \hookrightarrow \langle g', b\rangle\}$ and $\Delta_2 = \{\langle g, a\rangle \hookrightarrow \langle g'', a\rangle, \langle g', a\rangle \hookrightarrow \langle g'', b\rangle\}$. Then $post^*_{\leq 2}((g, a, a))$ is exactly the set $\mathcal{C}_0$ from above.

In general, therefore, the result of a context-bounded reachability query is only representable as a *union* of view tuples. For instance, $\mathcal{C}_0$ can be partitioned into the sets $\mathcal{C}_1 := \{(g, a, a), (g'', a, a)\}$ and $\mathcal{C}_2 := \{(g', b, a), (g'', b, b)\}$, which are both representable as view tuples. As we shall see, our work differs from [11, 12] in the way we choose the view tuples contained in this union; more to the point, our representation requires (in general) fewer tuples.

2.5 A meta-algorithm for context-bounded reachability

In this section we discuss a meta-algorithm to solve the context-bounded reachability problem that unifies the solutions in [11, 12] and in this paper. It can be characterised as a worklist algorithm that computes the effect of one context at a time. While the algorithms from [11, 12] differ in some details, they can – for the purposes of this paper – be summarised by Algorithm 1.

The entries of the worklist are triples (j, i, T), where T is a view tuple reachable within j contexts such that i was the process that made the last move ($i = 0$ iff $j = 0$). Initially, the worklist contains just one view tuple representing the initial configuration C_0. In each iteration, the algorithm picks a view tuple from the worklist and computes the configurations that can be reached through a single additional context. Notice that since we are dealing with regular sets of configurations, this can be done by solving the local reachability problem, see, e.g., [16] or [17] for details. The previously active process, i, is excluded from consideration because it would not add any new information.

```
Input: $\mathcal{N} = (G, \Gamma, (\Delta_i)_i \in [n])$, context bound $k$, initial configuration
       $C_0 = (g_0, \alpha_0^1, \dots, \alpha_0^n)$
Output: the set of reachable global configurations.
1  $result := \emptyset$;
2  $worklist := \{ (0, 0, (\{(g_0, \alpha_0^1)\}, \dots, \{(g_0, \alpha_0^n)\})) \}$;
3  while $worklist \neq \emptyset$ do
4      remove $(j, i, T)$ from $worklist$;
5      add $[\![T]\!]$ to $result$;
6      if $j < k$ then
7          forall $i' \in [n] \setminus \{i\}$ do
8              $P = \overline{post}_{i'}^*([\![T]\!])$;
9              forall $T' \in \text{split}(P)$ do
10                 add $(j+1, i', T')$ to $worklist$;
11 return $result$;
```

Algorithm 1: Worklist algorithm for context-bounded reachability

The result of the local reachability algorithm is denoted by P, and the principal problem is that P may no longer be representable as a single view tuple. The task of the split function in line 9 is to generate a set of view tuples such that $\bigcup_{T' \in \text{split}(P)} [\![T']\!] = P$. Our work differs from previous solutions in the way this function is implemented. In [11, 12], split works as follows:

$$\begin{aligned} \text{split}(P) &= \{ T_g \mid g \in G \}, \text{ where} \\ T_g &= P \cap \{ (g, \alpha_1, \dots, \alpha_n) \mid \alpha_i \in \Gamma^*,\ i \in [n] \} \end{aligned}$$

It can be shown that the resulting sets are always view tuples. However, after each context, every worklist entry is split $|G|$ different ways. In the following, we call this approach *eager splitting*. Loosely speaking, eager splitting processes $n^k \cdot |G|^k$ worklist entries. Moreover, after each split, the algorithm will consider every element of G individually, which does not lend itself to a meaningful symbolic implementation (e.g., using efficient set representations such as BDDs). These reasons have been the major obstacle for a practical adoption of these algorithms.

In Section 3, we identify a coarser partition of P that leads to fewer splits, in the hope of making the algorithm faster in practice. We call this approach *lazy splitting*. We also describe how the partition can be computed using BDDs, which gives rise to a symbolic implementation of our algorithm.

3 Lazy Splitting

As discussed in Section 2.5, the algorithm for context-bounded reachability is parametrised by a function that splits the result of a local reachability query into view tuples. In this section, we present the key ingredient for our *lazy splitting* approach, i.e. we show how to (symbolically) compute a coarse partitioning.

To simplify the presentation we consider the case of two processes and assume w.l.o.g. that the second process is active, i.e. given a view tuple $T = (V_1, V_2)$, the task is to (i) compute the set $\overline{post}_2^*([\![T]\!])$ and (ii) split this set into new view tuples. Recall that a global configuration of a pushdown network with two processes is a tuple $c = (g, \alpha_1, \alpha_2)$, where g is a global and α_i is a local configuration of the i-th process.

Throughout this section we identify a set $X \subseteq X_1 \times \ldots \times X_n$ and the predicate $X(x_1, \ldots, x_n)$ such that $X(a_1, \ldots, a_n)$ holds iff $(a_1, \ldots, a_n) \in X$. We liberally mix set and logical notations, and write for instance $A(x) = \exists\, y{:}\, B(x, y)$ to mean $A = \{\, x \mid \exists y{:}\, B(x, y) \,\}$. Abusing notation, we shall sometimes denote the set $[\![T]\!]$, where T is a view tuple, simply by T.

We proceed as follows: We first identify a property between globals (called *confluence*) that prevents certain configurations from being included in the same partition. We then show how the confluence relation can be computed symbolically, using BDDs, and finally how partitions can be computed from this relation.

3.1 Confluence and safe partitions

Let $R_2(g, \alpha, g', \alpha')$ be the reachability predicate for the second thread, i.e., $R_2(g, \alpha, g', \alpha')$ holds iff $(g, \alpha) \to_2^* (g', \alpha)$. (As usual, we use unprimed variables for the initial configuration and primed ones for the final configuration.) Using standard logical manipulations we obtain

$$\overline{post}_2^*(T)(g', \alpha_1, \alpha_2') = \exists g : \Big(V_1(g, \alpha_1) \wedge \underbrace{\exists \alpha_2 : V_2(g, \alpha_2) \wedge R_2(g, \alpha_2, g', \alpha_2')}_{=: U_2(g, g', \alpha_2')} \Big) \ .$$

Since g is existentially quantified, $\overline{post}_2^*(T)$ is not always a view tuple. We present a generic approach for representing it as a union of view tuples. The approach is parameterized by a partition of G. We need the following definition.

Definition 2. *Two distinct global values $g_a, g_b \in G$ are* confluent *if there exist $g', \alpha_{2a}', \alpha_{2b}'$ such that $U_2(g_a, g', \alpha_{2a}')$ and $U_2(g_b, g', \alpha_{2b}')$ hold. A partition of G is* safe *if none of its sets contains two confluent values.*

Intuitively, two values in the same set of a safe partition cannot be transformed by the second thread into the same value. For instance, let us return to the example from Section 2.4. If we choose T_0 such that $[\![T_0]\!] = \{(g, a, a), (g', b, a)\}$, then $\overline{post}_2^*(T_0) = \mathcal{C}_0$ because $(g, a, a) \to_2 (g'', a, a)$ and $(g', b, a) \to_2 (g'', b, b)$. In other words we have $U_2 = \{(g, g'', a), (g', g'', b)\}$. Therefore, g and g' are confluent, and any safe partition must keep these two values apart.

Notice that safe partitions always exist, because the partition that splits G into singletons is always safe. However, finding a coarser safe partition is not necessarily straightforward because U_2 may contain infinitely many tuples, and we will show how to deal with this problem later. For the time being, it suffices to point out that *any* safe partition can be used to represent $\overline{post}_2^*(T)$ as a union of view tuples. Let $G_1, \dots, G_m$ be a safe partition of G. We define sets $V'_{11}, \dots, V'_{1m}$ of 1-views and sets $V'_{21}, \dots, V'_{2m}$ of 2-views as follows:

$$V'_{1j}(g', \alpha_1) = \exists g : V_1(g, \alpha_1) \wedge G_j(g) \wedge \exists \alpha'_2 : U_2(g, g', \alpha'_2) \tag{1}$$

$$V'_{2j}(g', \alpha'_2) = \exists g : U_2(g, g', \alpha'_2) \wedge G_j(g) \tag{2}$$

Intuitively, V'_{1j} contains the local configurations of the first thread for which the second thread can reach the local configuration α'_2 while leaving the global variable in state g'. Therefore, if the first thread initially has (g, α_1) as 1-view, it ends with (g', α_1): the local configuration α_1 has not changed, but the value of the global variable has. The intuition behind V'_{2j} is similar.

In the example above, we could choose $G_1 = \{g, g''\}$ and $G_2 = \{g'\}$ as a safe partition. Under this assumption the view tuples (V'_{11}, V'_{21}) and (V'_{12}, V'_{22}) as defined above would represent the sets $\mathcal{C}_1$ and $\mathcal{C}_2$ from Section 2.4, whose union is indeed equal to $\mathcal{C}_0$. The following theorem, whose proof is given in the appendix, states that this works for *every* safe partition.

Theorem 1. *Let $\{V'_{1j}\}_{j\in[m]}$ and $\{V'_{2j}\}_{j\in[m]}$ be defined as in (1) and (2). Then*

$$\overline{post}^*_2(T)(g', \alpha_1, \alpha'_2) = \bigvee_{j=1}^{m} \Big(V'_{1j}(g', \alpha_1) \wedge V'_{2j}(g', \alpha'_2) \Big) \ .$$

Proof. ($\Rightarrow$):

$$\begin{array}{ll}
\overline{post}^*_2(T)(g', \alpha_1, \alpha'_2) & \\
= \exists g : V_1(g, \alpha_1) \wedge U_2(g, g', \alpha'_2) & (\text{def. of } \overline{post}^*_2(T)) \\
= \exists g : V_1(g, \alpha_1) \wedge U_2(g, g', \alpha'_2) \wedge \exists j \in [m] : G_j(g) & \\
= \exists j \in [m] : V'_{1j}(g', \alpha_1) \wedge V'_{2j}(g', \alpha'_2) & (\text{logic, def. of } V'_{1j}, V'_{2j}) \\
\Rightarrow \bigvee_{i=1}^{m} \Big(V'_{1j}(g', \alpha_1) \wedge V'_{2j}(g', \alpha'_2) \Big) &
\end{array}$$

($\Leftarrow$): Let $(g', \alpha_1, \alpha'_2)$ be a triple satisfying $V'_{1j}(g', \alpha_1) \wedge V'_{2j}(g', \alpha'_2)$ for some $j \in [m]$. By the definition of V'_{1j} and V'_{2j} there exist g_a, g_b, and α''_2 such that $V_1(g_a, \alpha_1)$, $G_j(g_a)$, $U_2(g_a, g', \alpha''_2)$, $U_2(g_b, g', \alpha'_2)$, and $G_j(g_b)$ hold. So, in particular, g_a and g_b belong to the same set of the partition of G, namely G_j. Furthermore, since $U_2(g_a, g', \alpha''_2)$, $U_2(g_b, g', \alpha'_2)$, it follows from Definition 2 that g_a and g_b are either confluent or equal. Since the partition used to construct $\{V'_{1j}\}_{j\in[m]}$ and $\{V'_{2j}\}_{j\in[m]}$ is safe, we get $g_a = g_b$. So, in particular, $U_2(g_a, g', \alpha'_2)$ holds, which together with $V_1(g_a, \alpha_1)$ implies $\overline{post}^*_2(T)(g', \alpha_1, \alpha'_2)$.

3.2 Computing the confluence relation

In this part, we show how to compute the relation $C(x, y)$ of confluent pairs x, y symbolically, using BDDs. By Definition 2, we have

$$C(g_a, g_b) = g_a \neq g_b \wedge \exists g', \alpha'_{2a}, \alpha'_{2b} : U_2(g_a, g', \alpha'_{2a}) \wedge U_2(g_b, g', \alpha'_{2b})$$

Notice that the relation U_2 contains stack words and cannot be directly represented by a BDD. However, we first show that U_2 can be represented as a *symbolic* finite automaton and then use the automaton to compute C.

Let us recall the definitions of $U_2(g, g', \alpha_2')$ and $post_2^*(V_2)(g', \alpha_2')$:

$$\begin{aligned} U_2(g, g', \alpha_2') &= \exists \alpha_2 : V_2(g, \alpha_2) \wedge R_2(g, \alpha_2, g', \alpha_2') \\ post_2^*(V_2)(g', \alpha_2') &= \exists g, \alpha_2 : V_2(g, \alpha_2) \wedge R_2(g, \alpha_2, g', \alpha_2') \end{aligned}$$

We now reduce the computation of U_2 to a local reachability problem w.r.t. a modified pushdown system $(G \times G, \Gamma, \Delta_2')$. In other words, we change the system by duplicating the globals. Moreover, we have $\langle(\bar{g}, g), \gamma\rangle \hookrightarrow \langle(\bar{g}, g'), \alpha\rangle$ in Δ_2' iff $\langle g, \gamma\rangle \hookrightarrow \langle g', \alpha\rangle$ in Δ_2, i.e. the value of the first copy is never changed by any transition rule. The reachability relation for the second thread of the modified system is given by $\overline{R}_2((\bar{g}, g), \alpha_2, (\bar{g}', g'), \alpha_2') = R_2(g, \alpha_2, g', \alpha_2') \wedge \bar{g} = \bar{g}'$. Define $\overline{V}_2((\bar{g}, g), \alpha_2) = V_2(g, \alpha_2) \wedge \bar{g} = g$. We have:

$$\begin{aligned} U_2(g, g', \alpha_2') &= \exists \alpha_2 : V_2(g, \alpha_2) \wedge R_2(g, \alpha_2, g', \alpha_2') \\ &= \exists \alpha_2 : \overline{V}_2((g, g), \alpha_2) \wedge \overline{R}_2((g, g), \alpha_2, (g, g'), \alpha_2') \\ &= \exists \bar{\bar{g}}, \bar{g}, \alpha_2 : \overline{V}_2((\bar{\bar{g}}, \bar{g}), \alpha_2) \wedge \overline{R}_2((\bar{\bar{g}}, \bar{g}), \alpha_2, (g, g'), \alpha_2') \\ &= post_2^*(\overline{V_2})((g, g'), \alpha_2') \end{aligned}$$

Recall that if $T = (V_1, V_2)$ is a view tuple, then V_1 and V_2 (and $\overline{V}_2$) are regular sets, representable by *symbolic* finite automata [17]. Moreover, [17] shows how to transform a symbolic automaton for $\overline{V}_2$ into one for $U_2 = post_2^*(\overline{V}_2)$.

We turn to the question how to compute C from the automaton representing U_2. For this, let us define $U_2'(g, g') := \exists \alpha : U_2(g, g', \alpha)$. Then we have:

$$C(g_a, g_b) = g_a \neq g_b \wedge \exists g' : U_2'(g_a, g') \wedge U_2'(g_b, g')$$

The modified pushdown system defined above has $G \times G$ as its set of globals. Thus, the symbolic automaton for U_2 uses $G \times G$ as initial states, and a configuration $((g, g'), \alpha)$ is accepted if, starting at state (g, g'), the automaton can read the input α and end up in a final state. Thus, $U_2'(g, g')$ holds if some input is accepted from the state (g, g'). Since the transitions of a symbolic automaton are represented by BDDs, this can be easily implemented with standard BDD operations.

3.3 Computing a safe partition

Given the confluence relation C, our final goal now is to compute a safe partition. Notice that a partition is safe if and only if its sets are cliques of $\neg C$, the complement of C. Since finding a minimal partition into cliques of a given graph is NP-complete, we restrict ourselves to finding a reasonably coarse safe partition in a symbolic manner. The resulting performance of the reachability algorithm is evaluated in Section 5.

```
Input: Confluence relation $C(x,y)$, total order $L(x,y)$
Output: A safe partition $G_1, \ldots G_m$ of $G$
1  $S(x,y) := \neg C(x,y)$; $j := 0$;
2  while $S \neq \emptyset$ do
3      pick $(x_0, y_0)$ from $S$;
4      $F(x) := S(x, y_0)$;
5      while true do
6          $D(x,y) := L(x,y) \wedge F(x) \wedge F(y) \wedge \neg S(x,y)$;
7          exit if $D = \emptyset$;
8          $F(x) := F(x) \wedge \neg(\exists y : D(x,y))$
9      $j := j + 1$;
10     $G_j(x) := F(x)$;
11     $S(x,y) := S(x,y) \wedge \neg F(x) \wedge \neg F(y)$;
```

Algorithm 2: An algorithm for computing equivalence classes

Algorithm 2 shows the computation of the partition. Its inputs are the confluence relation C and an arbitrary total order relation L on globals. The algorithm repeatedly computes sets of the partition. The inner loop makes sure that F is a clique of S when exiting the loop. D contains the confluent pairs (x_1, x_2) of $F \times F$ such that x_1 is smaller than x_2 with respect to the order L. If $D = \emptyset$ then F is a clique. Otherwise, for each $(x_1, x_2) \in D$ we remove x_1. The rôle of L is to guarantee that D is antisymmetric, and so that if x_1 and x_2 are confluent we remove exactly one of them from F.

Notice that the algorithm only uses boolean operations and existential quantification, and can therefore be easily implemented in a BDD library, given BDD representations of L and C. The computation of C was presented in Section 3.2, and a BDD representation for $L \subseteq G \times G$ is trivial to generate, because by assumption the set G is finite, and any total order (e.g. some lexicographical ordering based on the BDD variables) will do.

Finally, equations (1) and (2) only use G_j, V_1, U_2, which are all representable as BDDs or as symbolic automata, connected by boolean operations. Thus, the new view tuples can be obtained by standard operations on BDDs and automata.

4 Implementation

We implemented the algorithm presented here in jMoped [18, 15], an Eclipse plug-in for testing Java programs by means of model-checking techniques. To test a method, users specify the number of *bits* of the program variables and the *heap* size (no knowledge of model-checking techniques is required). jMoped computes the reachable states of the program for all values of the method's arguments within the given range. During the analysis, jMoped displays progress by labelling lines of code with diverse markers, e.g. red markers for assertion violations, green and black markers for reachable and unreachable lines, respectively.

Like Java virtual machines we use heaps to simulate Java objects. In particular, the heap size determines the number of objects that can be generated.

4.1 The Model

Internally, jMoped operates on pushdown networks that can also contain rules for thread creation (cf. Section 2.2). jMoped uses a symbolic representation of pushdown networks like in [17], where the stack symbols are tuples (l, γ) such that l is a valuation of local variables and γ a label, i.e. a possible value of the program counter. A network is stored as a list of symbolic rules. For instance, given labels $\gamma, \gamma', \gamma''$, the set of all rules of the form $\langle g, (l, \gamma)\rangle \hookrightarrow \langle g', (l', \gamma')(l'', \gamma'')\rangle$ is represented by one single rule annotated with a relation R:

$$\gamma \hookrightarrow \gamma'\gamma'' \quad R(g, l, g', l', l'')$$

R specifies which tuples correspond to a rule and is stored as a BDD.

4.2 The Translator

jMoped analyzes which classes are statically reachable from the starting method, and then translates their bytecodes into a pushdown network. The translation process is relatively simple: in most cases a bytecode instruction is mapped into one symbolic rule. However, the BDD for the symbolic rule is not computed beforehand; we only store the information needed to construct it on-the-fly if needed. Constructing BDDs only on demand saves considerable resources.

We maintain four types of variables when analyzing Java bytecodes. Static variables and local variables are modelled by globals and locals, respectively. Heaps are essentially arrays of globals. When an object is created, it occupies some parts of the array where it keeps relevant information such as fields, object type, and lock information. The object itself can be seen as a *pointer* to the array. Objects are never garbage collected in the current implementation.

The Java virtual machine uses an *operand stack* for each method call. This stack can be loaded with constants or values from local variables or fields. Many instructions pop operands from the stack, operate on them, and push the result back. Operand stacks are also used to prepare parameters for method calls and to receive method results. The maximum depth of the operand stack for a given method is determined at compile time and stored in the corresponding class file. jMoped models operand stacks by arrays of locals plus an extra top-of-stack pointer. The array lengths are equal to the maximum depths of the stacks.

We give a flavour of how jMoped works. Figure 1 shows a small Java program, its bytecodes, and a simplified version of the translation into a pushdown network. Bytecode instructions are translated one-to-one into transition rules. The operand stack is simulated by the array s and the top-of-stack pointer *sp*. Similarly, we use *heap* and *ptr* for the heap and the heap pointer. The top-of-stack and heap pointers are initialized to 0 and 1, respectively. The heap at index zero is reserved for null objects. Local variables have identifiers of the form lv_i.

```
class C {                                   0: iconst_0
    static int x;                           1: putstatic C.x
    static void m() {                       4: new Thread
        x = 0;                              7: dup
        new Thread(new Runnable() {         8: new C$1
            public void run() {            11: dup
                // New thread works        12: invokespecial C$1.<init>
        }}).start();                       15: invokespecial Thread.<init>
        // Main thread works               18: invokevirtual Thread.start
    }                                      21: ...
}                                           e: return
```

$m_0 \hookrightarrow m_1$	$(\mathrm{s}[\mathrm{sp}]' = 0 \wedge \mathrm{sp}' = \mathrm{sp} + 1)$
$m_1 \hookrightarrow m_4$	$(\mathrm{x}' = \mathrm{s}[\mathrm{sp} - 1] \wedge \mathrm{sp}' = \mathrm{sp} - 1)$
$m_4 \hookrightarrow m_7$	$(\mathrm{s}[\mathrm{sp}]' = \mathrm{ptr} \wedge \mathrm{sp}' = \mathrm{sp} + 1 \wedge \mathrm{heap}[\mathrm{ptr}]' = 1 \wedge \mathrm{ptr}' = \mathrm{ptr} + 2)$
$m_7 \hookrightarrow m_8$	$(\mathrm{s}[\mathrm{sp}]' = \mathrm{s}[\mathrm{sp} - 1] \wedge \mathrm{sp}' = \mathrm{sp} + 1)$
$m_8 \hookrightarrow m_{11}$	$(\mathrm{s}[\mathrm{sp}]' = \mathrm{ptr} \wedge \mathrm{sp}' = \mathrm{sp} + 1 \wedge \mathrm{heap}[\mathrm{ptr}]' = 2 \wedge \mathrm{ptr}' = \mathrm{ptr} + 1)$
$m_{11} \hookrightarrow m_{12}$	$(\mathrm{s}[\mathrm{sp}]' = \mathrm{s}[\mathrm{sp} - 1] \wedge \mathrm{sp}' = \mathrm{sp} + 1)$
$m_{12} \hookrightarrow c_0\ m_{15}$	$(\mathrm{lv}_0' = \mathrm{s}[\mathrm{sp} - 1] \wedge \mathrm{sp}'' = \mathrm{sp} - 1)$
$m_{15} \hookrightarrow t_0\ m_{18}$	$(\mathrm{lv}_0' = \mathrm{s}[\mathrm{sp} - 2] \wedge \mathrm{lv}_1' = \mathrm{s}[\mathrm{sp} - 1] \wedge \mathrm{sp}'' = \mathrm{sp} - 2)$
$m_{18} \hookrightarrow m_{21} \rhd r_0$	$(\mathrm{heap}[\mathrm{heap}[\mathrm{s}[\mathrm{sp} - 1] + 1]] = 2 \wedge \mathrm{lv}_0'' = \mathrm{s}[\mathrm{sp} - 1] \wedge \mathrm{sp}' = \mathrm{sp} - 1)$
...	...
$m_e \hookrightarrow \epsilon$	

Fig. 1. A small Java programs, its bytecodes, and a corresponding pushdown network

At the beginning of `m`, the global variable `x` is initialized to zero in two steps. The constant 0 is pushed onto the operand stack, retrieved, and stored in `x`. Then, a new object of type `Thread` is created, and a reference to the object is pushed onto the operand stack. jMoped simulates this behaviour by pushing the current value of the heap pointer and updating it to a next (empty) heap element. The heap at `ptr` is also set to the object type of `Thread`, which is 1 in this example. We update the heap pointer based on sizes of objects. Every object needs one heap element for each field plus an extra heap element for storing its type. The object for `Thread` has size two (see later), and so the pointer gets updated by two. The instruction `dup` duplicates the top element of the operand stack. At offset `8`, an object of type `C$1` is allocated. Class `C$1` is an inner class of `C` which implements the interface `Runnable`. `C$1` specifies the method `run` which will be executed when the thread starts. Note that `C$1` has type 2 and size 1.

Two initialization methods are called at offsets `12` and `15` for `C$1` and `Thread`, respectively. The corresponding translation also shows how arguments are passed. A reference to `C$1` (resp. to `Thread`) is passed to lv_0 when initializing `C$1` (resp. `Thread`). However, for `Thread` a reference to `C$1` is also passed as the second argument, and a reference to `C$1` is stored as its only field (not shown). Recall that `Thread` has size 2 for the purpose of storing an object reference which implements `Runnable` interface. This information is used later on at offset `18`.

There, we fork a new thread r_0 if the only field of the thread specified by the top element of the operand stack has type 2. Also, a reference to `C$1` is passed to the new thread. Note that we need information about object types to start the right thread. The same technique is used in the case of virtual method calls.

jMoped translates *all* Java bytecode instructions. Calls to Java libraries are not replaced by stubs, since the bytecodes of the library are available. Notice however that some classes contain native code, and for those stubs are necessary.

5 Experiments

All experiments were performed on an AMD 3 GHz machine with 64 GB memory.

5.1 `java.util.Vector` Class

In this experiment we consider class `java.util.Vector` from the Java library. The `Vector` class implements a growable array of objects. In [19], a race condition in a constructor of `Vector` was reported. The following test method illustrates the situation where the race condition can occur.

```
static void test(Integer x) {
    final Vector<Integer> v1 = new Vector<Integer>();
    v1.add(x);
    new Thread(new Runnable() {
        public void run() { v1.removeAllElements(); }
    }).start();
    Vector<Integer> v2 = new Vector<Integer>(v1);
    assert(v2.isEmpty() || v2.elementAt(0) == x);
}
```

The method creates two vectors. First an empty vector `v1` is created, and then an integer `x` is added to it as its first element. After that, a new thread is forked, which removes all elements from `v1` (only `x` in this case). In parallel, the first thread creates a copy `v2` of `v1`. Intuitively, only two cases are possible: if the elements of `v1` are removed before `v2` is created, then `v2` is empty; if `v2` is created before the elements of `v1` are removed, then the first element of `v2` is equal to `x`. The last line of code asserts this property.

However, in Java 5.0, the constructor of `v2` is not atomic, and as a result the assertion can be violated. jMoped detects this bug. The first half of Table 1 shows the time until the bug is found, the numbers of BDD nodes required, and the numbers of view tuples inspected in several experiments. In all experiments the bit size of all variables except `x` is set to 8, the heap size to 50 blocks, and the context bound to 3. The experiments differ on the size of `x` (1 to 8 bits), and on the splitting mode (eager or lazy).

In the current version of Java (version 6.0), the bug has been fixed. We reran all experiments with Java 6.0 and verified that, within the given bounds, the assertion is not violated. The second half of Table 1 presents the results.

Sizes of x (bits)			1	2	3	4	5	6	7	8
Java 5.0	Eager	Time (s)	9.3	10.8	16.9	31.1	67.9	117.8	225.7	457.9
		Nodes ($\times 10^6$)	0.4	0.5	0.8	1.4	2.5	5.2	9.0	18.1
		View tuples	48	87	167	327	648	1348	2567	5126
	Lazy	Time (s)	19.7	17.7	19.6	17.5	17.2	18.9	16.7	18.8
		Nodes ($\times 10^6$)	1.2	1.2	1.2	1.3	1.2	1.3	1.3	1.3
		View tuples	3	3	3	3	3	3	3	3
Java 6.0	Eager	Time (s)	15.1	18.6	37.5	64.3	147.7	301.7	642.0	1732.0
		Nodes ($\times 10^6$)	0.4	0.7	1.1	2.0	3.7	7.1	13.9	27.9
		View tuples	105	209	417	833	1655	3329	6657	13313
	Lazy	Time (s)	20.9	20.8	19.4	22.3	20.8	18.8	23.4	23.2
		Nodes ($\times 10^6$)	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3
		View tuples	3	3	3	3	3	3	3	3

Table 1. Experimental results: `java.util.Vector` class

The behaviour of the program is independent of the value of `x`. The lazy approach benefits from this, and does not split at all when switching contexts. Therefore, the running time remains essentially constant when the number of bits of `x` increases. On the other hand, the time for eager splitting increases exponentially. However, the eager approach is faster and requires fewer BDD nodes when `x` is small. One of the reasons is that the lazy approach requires an extra copy of globals for keeping relations between current values of globals and initial values when the thread is awakened, which results in bigger BDDs.

One could argue that, since the Java 5.0 bug is already detected when `x` has 1 bit, the lazy approach does not give any advantage in this case. For Java 6.0, however, the analysis of larger ranges provides more confidence in the correctness of the code, and here the lazy approach clearly outperforms eager splitting.

Finally, we remark that the example is not as small as it seems. While the test method has only a few lines of code, the class `Vector` actually involves around 130 classes which together translate into a pushdown network of 30,000 rules. We are able to automatically translate all classes without any manipulations except `java.lang.System`, where the method `arraycopy` is implemented in native code. We need to manually create a stub in this case.

5.2 Windows NT Bluetooth Driver

In this experiment, we consider three versions of a Windows NT Bluetooth driver [20, 9]. Figure 2 shows a Java implementation of the second version. All three versions follow the same idea and differ only in some implementation details. All versions use the following class `Device`, which contains four fields:

```
int pendingIo; boolean stopFlag, stopEvent, stopped;
Device(){ pendingIo = 1; stopFlag = stopEvent = stopped = false; }
```

– `pendingIo` counts the number of threads that are currently executing in the driver. It is initialized to one in the constructor, increased by one when a new thread enters the driver, and decreased by one when a thread leaves.
– `stopFlag` becomes true when a thread tries to stop the driver.
– `stopEvent` models a stopping event, fired when `pendingIo` becomes zero. The field is initialized to false and set to true when the event happens.
– `stopped` is introduced only to check a safety property. Initially false, it is set to true when the driver is successfully stopped.

The drivers has two types of threads, *stoppers* and *adders*. A stopper calls `stop` to halt the driver. It first sets `stopFlag` to true before decrementing `pendingIo` via a call to `dec`. The method `dec` fires the stopping event when `pendingIo` is zero. An adder calls the method `add` to perform I/O in the driver. It calls the method `inc` to increment `pendingIo`; `inc` returns a successful status if `stopFlag` is not yet set. It then asserts that `stopped` is false before start performing I/O in the driver. The adder decrements `pendingIo` before exiting.

```
static void add(Device d) {
    int status = inc(d);
    if (status > 0) {
        assert(!d.stopped);
        // Performs I/O
    }
    dec(d);
}
static void stop(Device d) {
    d.stopFlag = true;
    dec(d);
    while (!d.stopEvent) {}
    d.stopped = true;
}
static int inc(Device d) {
    int status;
    synchronized(d) {
        d.pendingIo++;
    }
    if (d.stopFlag) {
        dec(d);
        status = -1;
    } else status = 1;
    return status;
}
static void dec(Device d) {
    int pio;
    synchronized (d) {
        d.pendingIo--;
        pio = d.pendingIo;
    }
    if (pio == 0)
        d.stopEvent = true;
}
```

Fig. 2. Version 2 of Bluetooth driver

In the first version of the driver, the method `inc` was implemented differently:

```
private int inc(Device d) {
    if (d.stopFlag) return -1;
    synchronized (d) { d.pendingIo++; }
    return 0;
}
```

Moreover, the if-statement in `add` reads `if (status == 0)`. [20] reports a race condition for this version, which occurs when the adder first runs until it checks

	Version 1		Version 2		Version 3	
	Eager	Lazy	Eager	Lazy	Eager	Lazy
Time (s)	1.1	1.3	51.7	36.0	11.9	6.0
Nodes ($\times 10^3$)	46	88	720	1851	195	518
View tuples	21	4	1460	154	234	16
Contexts	3		5		4	

Threads, Contexts	Time(s)
$1+1,3$	3.8
$1+1,4$	8.3
$2+1,4$	127.1
$2+1,5$	712.3
$2+1,6$	5528.2
$2+2,5$	6488.0
$2+2,6$	timeout

Table 2. Experimental results: Bluetooth drivers (left) and binary search trees (right)

the value of `stopFlag`. Then, the stopper thread runs until the end, where it successfully stops the driver. When the context switches back to the adder, it returns from `inc` with status zero and finds out that the assertion is violated.

In [9] a bug in the second version of the driver was reported. The bug only occurs in the presence of at least two adders, and four context switches are required to unveil it: (i) The first adder increases `pendingIo` to 2 and halts just before the assertion statement. (ii) The stopper sets `stopFlag` to true, decreases `pendingIo` back to 1, and waits for the stopping event. (iii) The second adder increases `pendingIo` to 2. However, since `stopFlag` is already set it decreases `pendingIo` back to 1 again. It returns from `inc` with status -1, which makes `pendingIo` become 0 and fires the stopping event. (iv) The stopper acknowledges the stopping event and sets `stopped` to true. (v) The first adder violates the assertion. Note that the bug can also be found in a slightly different manner where the second adder starts before the stopper.

The third version moves `dec(d)` inside the if-block in the method `add`. This eliminates the bug for the case with two adders and one stopper. However, jMoped found another assertion violation for one adder and two stoppers. We believe that this has not been previously reported, although it is less subtle than the previous bugs, requiring three context switches: (i) The adder increases `pendingIo` to 2 and halts just before the assertion statement. (ii) The first stopper decreases `pendingIo` to 1. (iii) The second stopper decreases `pendingIo` to 0 and sets `stopped` to true. (iv) The adder violates the assertion.

Table 2 reports experimental results on these three versions. Notice that the lazy approach always involves fewer view tuples. This becomes more obvious when the number of contexts grows. We argue that by splitting lazily we can palliate explosions in the context-bounded reachability problem.

5.3 Binary Search Trees

We briefly give an intuition on the scalability of our approach by considering a binary search tree implementation [21] that supports concurrent manipulations on trees. Unlike the previous two experiments, this algorithm is recursive. There are two types of threads, *inserter* and *searcher*. An inserter puts a node into the tree while a searcher looks for a node with a given value. We consider the situation where inserters insert non-deterministic values into the tree and searchers search for the same values. We run jMoped with different numbers of inserters and searchers, and generate all reachable configurations within given contexts.

Table 2 gives the running times. The numbers of threads are in the form $x + y$, where x and y are the numbers of inserters and searchers, respectively. The analysis took more than three hours in the case of $2 + 2, 6$.

6 Conclusions

We have reported on (to the best of our knowledge) the first implementation of the context-bounded technique of Qadeer and Rehof [11]. The implementation extends the jMoped tool, and allows to deal directly with Java code, mostly without having to manually manipulate it or replace Java libraries by stubs.

The algorithm for context-bounded reachability as presented in [11] explicitly deals with each possible trace of the system within the context bound. Therefore, if the number of traces is exponential, then the algorithm *necesarily* takes exponential time. We have presented a symbolic technique, lazy splitting, to palliate this problem. Loosely speaking, the technique tries to symbolically examine all traces in the same computation, and splits the set of traces only when necessary.

We have tested our implementation on a number of examples. Our best result so far is the fact that we can find the bug in the `Vector` Java class reported in [19] without any need for manual manipulation or unsound steps. The program is compiled including all Java libraries, and the Java bytecode is automatically translated into our formal model, without manual supervision.

Lal et al. have proposed a new algorithm which does not require to explicitly examine all possible traces of the system [14]. The main idea is to compute for each thread a regular transducer accepting the reachability relation of the thread. Even though this leads to an attractive fully symbolic solution to the problem, its performance in practice still needs investigation. Even for finite-state systems, experiments show that the symbolic computation of the reachability relation by means of iterative squaring is substantially more expensive than the computation of the set of reachable states (see for instance [22]). While in the case of multithreaded programs the advantage of a fully symbolic procedure may compensate for the overhead of computing the reachability relation, this remains to be seen. We have not followed this path because the algorithm for the computation of the reachable states is the core of jMoped, and the result of many optimizations, and so naturally we wished to reuse it.

On a more abstract level, the idea of [14] is that the effect of running a context on a particular thread can be expressed by a summary. The idea of reusing summaries could also be useful in our setting: as a side effect, the pushdown reachability algorithm implementing the $post_i^*$ function computes (partial) procedure summaries. Summaries are largely independent of the context for which they were computed and could potentially be re-used many times during other called to $post_i^*$. We did not yet use this trick in our implementation.

Acknowledgements The authors thank Tomáš Brázdil for his helpful comments and Tayssir Touili and Mihaela Sighireanu for pointing us to some examples.

References

1. Ramalingam, G.: Context-sensitive synchronisation-sensitive analysis is undecidable. ACM Trans. Programming Languages and Systems **22** (2000) 416–430
2. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: Proc. CONCUR. LNCS 3653 (2005) 473–487
3. Kahlon, V., Ivančić, F., Gupta, A.: Reasoning about threads communicating via locks. In: Proc. CAV. LNCS 3576 (2005) 505–518
4. Kahlon, V., Gupta, A.: On the analysis of interacting pushdown systems. In: Proc. POPL, ACM (2007) 303–314
5. Qadeer, S., Rajamani, S.K., Rehof, J.: Summarizing procedures in concurrent programs. In: Proc. POPL, ACM (2004) 245–255
6. Sen, K., Viswanathan, M.: Model checking multithreaded programs with asynchronous atomic methods. In: Proc. CAV. LNCS 4144 (2006) 300–314
7. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: Proc. POPL, ACM Press (2003) 62–73
8. Bouajjani, A., Esparza, J., Touili, T.: Reachability analysis of synchronized PA-systems. In: Proc. Infinity. (2004)
9. Chaki, S., Clarke, E.M., Kidd, N., Reps, T., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: Proc. TACAS. LNCS 3920 (2006) 334–349
10. Patin, G., Sighireanu, M., Touili, T.: Spade: Verification of multithreaded dynamic and recursive programs. In: Proc. CAV. LNCS 4590 (2007) 254–257
11. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Proc. TACAS. LNCS 3440 (2005) 93–107
12. Bouajjani, A., Esparza, J., Schwoon, S., Strejček, J.: Reachability analysis of multithreaded software with asynchronous communication. In: Proc. FSTTCS. LNCS 3821 (2005) 348–359
13. La Torre, S., Madhudusan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Proc. TACAS. LNCS 4963 (2008) 299–314
14. Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural analysis of concurrent programs under a context bound. In: Proc. TACAS. LNCS 4963 (2008) 282–298
15. jMoped: The tool's website (http://www7.in.tum.de/tools/jmoped/)
16. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Proc. CAV. LNCS 1855 (2000) 232–247
17. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Proc. CAV. LNCS 2102 (2001) 324–336
18. Suwimonteerabuth, D., Berger, F., Schwoon, S., Esparza, J.: jMoped: A test environment for Java programs. In: Proc. CAV. LNCS 4590 (2007) 164–167
19. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Software Eng. **32**(2) (2006) 93–110
20. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: PLDI. (2004) 14–24
21. Kung, H.T., Lehman, P.L.: Concurrent manipulation of binary search trees. ACM Trans. Database Syst. **5**(3) (1980) 354–382
22. Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L., Dill, D.L.: Symbolic model checking for sequential circuit verification. IEEE TCAD **13**(4) (1994) 401–424