Assume-Guarantee Model Checking

Cormac Flanagan[1] and Shaz Qadeer[2]

[1] Systems Research Center, HP Labs, 1501 Page Mill Road, Palo Alto, CA 94304
[2] Microsoft Research, One Microsoft Way, Redmond, WA 98052

Abstract. We present assume-guarantee model checking, a novel technique for verifying correctness properties of *loosely-coupled* multithreaded software systems. Assume-guarantee model checking verifies each thread of a multithreaded system separately by constraining the actions of other threads with an automatically inferred environment assumption. Separate verification of each thread allows the enumeration of the local state of only one thread at a time, thereby yieldig significant savings in the time and space needed for model checking. Suppose G is the size of the global store, L the size of the local store per thread, and n the number of threads. If each thread is finite-state (without a stack), the naive model checking algorithm is $O(n.G^2.L^{n+1})$ whereas assume-guarantee model checking is $O(n.G^2.L.(n + L))$. If each thread has a stack, the reachability problem is undecidable. However, assume-guarantee model checking terminates in time $O(n.G^3.L^3.F)$ where F is the number of stack symbols.

1 Introduction

Designing correct multithreaded software is difficult due to subtle interactions among threads operating concurrently on shared data. Errors in such systems are easy to introduce but difficult to diagnose and fix. Model checking [CE81,QS81] is a promising technique for verifying correctness properties of multithreaded software systems. However, due to the large state spaces of such systems, they are difficult to model check. In this paper, we present a novel technique called assume-guarantee model checking to alleviate the problem of exploring large state spaces of multithreaded software.

We consider multithreaded software systems with a finite number of threads where the shared global store and the local store of each thread are finite. However, each thread also has an unbounded stack which allows us to model procedure calls and recursion. We focus on the verification of safety properties such as assertions and global invariants. Verification of such safety properties can be reduced to the problem of checking whether an error state is reachable from the system's initial state. This problem is undecidable [HU79,Ram00] in general. Assume-guarantee model checking is a conservative (sound and incomplete) algorithm for this problem that is powerful enough to verify a variety of multithreaded software systems occurring in practice.

Assume-guarantee reasoning for shared-memory programs was first introduced by Jones [Jon83]. The basic idea behind this technique is to verify each thread separately using an environment assumption to model interleaved steps of the other threads. The environment assumption of each thread is a binary relation over the set of global stores, and includes all global store updates that may be performed by other threads.

In earlier work, we extended the assume-guarantee proof rule of Jones and implemented it in the Calvin checker [FFQ02,FQS02] for multithreaded Java programs. Our experience using Calvin indicates that the threads in most software systems are *loosely-coupled*, *i.e.*, there is little correlation among the local states of the various threads, and assume-guarantee reasoning is sufficiently powerful to verify these systems. However, a significant cost of using Calvin is that the programmer is required to provide the appropriate environment assumption. The assume-guarantee model checking technique in this paper avoids this cost by automatically inferring these environment assumptions.

Assume-guarantee model checking infers the environment assumption for each thread by first infering a *guarantee* for each thread, which models all global store updates performed by that thread. The environment assumption of a thread is then the disjunction of the guarantees of all the other threads. The guarantee of each thread is initially the empty relation, and is iteratively extended during the model checking process. Each thread is verified using the standard algorithm for model checking a sequential pushdown system except that at each control point of the thread, the global state is allowed to mutate according to the guarantees of the other threads. In addition, whenever a thread modifies the global store, that transition on the global states is added to that thread's guarantee. The iteration continues until the reachable state space and guarantee of each thread converges. The complexity of this procedure is $O(n.G^3.L^3.F)$, where n is the number of threads, F is the number of stack symbols, G is the size of the global store, and L is the size of local store per thread.

Even if the threads do not have a stack and are consequently finite-state, assume-guarantee model checking offers significant savings over standard model checking. The naive model checking algorithm explicitly models the program counters of all threads. Therefore, it explores all interleavings of the various threads and its complexity is exponential in the number of threads. However, assume-guarantee model checking verifies each thread separately and its complexity $O(n.G^2.L.(n{+}L))$ is significantly better than that of the naive algorithm.

1.1 Example

To illustrate the benefits of assume-guarantee model checking, we consider its application to a simple multithreaded program. The multithreaded program $\mathit{Simple}(n)$ has n threads which are concurrently executing the procedure `p`. Each thread is identified by unique integer value from the set $\mathit{Tid} = \{1, \ldots, n\}$. These threads manipulate a shared integer variable `x` initialized to 1. The variable `x` is protected by a mutex `m`, which is either the (non-zero) identifier of the thread holding the lock, or else 0, if the lock is not held by any thread. Thus, the type `Mutex` $= \{0\} \cup \mathit{Tid}$. The mutex `m` is manipulated by two operations, `acquire` and `release`. The operation `acquire` blocks until `m` $= 0$ and then atomically sets `m` to *tid*, the identifier of the current thread. The operation `release` sets `m` back to 0. For each thread, there is an implicit local variable called `pc`, which is the program counter of the thread. The variable `pc` takes values from the set $\texttt{Loc} = \{1, \ldots, 6\}$ of control locations.

A simple multithreaded program

```
int x := 1;
Mutex m := 0;

    void p() {
1:    acquire;
2:    x := 0;
3:    x := x + 1;
4:    assert x > 0;
5:    release;
6:  }
```

$$Simple(n) = \underbrace{\texttt{p()} \mid \cdots \mid \texttt{p()}}_{n \text{ times}}$$

We would like to verify three correctness properties for the program $Simple(n)$.

1. There are no races on the data variable `x`.
2. The assertion at control location 4 does not fail for any thread.
3. Every reachable state satisfies the invariant $\texttt{m} = 0 \Rightarrow \texttt{x} = 1$.

Assume-guarantee model checking can verify these correctness properties. Our algorithm computes the guarantee

$$\mathcal{G} \subseteq Tid \times (\texttt{Mutex} \times \texttt{int}) \times (\texttt{Mutex} \times \texttt{int})$$

where $\texttt{Mutex} \times \texttt{int}$ is the set of all global stores, and the thread-local reachable set

$$\mathcal{R} \subseteq Tid \times (\texttt{Mutex} \times \texttt{int}) \times \texttt{Loc}.$$

The set $\mathcal{G}$ has the property that if the thread with identifier *tid* ever takes a step in which the pair $(\texttt{m}, \texttt{x})$ of global variables is modified from (m_1, x_1) to (m_2, x_2), then $(tid, (m_1, x_1), (m_2, x_2)) \in \mathcal{G}$. The set $\mathcal{R}$ has the property that if there is a reachable state in which the pair $(\texttt{m}, \texttt{x})$ has the value (m, v) and the program counter of thread with identifier *tid* has the value *pc*, then $(tid, (m, x), pc) \in \mathcal{R}$.

These sets are given by the following predicates in which `pc` denotes the program counter of the thread with identifier *tid*.

$$\mathcal{G} \stackrel{\text{def}}{=} \begin{array}{l} \vee\ \texttt{m} = 0 \wedge \texttt{m}' = tid \wedge \texttt{x} = \texttt{x}' = 1 \\ \vee\ \texttt{m} = tid \wedge \texttt{m}' = 0 \wedge \texttt{x} = \texttt{x}' = 1 \\ \vee\ \texttt{m} = \texttt{m}' = tid \wedge \texttt{x} = 0 \wedge \texttt{x}' = 1 \\ \vee\ \texttt{m} = \texttt{m}' = tid \wedge \texttt{x} = 1 \wedge \texttt{x}' = \{0, 1\} \end{array}$$

$$\mathcal{R} \stackrel{\text{def}}{=} \begin{array}{l} \vee\ \texttt{pc} \in \{1, 6\} \wedge \texttt{m} = 0 \wedge \texttt{x} = 1 \\ \vee\ \texttt{pc} \in \{1, 6\} \wedge \texttt{m} \in Tid \setminus \{tid\} \wedge \texttt{x} \in \{0, 1\} \\ \vee\ \texttt{pc} \in \{2, 4, 5\} \wedge \texttt{m} = tid \wedge \texttt{x} = 1 \\ \vee\ \texttt{pc} = 3 \wedge \texttt{m} = tid \wedge \texttt{x} = 0 \end{array}$$

The environment assumption of the thread *tid* can be computed from the guarantee as follows:

$$\mathcal{E}(tid) \stackrel{\text{def}}{=} \exists t \in Tid : t \neq tid \wedge \mathcal{G}[tid := t]$$

An examination of $\mathcal{R}$ proves that $Simple(n)$ satisfies its three correctness properties:

1. The thread with identifier *tid* accesses `x` only when its program counter $\texttt{pc} \in \{2,3,4\}$. Every member of $\mathcal{R}$ satisfies the property that if $\texttt{pc} \in \{2,3,4\}$ then $\texttt{m} = tid$. Therefore, it is impossible for two different threads to be at a control location in $\{2,3,4\}$ simulaneously. Consequently, there is no race on the variable `x`.
2. Every member of $\mathcal{R}$ satisfies the property that $\texttt{m} = 1$ when $\texttt{pc} = 4$. Therefore, the assertion at control location 4 holds.
3. Every member of $\mathcal{R}$ satisfies the predicate $\texttt{m} = 0 \Rightarrow \texttt{x} = 1$, which is therefore an invariant of $Simple(n)$.

To verify the program $Simple(n)$, the assume-guarantee model checking algorithm analyzes each thread separately. When analyzing thread *tid*, each global state stored by the algorithm contains values for `m`, `x`, and the program counter of thread *tid*. The algorithm explores $O(n)$ states and transitions for each thread. Since there are n threads, the number of explored states and transitions is $O(n^2)$.

On the other hand, each state stored by a naive model checking algorithm will provide values for `m`, `x`, and the program counters of all the threads. Consequently, the number of states and transitions explored are $O(2^n)$. Thus, for this example, the assume-guarantee model checking algorithm provides exponential savings in the time and space required for state-space enumeration.

Our model of the mutex `m` is an important reason for the success of assume-guarantee model checking on this example. Although a mutex can be modeled as a single bit, we chose to model `m` as a variable whose value indicates the identifier of the thread that currently holds the mutex. This mutex model is crucial for inferring thread guarantees and environment assumptions that are strong enough to verify each thread separately.

1.2 Related work

We refer the reader to our earlier papers [FFQ02,FQS02] for a discussion of the related work on verification of multithreaded software by compositional reasoning and model checking.

Cobleigh et al. [CGP03] share our motivation of reducing the annotation cost of compositional reasoning. They use a counterexample-guided learning algorithm to infer environment assumptions, an approach that is very different from ours. Our algorithm is based entirely on model checking; the correctness properties of the program are verified and appropriate environment assumptions are inferred solely by state-space enumeration.

Bouajjani et al. [BET03] provide several methods for computing abstractions of context-free languages. These abstractions have the property that it is possible to decide whether their intersection is empty. Thus, their work provides a sound but incomplete procedure for verifying multithreaded software. The guarantee of a thread computed by our algorithm is also an abstraction of the thread's program. However, our compositional method for computing the abstraction is very different from their methods.

2 Concurrent finte-state systems

A *concurrent finite state system* consists of a number of concurrently executing threads. The threads communicate through a global store, which is shared by all threads. In addition, each thread has its own local store containing data not manipulated by other threads, such as the program counter of the thread. Each thread also has an associated thread identifier. A state of the system consists of a global store g and a mapping ls from thread identifiers to local stores. We use the notation $ls[t := l]$ to denote a mapping that is identical to ls except that it maps thread identifier t to local store l.

Domains

$$
\begin{aligned}
t, e &\in \mathit{Tid} = \{1, \ldots, n\} \\
g &\in \mathit{GlobalStore} \\
l &\in \mathit{LocalStore} \\
ls &\in \mathit{LocalStores} = \mathit{Tid} \to \mathit{LocalStore} \\
\Sigma &\in \mathit{State} = \mathit{GlobalStore} \times \mathit{LocalStores}
\end{aligned}
$$

We model the behavior of the individual threads as the transition relation T:

$$T \subseteq \mathit{Tid} \times (\mathit{GlobalStore} \times \mathit{LocalStore}) \times (\mathit{GlobalStore} \times \mathit{LocalStore})$$

The relation $T(t, g, l, g', l')$ holds if the thread t can take a step from a state with global store g and where thread t has local store l, yielding a new state with global and local stores g' and l', respectively.

We assume that program execution starts in an initial state $\Sigma_0 = (g_0, ls_0)$ consisting of an initial global store g_0 and a mapping ls_0 that provides the initial local store for each thread. The correctness condition for the program in our system is provided by an *error predicate* $E \subseteq \mathit{Tid} \times \mathit{GlobalStore}\mathit{LocalStore}$. The error predicate E allows us to uniformly model both invariants and thread-local assertions. A state (g, ls) is *erroneous* if there exists $t \in \mathit{Tid}$ such that $E(t, g, ls(t))$ is true. Our goal is to determine if, when started from the initial state Σ_0, the system can reach an erroneous state.

2.1 Standard model checking

Since the set of possible states is finite, we can use standard model checking to determine if any erroneous state is reachable from the initial state. In particular, the least solution $R \subseteq \mathit{State}$ to the following inference rules describes the set of reachable states.

Standard model checking

(BASIC INIT)

$$\frac{}{R(g_0, ls_0)}$$

(BASIC STEP)

$$\frac{R(g, ls) \qquad T(t, g, ls(t), g', l')}{R(g', ls[t := l'])}$$

Although we provide a declarative definition of R here, it is easily computed using a worklist-based algorithm. Having computed R, it is straightforward to determine if any erroneous state is reachable, *i.e.*, if there exist t, g, and ls such that $R(g, ls) \wedge E(t, g, ls)$.

Unfortunately, the computational cost of this algorithm becomes excessive in the presence of multiple threads. Let $n = |\mathit{Tid}|$ be the number of threads and let $G = |\mathit{GlobalStore}|$ and $L = |\mathit{LocalStore}|$ be the sizes of the global and local stores, respectively. Then the size of R is $G.L^n$. Furthermore, for each entry in R there may be $n.G.L$ applications of (BASIC STEP). Hence the time complexity of this algorithm is $O(n.G^2.L^{n+1})$.

2.2 Assume-guarantee model checking

The complexity of standard model checking is exponential in the number of threads, since it explicitly correlates the local states (and program counters) of all the various threads. However, since the threads in most software systems are predominantly loosely-coupled, this correlation is largely redundant. Assume-guarantee model checking provides a means to avoid this redundancy.

Under assume-guarantee model checking, each thread is checked separately, using the guarantees that abstract the behavior of interleaved steps of other threads. The algorithm works by computing two relations: $\mathcal{R}$, which specifies the reachable states of each thread, and $\mathcal{G}$, which is the guarantee of each thread. Thus, the guarantee is inferred automatically during the model checking process.

$$\mathcal{R} \subseteq \mathit{Tid} \times \mathit{GlobalStore} \times \mathit{LocalStore}$$
$$\mathcal{G} \subseteq \mathit{Tid} \times \mathit{GlobalStore} \times \mathit{GlobalStore}$$

The relation $\mathcal{R}(t, g, l)$ holds if the system can reach a state with global store g and where the thread t has local store l. Similarly, $\mathcal{G}(t, g, g')$ holds if a step by thread t can go from a reachable state with global store g to a state with global store g'. While model checking a thread with identifier different from t, we know that whenever the global store is g and $\mathcal{G}(t, g, g')$ holds, an interleaved step of thread t can change the global store to g'. The relations $\mathcal{R}$ and $\mathcal{G}$ are defined as the least solution to the following rules.

Assume-guarantee model checking

$$\text{(AG INIT)}\quad \frac{}{\mathcal{R}(t, g_0, ls(t))} \qquad \text{(AG ENV)}\quad \frac{\mathcal{R}(t,g,l) \quad \mathcal{G}(e,g,g') \quad t \neq e}{\mathcal{R}(t,g',l)} \qquad \text{(AG STEP)}\quad \frac{\mathcal{R}(t,g,l) \quad T(t,g,l,g',l')}{\mathcal{R}(t,g',l') \quad \mathcal{G}(t,g,g')}$$

The set of reachable states determined using assume guarantee reasoning is a conservative approximation of the set of actual reachable states, as illustrated by the following lemma.

Lemma 1. *For all global stores g and local store maps ls, if $R(g, ls)$ then for all thread identifiers t, $\mathcal{R}(t, g, ls(t)))$.*

In particular, if software error causes an erroneous state to be reachable, *i.e.*,

$$\exists t, g, ls.\ (R(g, ls) \wedge E(t, g, ls(t)))\ ,$$

then the assume guarantee algorithm will catch that error, *i.e.*

$$\exists t, g, l.\ (\mathcal{R}(t, g, l) \wedge E(t, g, l))\ .$$

Assume guarantee model checking can be performed using a worklist-based algorithm, whose complexity is much less than that of standard model checking. There may be $n^2.G^2.L$ applications of (AG ENV) and $n.G^2.L^2$ applications of (AG STEP). Hence the time complexity of this algorithm is $O(n.G^2.L.(n + L))$.

3 Concurrent pushdown systems

The assume-guarantee approach described so far works well for checking multithreaded finite state software systems. However, its applicability to realistic systems is somewhat limited, because such systems are typically constructed using procedures and procedure calls, and hence rely on the presence of an unbounded stack for each thread. In this section, we extend our assume-guarantee approach to handle such systems.

We assume that, in addition to a local store, each thread now also has its own private stack, which is sequence of frames. We leave the exact structure of each frame unspecified, but it might contain, for example, the return address for a procedure call. A state of the concurrent pushdown system consists of a global store, a collection of local stores, one for each thread, and a collection of stacks, one for each thread.

Domains

$$\begin{aligned} f &\in \mathit{Frame} \\ s &\in \mathit{Stack} = \mathit{Frame}^* \\ ss &\in \mathit{Stacks} = \mathit{Tid} \rightarrow \mathit{Stack} \\ \Sigma &\in \mathit{State} = \mathit{GlobalStore} \times \mathit{LocalStores} \times \mathit{Stacks} \end{aligned}$$

We model the behavior of the individual threads using three relations:

$$\begin{aligned} T &\subseteq \mathit{Tid} \times (\mathit{GlobalStore} \times \mathit{LocalStore}) \times (\mathit{GlobalStore} \times \mathit{LocalStore}) \\ T^+ &\subseteq \mathit{Tid} \times (\mathit{GlobalStore} \times \mathit{LocalStore}) \times (\mathit{LocalStore} \times \mathit{Frame}) \\ T^- &\subseteq \mathit{Tid} \times (\mathit{GlobalStore} \times \mathit{LocalStore} \times \mathit{Frame}) \times \mathit{LocalStore} \end{aligned}$$

The relation T models thread steps that do not manipulate the stack. The relation $T(t, g, l, g', l')$ holds if the thread t can take a step from a state with global and local stores g and l, respectively, yielding (possibly modified) stores g' and l', and where the stack is not accessed or updated during this step. The relation $T^+(t, g, l, l', f)$ models steps of thread t that push a frame onto the stack. The global and local stores are initially g and l, the global store is unmodified during this step, the local store is updated to l', and the frame f is pushed onto the stack. Similarly, the relation $T^-(t, g, l, f, l')$ models steps of thread t that pop a frame from the stack. The global and local stores are initially g and l and the frame f is initially on top of the stack. After the step, the global store is unmodified, the local store is updated to l', and the frame f has been popped from the stack.

We assume that all stacks are empty in the initial state, and let ss_0 map each thread identifier to the empty stack. The set of reachable states is then defined by the least relation $R \subseteq State$ satisfying the following rules.

Basic PDA model checking

(BASIC PDA INIT)
$$\frac{}{R(g_0, ls_0, ss_0)}$$

(BASIC PDA STEP)
$$\frac{R(g, ls, ss) \quad T(t, g, ls(t), g', l')}{R(g', ls[t := l'], ss)}$$

(BASIC PDA PUSH)
$$\frac{R(g, ls, ss) \quad T^+(t, g, ls(t), l', f)}{R(g', ls[t := l'], ss[t := ss(t).f])}$$

(BASIC PDA POP)
$$\frac{R(g, ls, ss) \quad ss(t) = s.f \quad T^-(t, g, ls(t), f, l')}{R(g, ls[t := l'], ss[t := s])}$$

Since the stack sizes are unbounded, the set of reachable states may also be unbounded. Consequently, any algorithm to compute R may diverge. In fact, the model checking problem for concurrent pushdown systems is undecidable, a result that can be proved by reduction from the undecidable problem of determining if the intersection of two context-free languages is empty [Ram00].

3.1 Assume-guarantee model checking

Although sound and complete model checking of concurrent pushdown systems is undecidable, assume-guarantee reasoning allows us to model check such systems an a conservative yet useful manner. Again, we model check each thread separately, using the guarantees to reason about the effect of interleaved steps of other threads. The algorithm works by computing the guarantee relation $\mathcal{G}$ and the reachability relations $\mathcal{P}$ and $\mathcal{Q}$.

$$\mathcal{G} \subseteq Tid \times GlobalStore \times GlobalStore$$
$$\mathcal{P} \subseteq Tid \times GlobalStore \times LocalStore \times GlobalStore \times LocalStore$$
$$\mathcal{Q} \subseteq Tid \times GlobalStore \times LocalStore \times Frame \times GlobalStore \times LocalStore$$

The guarantee $\mathcal{G}(t, g, g')$ holds if a step by thread t can go from a reachable states with global store g to a state with global store g'. The reachability relation $\mathcal{P}(t, g, l, g', l')$ holds if (1) the system can reach a state with global store g and where thread t has local store l, and (2) from any such state, the system can later reach a state with global store g' and where thread t has local store l', and where the stack is identical to that in the first state. Similarly, the reachability relation $\mathcal{Q}(t, g, l, f, g', l')$ holds if (1) the system can reach a state with global store g and where thread t has local store l, and (2) from any such state, the system can later reach a state with global store g' and where thread t has local store l', and where the stack is identical to that in the first state except that the frame f has been added to it. These relations are defined as the least solution to the following rules.

Assume-guarantee PDA model checking

(AG PDA INIT)

$$\frac{}{\mathcal{P}(t, g_0, ls_0(t), g_0, ls_0(t))}$$

(AG PDA ENV1)

$$\frac{\mathcal{P}(t, g_1, l_1, g_2, l_2) \quad \mathcal{G}(e, g_2, g_3) \quad e \neq t}{\mathcal{P}(t, g_1, l_1, g_3, l_2)}$$

(AG PDA ENV2)

$$\frac{\mathcal{Q}(t, g_1, l_1, f, g_2, l_2) \quad \mathcal{G}(e, g_2, g_3) \quad e \neq t}{\mathcal{Q}(t, g_1, l_1, f, g_3, l_2)}$$

(AG PDA STEP1)

$$\frac{\mathcal{P}(t, g_1, l_1, g_2, l_2) \quad T(t, g_2, l_2, g_3, l_3)}{\mathcal{P}(t, g_1, l_1, g_3, l_3) \quad \mathcal{G}(t, g_2, g_3)}$$

(AG PDA PUSH)

$$\frac{\mathcal{P}(t, g_1, l_1, g_2, l_2) \quad T^+(t, g_2, l_2, l_3, f)}{\mathcal{Q}(t, g_1, l_1, f, g_2, l_3) \quad \mathcal{P}(t, g_2, l_3, g_2, l_3)}$$

(AG PDA STEP2)

$$\frac{\mathcal{Q}(t, g_1, l_1, f, g_2, l_2) \quad \mathcal{P}(t, g_2, l_2, g_3, l_3)}{\mathcal{Q}(t, g_1, l_1, f, g_3, l_3)}$$

(AG PDA POP)

$$\frac{\mathcal{Q}(t, g_1, l_1, f, g_2, l_2) \quad T^-(t, g_2, l_2, f, l_3)}{\mathcal{P}(t, g_1, l_1, g_2, l_3)}$$

The set of reachable states determined using assume guarantee reasoning is a conservative approximation of the set of actual reachable states, as illustrated by the following lemma.

Lemma 2. *For all global stores g and local store maps ls and stack maps ss, if $R(g, ls, ss)$ then for all thread identifiers t, there exists some g', l' such that $\mathcal{P}(t, g', l', g, ls(t))$.*

Again, if a software error causes an erroneous state to be reachable, *i.e.*,

$$\exists t, g, ls.\ (R(g, ls, ss) \wedge E(t, g, ls(t)))$$

then the assume guarantee algorithm will catch that error, *i.e.*

$$\exists t, g, l, g', l'.\ (\mathcal{P}(t, g', l', g, l) \wedge E(t, g, l))\ .$$

The complexity of this algorithm is $O(n.G^3.L^3.F)$, where $F = |Frame|$, since each inference rule can be applied at most this many times.

4 Discussion

We have presented a new technique called assume-guarantee model checking for verifying multithreaded software systems. Although incomplete for general systems, this technique is particularly effective for loosely-coupled multithreaded software where the the various threads synchronize using primitives such as mutexes, readers-writer locks, etc. If the synchronization primitives are modeled with appropriate auxiliary information, these systems can be verified one thread at a time.

Realistic software systems often have dynamic thread creation that may lead to unbounded number of threads. This aspect of multithreaded software is currently not handled by our algorithm. However, the set of thread identifers, even if infinite, is a scalarset type [ID96]. Consequently, these systems are amenable to symmetry reduction which we plan to exploit in future work.

The assume-guarantee model checking algorithm constructs a particular abstraction of multithreaded software using environment assumptions. However, the abstraction might be too coarse to verify the relevant correctness property. If the algorithm reports an error, we would like an efficient procedure to check whether the violation is real or introduced due to the abstraction process. In the second case, we would like to automatically refine the environment assumptions by possibly explicating some aspect of the program counters of the other threads in the environment. After the refinement, the model checking algorithm can be repeated. Thus, the assume-guarantee model checking algorithm may be converted to a semi-algorithm that is sound and also complete on termination.

References

[BET03] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL 03: Principles of Programming Languages*, 2003. to appear.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logic of Programs*, Lecture Notes in Computer Science 131, pages 52–71. Springer-Verlag, 1981.

[CGP03] J.M. Cobleigh, D. Giannakopoulou, and C.S. Păsăreanu. Learning assumptions for compositional verification. In *TACAS 03: Tools and Algorithms for the Construction and Analysis of Systems*, 2003. to appear.

[FFQ02] C. Flanagan, S.N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *ESOP 02: European Symposium on Programming*, Lecture Notes in Computer Science 2305, pages 262–277, 2002.

[FQS02] C. Flanagan, S. Qadeer, and S. Seshia. A modular checker for multithreaded programs. In *CAV 02: Computer Aided Verification*, Lecture Notes in Computer Science 2404, pages 180–194, 2002.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[ID96] C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1–2):41–75, 1996.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[QS81] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Fifth International Symposium on Programming*, Lecture Notes in Computer Science 137, pages 337–351. Springer-Verlag, 1981.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000.