Modular Verification of Multithreaded Programs
Williams College Technical Note 04-08

Cormac Flanagan

Computer Science Department, University of California at Santa Cruz, Santa Cruz, CA 95064

Stephen N. Freund

Computer Science Department, Williams College, Williamstown, MA 01267

Shaz Qadeer

Microsoft Research, One Microsoft Way, Redmond, WA 98052

Sanjit A. Seshia

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

Abstract

Multithreaded software systems are prone to errors due to the difficulty of reasoning about multiple interleaved threads operating on shared data. Static checkers that analyze a program's behavior over all execution paths and all thread interleavings are a powerful approach to identifying bugs in such systems. In this paper, we present Calvin, a scalable and expressive static checker for multithreaded programs based on automatic theorem proving. To handle realistic programs, Calvin performs modular checking of each procedure called by a thread using specifications of other procedures and other threads. Our experience applying Calvin to several real-world programs indicates that Calvin has a moderate annotation overhead and can catch common defects in multithreaded programs, such as synchronization errors and violations of data invariants.

---

1 Introduction

Many important software systems, such as operating systems and databases, are multithreaded. Ensuring the reliability of these systems is an essential but challenging task. It is difficult to ensure reliability through testing alone, because of subtle, nondeterministic interactions between threads. A timing-dependent bug may remain hidden despite months of testing, only to show up after the system is deployed. Static checkers complement testing by analyzing program behavior over all execution paths and all thread interleavings. However, current static checking techniques for multithreaded programs are unable to scale to large programs and handle complicated synchronization mechanisms.

To obtain scalability, static checkers often employ modular analysis techniques that analyze each component of a system separately, using only a specification of other components. A standard notion of modularity for sequential programs is *procedure-modular* reasoning [33], where a call site of a procedure is analyzed using a precondition/postcondition specification of that procedure. However, this style of procedure-modular reasoning does not generalize to multithreaded programs [9,30]. An orthogonal notion of modularity for multithreaded programs is *thread-modular* reasoning [28], which avoids the need to consider all possible interleavings of threads explicitly. This technique analyzes each thread separately using a specification, called an *environment assumption*, that constrains the updates to shared variables performed by interleaved actions of other threads. Checkers based on this style of thread-modular reasoning have typically relied upon the inherently non-scalable method of inlining procedure bodies. Consequently, approaches based purely on only one of procedure-modular or thread-modular reasoning are inadequate for large programs with many procedures and many threads.

We present a verification methodology that combines thread-modular and procedure-modular reasoning. In our methodology, a procedure specification consists of an environment assumption and an abstraction. The environment assumption, as in pure thread-modular reasoning, is a two-store predicate that constrains updates to shared variables performed by interleaved actions of other threads. The abstraction is a program that simulates the procedure implementation in an environment that behaves according to the environment assumption. Since each procedure may be executed by any thread, the implementation, environment assumption, and abstraction of a procedure are all parameterized by the thread identifier `tid`.

The specification of a procedure p is correct if two proof obligations are satisfied. First, the abstraction of p must simulate the implementation of p. Second, each step of the implementation must satisfy the environment assumption of p for every thread other than `tid`. These two properties are checked for all `tid`, and they need to hold only in an environment that behaves according to the environment assumption of p. In addition, our checking technique proves them by inlining the abstractions rather than the implementations of procedures called in the implementation of p. We reduce these two checks to verifying the correctness of a sequential program and present an algorithm to produce this sequential program. This approach allows us to leverage existing techniques for verifying sequential programs based on verification conditions and automatic theorem proving.

We have implemented our methodology for multithreaded Java [6] programs in the Calvin checking tool. We have applied Calvin to several multithreaded programs, the largest of which is a 1500 line portion of the web crawler Mercator [26] in use at Altavista. Our experience indicates that Calvin has the following useful features:

(1) *Scalability via modular reasoning:* It naturally scales to programs with many procedures and threads since each procedure implementation is analyzed separately using the specifications for the other threads and procedures.
(2) *Ability to handle varied synchronization idioms:* The checker is sufficiently expressive to handle the variety of synchronization idioms commonly found in systems code, e.g., readers-writer locks, producer-consumer synchronization, and time-varying mutex synchronization.
(3) *Expressive abstractions:* Although a procedure abstraction can describe complex behaviors (and in an extreme case could detail every step of the implementation), in general the appropriate abstraction for a procedure is relatively succinct. In addition, the necessary environment assumption annotations are simple and intuitive for programs using common synchronization idioms, such as mutual exclusion or reader-writer locks.
(4) *Moderate annotation overhead:* Annotations are not brittle with respect to program changes. That is, code modifications having little effect on a program's overall behavior typically require only small changes to any annotations.

The moderate annotation overhead of our checker suggests that static checking may be a cost-effective approach for ensuring the reliability of multithreaded software, simply due to the extreme difficulty of ensuring reliability via traditional methods such as testing.

The following section introduces *Plato*, an idealized multithreaded language that we use to formalize our analysis. Section 3 presents several example programs that motivate and provide an overview of our analysis technique. Section 4 and 5 present a complete, formal description of our analysis. Section 6 describes our implementation and Section 7 describes its application to some real-world programs. Section 8 surveys related work, and Section 9 concludes. Proofs of theorems stated in the paper are provided in the Appendix.

This paper is a unified description of results presented in preliminary form at conferences [21,23]. In particular, this extended presentation includes a revised formal semantics, a correctness proof for our verification methodology based on this semantics, and an additional case study (the Apprentice challenge problem proposed by Moore and Porter [37]).

2 The parallel language Plato

In this section, we present the idealized parallel programming language Plato (parallel language of atomic operations), and introduce notation and terminology for the rest of the paper. In order to avoid the complexity of reasoning about programs written in a large, complex language like Java, our theoretical discussion focuses on verification of programs in Plato. The topic of translating Java into Plato is addressed in Section 6.

Figure 1: Plato syntax

$$
\begin{array}{rcll}
s,t \in \mathit{Tid} & = & \{1,2,3,\ldots\} & \\
\sigma \in \mathit{GlobalStore} & = & \mathit{GlobalVar} \to \mathit{Value} & \\
\lambda \in \mathit{LocalStore} & = & \mathit{LocalVar} \to \mathit{Value} & \\
\mathit{VisibleStore} & = & \mathit{GlobalStore} \times \mathit{LocalStore} & \\
\Lambda \in \mathit{Stack} & = & \mathit{LocalStore}^* & \\
z \in \mathit{Stacks} & = & \mathit{Tid} \to \mathit{Stack} & \\
\mathit{Store} & = & \mathit{GlobalStore} \times \mathit{Stacks} & \\
p,q \in \mathit{Predicate} & \subseteq & \mathit{Tid} \times \mathit{VisibleStore} & \\
X,Y \in \mathit{Action} & \subseteq & \mathit{Tid} \times \mathit{VisibleStore} \times \mathit{VisibleStore} & \\
m \in \mathit{Proc} & & & \\
\mathcal{B} \in \mathit{Defn} & = & \mathit{Proc} \to \mathit{Stmt} & \\
 & & & \\
P,Q \in \mathit{Program} & ::= & \| S & \\
S,T,U \in \mathit{Stmt} & ::= & a & \text{atomic op} \\
 & | & S_1; S_2 & \text{composition} \\
 & | & S_1 \Box S_2 & \text{choice} \\
 & | & S^* & \text{iteration} \\
 & | & m() & \text{procedure call} \\
a,b,c \in \mathit{AtomicOp} & ::= & p?X &
\end{array}
$$

Figure 1 shows the Plato syntax. A Plato program P is the parallel composition of an unbounded number of threads. Every thread has an associated thread identifier, which is a positive integer. The set Tid is the set of all thread identifiers. Each thread executes the same statement S, but S is parameterized by the identifier of the current thread, which allows different threads to exhibit different behavior.

When the program P is executed, the steps of its threads are interleaved nondeterministically. Threads operate on a store (σ, z), where σ is a global store and z maps each $t \in \mathit{Tid}$ to the stack of thread t. The global store maps global variables to values. The set of values is left unspecified because it is orthogonal to our development. The stack of a thread is a sequence of local stores, where each local store maps local variables to values. A sequential statement may be an atomic operation (described below); a sequential composition $S_1; S_2$; a nondeterministic choice $S_1 \Box S_2$ that executes either S_1 or S_2; an iteration statement S^* that executes S an arbitrary (zero or more) number of times; or a procedure call $m\texttt{()}$. The names of procedures are drawn from the set *Proc*, and the function $\mathcal{B}$ maps procedure names to their implementations.

Atomic operations generalize many of the basic constructs found in programming languages, such as assignment and assertion. An atomic operation has the form $p?X$. Both the predicate p and the action X are parameterized by the identifier of the current thread. The predicate p must be true in the pre-store of the operation. This predicate cannot access the entire state (σ, z). Instead, it can only access the *visible store*, which consists of the global store and the local store at the top of the current thread's stack. For convenience, we extend the interpretation of p to the full store and write $p(t, (\sigma, z))$ to mean $\exists \lambda, \Lambda.\ z(t) = \lambda \cdot \Lambda \wedge p(t, (\sigma, \lambda))$. The *action* X is a predicate over two stores, and it describes the effect of performing the operation in terms of the pre-store and post-store. The action X also refers only to the visible store. We extend the interpretation of X to the full store and write $X(t, (\sigma, z), (\sigma', z'))$ to mean

$$\exists \lambda, \Lambda.\ z(t) = \lambda \cdot \Lambda \wedge X(t, (\sigma, \lambda), (\sigma', \lambda')) \wedge z' = z[t \mapsto \lambda' \cdot \Lambda]$$

When a thread with identifier t executes the atomic operation $p?X$ in store (σ, z), there are two possible outcomes. If $p(t, (\sigma, z))$ is false, then execution of the multithreaded program terminates in a special global state **wrong** to indicate that an error has occurred. If $p(t, (\sigma, z))$ holds, the program moves into a post-store σ' such that the constraint $X(t, (\sigma, z), (\sigma', z'))$ is satisfied. If no such σ' exists, the atomic operation blocks. Other threads may continue while this operation is blocked. Updates to the store performed by the other threads might unblock this thread later. The formal semantics of an atomic operation as a transition relation is given in Section 2.1.

An action is typically written as a formula containing unprimed and primed variables and a special variable `tid`. Unprimed variables refer to their value in the pre-store of the action, primed variables refer to their value in the post-store of the action, and `tid` is the identifier of the currently executing thread. A predicate is written as a formula with only unprimed variables and `tid`.

For any action X and set of variables $V \subseteq \mathit{Var}$, we use the notation $\langle X \rangle_V$ to mean the action that satisfies X and only allows changes to variables in V between the pre-store and the post-store, and we use $\langle X \rangle$ to abbreviate $\langle X \rangle_\emptyset$.

Finally, we abbreviate the atomic operation $\texttt{true}?X$ to the action X.

Figure 2: Conventional constructs in Plato

$$
\begin{aligned}
\texttt{x = e} &\stackrel{\text{def}}{=} \langle \texttt{x}' = \texttt{e} \rangle_{\texttt{x}} \\
\texttt{assert e} &\stackrel{\text{def}}{=} \texttt{e}?\langle \texttt{true} \rangle \\
\texttt{assume e} &\stackrel{\text{def}}{=} \langle \texttt{e} \rangle \\
\texttt{if (e) \{ } S \texttt{ \}} &\stackrel{\text{def}}{=} (\texttt{assume e}; S) \Box (\texttt{assume} \neg\texttt{e}) \\
\texttt{while (e) \{ } S \texttt{ \}} &\stackrel{\text{def}}{=} (\texttt{assume e}; S)^*; (\texttt{assume} \neg\texttt{e}) \\
\\
\texttt{acquire(mx)} &\stackrel{\text{def}}{=} \langle \texttt{mx} = 0 \wedge \texttt{mx}' = \texttt{tid} \rangle_{\texttt{mx}} \\
\texttt{release(mx)} &\stackrel{\text{def}}{=} \langle \texttt{mx}' = 0 \rangle_{\texttt{mx}} \\
\texttt{skip} &\stackrel{\text{def}}{=} \langle \texttt{true} \rangle \\
\texttt{havoc} &\stackrel{\text{def}}{=} \langle \texttt{true} \rangle_{\mathit{Var}} \\
\texttt{CAS(l,e,n)} &\stackrel{\text{def}}{=} \left\langle \begin{array}{l} \wedge\, \texttt{l} \neq \texttt{e} \Rightarrow (\texttt{l}' = \texttt{l} \wedge \texttt{n}' = \texttt{n}) \\ \wedge\, \texttt{l} = \texttt{e} \Rightarrow (\texttt{l}' = \texttt{n} \wedge \texttt{n}' = \texttt{l}) \end{array} \right\rangle_{\texttt{l,n}}
\end{aligned}
$$

Using atomic operations, Plato can express many conventional constructs, including assignment, assert, assume, if, and while statements. Figure 2 presents the encoding of these statements in Plato. Let `e` be an expression. The statement "`assert e`" goes wrong from a state in which e is false. Otherwise, it terminates without modifying the store. The statement "`assume e`" blocks until `e` is true and then terminates without modifying the store. Atomic operations can also express primitive synchronization operations such as acquiring and releasing locks. A lock is modeled as a variable `mx` which is either 0, if the lock is not held, or otherwise is a positive integer identifying the thread holding the lock. The statement "`CAS(l,e,n)`", in which `l` and `n` are variables and `e` is an expression, models the atomic compare-and-swap operation often used for synchronization. If $\texttt{l} = \texttt{e}$, then the operation terminates after swapping the contents of `l` and `n`. Otherwise, the operation terminates without modifying the store.

2.1 Semantics

For the remainder of this paper, we assume a fixed function $\mathcal{B}$ mapping procedure names to procedure bodies. We define the semantics of a statement S as a set $[\![S]\!]$ of sequences of atomic operations that could be performed by executing S. To give semantics to method calls, we introduce two new atomic operations *Push* and *Pop*. We first define the set $[\![S]\!]^d$ of sequences through S where the stack depth never exceeds d (see Figure 3). The set of sequences $[\![S]\!]$ is then obtained as the union of $[\![S]\!]^d$ for all $d \geq 0$.

A *thread* is a pair $|t,S|$ consisting of a thread identifier t and a statement S being executed by thread t. A *step* $|t,a|$ is a thread whose statement component is an atomic operation. A *path* is a finite sequence of steps. If $\bar{a} = a_1; \ldots; a_n$, then $|t,\bar{a}|$ represents the path $|t,a_1|; \ldots; |t,a_n|$, where all steps are taken by the same thread. A thread $|t,S|$ yields the set of paths $[\![|t,S|]\!] = \{|t,\bar{a}| \mid \bar{a} \in [\![S]\!]\}$.

A parallel program P can be translated into the set of paths $[\![P]\!]$, as shown in Figure 3. The path $\bar{u};\bar{w}$ is the concatenation of paths $\bar{u}$ and $\bar{w}$. We will refer to a set of paths as a *pathset*. The pathset $\varphi_1;\varphi_2$ is the set of all paths obtained by the concatenation of a path from pathset φ_1 and a path from pathset φ_2. Note that we are overloading the operator ";" to mean both the sequential composition of statements and steps as well as the concatenation of paths and pathsets. The pathset φ^* is the Kleene closure of the pathset φ. The pathset $\bar{u}_1 \otimes \ldots \otimes \bar{u}_n$ is the set of all interleavings of the paths $\bar{u}_1, \ldots, \bar{u}_n$. The pathset $\varphi_1 \otimes \ldots \otimes \varphi_n$ is the union of all pathsets obtained by taking the interleavings of a path from each φ_i for $1 \le i \le n$.

Figure 3: Program paths

$$
\begin{array}{rcll}
a, b & \in & \mathit{AtomicOp} \cup \{\mathit{Push}, \mathit{Pop}\} & \\
\bar{a}, \bar{b} & \in & \mathit{Seq} & = a_1; \ldots; a_n \\
u, w & \in & \mathit{Step} & = |t, a| \\
\bar{u}, \bar{w} & \in & \mathit{Path} & = u_1; \ldots; u_n \\
\varphi & \in & \mathit{PathSet} &
\end{array}
$$

$$
\begin{array}{rcl}
[\![\bullet]\!]^{\bullet} & : & \mathit{Stmt} \times \mathbb{N} \to 2^{\mathit{Seq}} \\
[\![a]\!]^d & = & \{a\} \\
[\![S_1; S_2]\!]^d & = & [\![S_1]\!]^d; [\![S_2]\!]^d \\
[\![S_1 \Box S_2]\!]^d & = & [\![S_1]\!]^d \cup [\![S_2]\!]^d \\
[\![S^*]\!]^d & = & ([\![S]\!]^d)^* \\
[\![m()]\!]^d & = & \begin{cases} \{\mathit{Push}\}; [\![\mathcal{B}(m)]\!]^{d-1}; \{\mathit{Pop}\} & \text{if } d > 0 \\ \emptyset & \text{if } d = 0 \end{cases}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\bullet]\!] & : & \mathit{Stmt} \to 2^{\mathit{Seq}} \\
[\![S]\!] & = & \bigcup_{d \ge 0} \; [\![S]\!]^d
\end{array}
$$

$$
\begin{array}{rcl}
[\![\bullet]\!] & : & \mathit{Program} \to \mathit{PathSet} \\
[\![\|_{i=1}^n S]\!] & = & [\![|1,S|]\!] \;\otimes\; \ldots \;\otimes\; [\![|n,S|]\!] \\
[\![\|S]\!] & = & \bigcup_{n \ge 1} \; [\![\|_{i=1}^n S]\!]
\end{array}
$$

We formalize the behavior of an atomic operation as a *transition relation*, which is a partial map from a store and an execution step to a state. A state contains the global state, which is either a global store or the special state **wrong**, together with the stacks of the threads.

Figure 4: Transition relation

$$
\begin{array}{rcl}
\omega \in \mathit{GlobalState} & = & \mathit{GlobalStore} \cup \{\mathbf{wrong}\} \\
\mathit{State} & = & \mathit{GlobalState} \times \mathit{Stacks} \\
\bullet \xrightarrow{\bullet} \bullet & \subseteq & \mathit{Store} \times \mathit{Step} \times \mathit{State}
\end{array}
$$

Given $u = |t, p?X|$,

$$(\sigma, z) \xrightarrow{u} (\sigma', z') \qquad \text{if } p(t, (\sigma, z)) \text{ and } X(t, (\sigma, z), (\sigma', z'))$$
$$(\sigma, z) \xrightarrow{u} (\mathbf{wrong}, z) \qquad \text{if } \neg p(t, (\sigma, z))$$

Given $u = |t, \mathit{Push}|$,

$$(\sigma, z) \xrightarrow{u} (\sigma, z[t \mapsto \lambda \cdot \Lambda]) \qquad \text{if } z(t) = \Lambda \text{ and } \lambda \in \mathit{LocalStore}$$

Given $u = |t, \mathit{Pop}|$,

$$(\sigma, z) \xrightarrow{u} (\sigma, z[t \mapsto \Lambda]) \qquad \text{if } z(t) = \lambda \cdot \Lambda$$

If $\bar{u} = |t_1, a_1|; \ldots ; |t_n, a_n|$ is a path, then

$$r = (\sigma_1, z_1) \xrightarrow{|t_1,a_1|} (\sigma_2, z_2) \cdots (\sigma_k, z_k) \xrightarrow{|t_k,a_k|} (\omega, z_{k+1})$$

for some $1 \leq k \leq n$ is a *run* of $\bar{u}$. If $k = n$ or $\omega = \mathbf{wrong}$, then r is a *full run*. Corresponding to each such run, there is a *trace*

$$\tau = \sigma_1 \xrightarrow{t_1} \sigma_2 \cdots \sigma_k \xrightarrow{t_k} \omega$$

obtained by ignoring the stacks in the states and atomic operations in the transitions between adjacent states in the run. We denote the trace τ by $\mathit{trace}(r)$. If r is a run of $\bar{u} \in \varphi$, it is defined to be a run of φ and $\mathit{trace}(r)$ is defined to be a trace of φ. If r is a full run, we say that $\mathit{trace}(r)$ is a *full trace*. If $\varphi = [\![P]\!]$, a run (respectively, a trace) of φ is also a run (resp., a trace) of P.

We say that a program P *goes wrong from* σ if a run of P starting in σ ends in **wrong**. A program P *goes wrong* if P goes wrong from some store σ. A set of global stores I is an *invariant* of the program P if for all traces $\sigma_1 \xrightarrow{t_1} \sigma_2 \cdots \sigma_k \xrightarrow{t_k} \sigma_{k+1}$ of P, whenever $\sigma_1 \in I$ then $\sigma_{k+1} \in I$.

3 Overview of modular verification

In the remainder of this paper, we develop a scheme for modularly checking that a multithreaded program does not go wrong and satisfies specified invariants. We start by considering an example that provides an overview and motivation of our modular verification method. Consider the multithreaded program SimpleLock in Figure 5. It consists of two modules, `Top` and `Mutex`. A module is defined informally to be a collection of procedures and global variables. The module `Top` contains two procedures that manipulate a shared integer variable `x`, which is initially zero and is protected by a mutex `m`. The module `Mutex` provides acquire and release operations on that mutex. The mutex variable `m` is either the (positive) identifier of the thread holding the lock, or else 0, if the lock is not held by any thread. The implementation of `acquire` is non-atomic, and uses busy-waiting based on the atomic compare-and-swap instruction (`CAS`) described earlier. The local variable `t` cannot be modified by other threads. We assume the program starts execution by concurrently calling procedures `t1` in thread 1 and `t2` in thread 2. Note that this program can be expressed as the following multithreaded Plato program:

$$\| \; ((\texttt{assume tid} = 1; \texttt{t1()}) \;\Box\; (\texttt{assume tid} = 2; \texttt{t2()}))$$

Figure 5: SimpleLock program

```
// module Top
int x = 0;

void t1() {            void t2() {
  acquire();             acquire();
  x++;                   x = 0;
  assert x > 0;          release();
  release();           }
}
```

```
// module Mutex
int m = 0;

void acquire() {       void release()
  var t = tid;         {
  while (t == tid)       m = 0;
    CAS(m,0,t);        }
}
```

We would like the checker to verify that the assertion in `t1` never fails. This assertion should hold because `x` is protected by `m` and because the mutex implementation is correct.

To avoid considering all possible interleavings of the various threads, our checker performs thread-modular reasoning, and relies on the programmer to specify an *environment assumption* constraining the interactions among threads. The environment assumption is an action that refers only to the global program variables and the variable `tid`. This action has the property that its execution by thread t mimics updates to the global variables by threads other than t. For SimpleLock, an appropriate environment assumption is:

$$E \stackrel{\text{def}}{=} \begin{array}{l} \land\ \texttt{m} = \texttt{tid} \Rightarrow \texttt{m} = \texttt{m}' \\ \land\ \texttt{m} = \texttt{tid} \Rightarrow \texttt{x} = \texttt{x}' \end{array}$$

The two conjuncts state that if thread `tid` holds the lock `m`, then other threads cannot modify either `m` or the protected variable `x`. No environment assumption is required for the local variable `t` since it cannot be accessed by concurrent threads. We also specify an invariant I stating that whenever the lock is not held, $\mathtt{x}$ is at least zero:

$$I \stackrel{\text{def}}{=} \mathtt{m} = 0 \Rightarrow \mathtt{x} \geq 0$$

This invariant is necessary to ensure, after `t1` acquires the lock and increments `x`, that `x` is strictly positive.

3.1 Thread-modular verification

For small programs, it is not strictly necessary to perform procedure-modular verification. Instead, our checker could inline procedure implementations at corresponding call sites (at least for non-recursive procedures).

Let $\mathit{InlineBody}(S)$ denote the statement obtained by inlining the implementation of called procedures in a statement S. Let us consider procedure `t1` in the example of Figure 5. Figure 6(a) shows the implementation $\mathcal{B}(\mathtt{t1})$ of `t1`. Figure 6(b) shows $\mathit{InlineBody}(\mathcal{B}(\mathtt{t1}))[\mathtt{tid} := 1]$, the result of replacing `tid` with the thread identifier 1 in the statement $\mathit{InlineBody}(\mathcal{B}(\mathtt{t1}))$. (All statements are represented in terms of atomic operations.)

Let E_i be the action obtained by replacing `tid` with i in E and let E_i^* be the transitive closure of E_i. Thread-modular verification of thread 1 consists of checking the following property:

> $\mathit{InlineBody}(\mathcal{B}(\mathtt{t1}))[\mathtt{tid} := 1]$ is simulated by E_2^* with respect to the environment assumption E_1 from any state satisfying $\mathtt{m} = 0 \wedge \mathtt{x} = 0$.
> (TMV1)

The notion of simulation is formalized later in the paper. For now, we give an intuitive explanation of Property TMV1. Consider Figure 6(c), which shows the interleaving of atomic operations in $\mathit{InlineBody}(\mathcal{B}(\mathtt{t1}))[\mathtt{tid} := 1]$ with ${E_1}^*$ to mimic an arbitrary sequence of atomic operations of thread 2. (Operations mimicking actions of thread 2 are underlined to distinguish them from operations of thread 1.) Checking Property TMV1 involves verifying that when executed from an initial state where both `x` and `m` are zero, the statement in Figure 6(c) does not go wrong, and that each non-underlined atomic operation satisfies E_2. Note that the statement in Figure 6(c) can be viewed as a sequential program, and that Property TMV1 can be checked using sequential program verification techniques.

The procedure `t2` satisfies a corresponding property TMV2 with the roles of E_1 and E_2 swapped. Using assume-guarantee reasoning, our checker infers from TMV1 and TMV2 that the SimpleLock program does not go wrong, no matter how the scheduler interleaves the execution of the two threads.

Figure 6: Thread-modular verification of `t1`

```
acquire();
x++;
assert x > 0;
release();
```

(a) $\mathcal{B}(\texttt{t1})$

$$
\begin{array}{l}
\langle \texttt{t}' = 1\rangle_{\texttt{t}};\\
(\langle \texttt{t} = 1\rangle;\ \texttt{CAS(m,0,1)};)^*;\\
\langle \texttt{t} \neq 1\rangle\\
\langle \texttt{x}' = \texttt{x} + 1\rangle_{\texttt{x}};\\
\texttt{x} > 0?\langle \texttt{true}\rangle;\\
\langle \texttt{m}' = 0\rangle_{\texttt{m}};
\end{array}
$$

(b) $\mathit{InlineBody}(\mathcal{B}(\texttt{t1}))$

$$
\begin{array}{l}
\underline{E_1^*};\ \langle \texttt{t}' = 1\rangle_{\texttt{t}};\\
(\underline{E_1^*};\ \langle \texttt{t} = 1\rangle;\ \underline{E_1^*};\ \texttt{CAS(m,0,1)};)^*;\\
\underline{E_1^*};\ \langle \texttt{t} \neq 1\rangle\\
\underline{E_1^*};\ \langle \texttt{x}' = \texttt{x} + 1\rangle_{\texttt{x}};\\
\underline{E_1^*};\ \texttt{x} > 0?\langle \texttt{true}\rangle;\\
\underline{E_1^*};\ \langle \texttt{m}' = 0\rangle_{\texttt{m}};\ \underline{E_1^*};
\end{array}
$$

(c) $\mathit{InlineBody}(\mathcal{B}(\texttt{t1}))$ interleaved with operations of `t2` satisfying E_1

3.2 Adding procedure-modular verification

The inlining of procedure implementations at call sites prevents the simple approach sketched above from analyzing large systems. To scale to larger systems, our checker performs a procedure-modular analysis that uses procedure specifications in place of procedure implementations. In this context, the main question is: What is the appropriate specification for a procedure in a multi-threaded program?

A traditional precondition/postcondition specification for `acquire` is:

$$\texttt{requires}\ I;\ \ \texttt{modifies m};\ \ \texttt{ensures m} = \texttt{tid} \wedge \texttt{x} \geq 0$$

This specification records that:

- The precondition is I;
- `m` can be modified by the body of `acquire`;
- When `acquire` terminates, `m` is equal to the current thread identifier and `x` is at least 0.

This last postcondition is crucial for verifying the assertion in `t1`.

However, although this specification suffices to verify the assertion in `t1`, it suffers from a serious problem: it mentions the variable `x`, even though `x` should properly be considered a private variable of the separate module `Top`. This problem arises because the postcondition, which describes the final state of the procedure's execution, needs to record store updates performed during execution of the procedure, both by the thread executing this procedure, and also by other concurrent threads (which may modify `x`).

In order to overcome the aforementioned problem and still support modular specification and verification, we allow specifications that can describe intermediate atomic steps of a procedure's execution, and need not summarize effects of interleaved actions of other threads.

In the case of `acquire`, the appropriate specification is that `acquire` first performs an arbitrary number of *stuttering* steps that do not modify `m`; it then performs a single atomic action that acquires the lock; after which it may perform additional stuttering steps before returning. The actions in the specification refer only to the global variables and implicitly allow arbitrary updates to the local variables. The code fragment $\mathcal{A}(\texttt{acquire})$ specifies this behavior:

$$\mathcal{A}(\texttt{acquire}) \quad \stackrel{\text{def}}{=} \quad \langle \texttt{true} \rangle^*; \langle \texttt{m} = 0 \wedge \texttt{m}' = \texttt{tid} \rangle_{\texttt{m}}; \langle \texttt{true} \rangle^*$$

This abstraction specifies only the behavior of thread `tid` and therefore does not mention `x`. Our checker validates the specification of `acquire` by checking that the statement $\mathcal{A}(\texttt{acquire})$ is a correct abstraction of the behavior of `acquire`, i.e.: the statement $\mathcal{B}(\texttt{acquire})$ is simulated by $\mathcal{A}(\texttt{acquire})$ from the set of states satisfying $\texttt{m} = 0$ with respect to the environment assumption `true`.

After validating a similar specification for `release`, our checker replaces calls to `acquire` and `release` from the module `Top` with the corresponding abstractions $\mathcal{A}(\texttt{acquire})$ and $\mathcal{A}(\texttt{release})$. If *InlineAbs* denotes this operation of inlining abstractions, then $\mathit{InlineAbs}(\mathcal{B}(\texttt{t1}))$ and $\mathit{InlineAbs}(\mathcal{B}(\texttt{t2}))$ are free of procedure calls, and so we can apply thread-modular verification, as outlined in Section 3.1, to the module `Top`. In particular, by verifying that

> $\mathit{InlineAbs}(\mathcal{B}(\texttt{t1}))[\texttt{tid} := 1]$ is simulated by E_2^* with respect to E_1 from any state satisfying $\texttt{m} = 0 \wedge \texttt{x} = 0$

and verifying a similar property for `t2`, our checker infers by assume-guarantee reasoning that the complete SimpleLock program does not go wrong.

4 Modular verification

In this section, we formalize our modular verification method sketched in the previous section. Our method requires for each procedure a specification that may refer only to the global variables. To allow us to express such a specification, we introduce a few new definitions:

$$
\begin{array}{rcl}
r \in \mathit{SpecPredicate} & \subseteq & \mathit{Tid} \times \mathit{GlobalStore} \\
Z \in \mathit{SpecAction} & \subseteq & \mathit{Tid} \times \mathit{GlobalStore} \times \mathit{GlobalStore} \\
r?Z \in \mathit{SpecAtomicOp} & & \\
T \in \mathit{SpecStmt} & ::= & r?Z \\
 & \mid & S_1; S_2 \\
 & \mid & S_1 \Box S_2 \\
 & \mid & S^*
\end{array}
$$

Consider the execution of a procedure m by the current thread `tid`. We assume m is accompanied by a specification consisting of three parts: (1) an invariant $\mathcal{I}(m) \in \mathit{SpecPredicate}$ that must be maintained by all threads while executing m, (2) an environment assumption $\mathcal{E}(m) \in \mathit{SpecAction}$ that models the behavior of threads executing concurrently with `tid`'s execution of m, and (3) an abstraction $\mathcal{A}(m) \in \mathit{SpecStmt}$ that summarizes the behavior of thread `tid` executing m. Note that the abstraction $\mathcal{A}(m)$ does not contain any procedure calls.

In order for the abstraction $\mathcal{A}(m)$ to be correct, we require that the implementation $\mathcal{B}(m)$ be simulated by $\mathcal{A}(m)$ with respect to the environment assumption $\mathcal{E}(m)$. Informally, this simulation requirement holds if, assuming other threads perform actions consistent with $\mathcal{E}(m)$, each action of the implementation corresponds to some action of the abstraction. The abstraction may allow more behaviors than the implementation, and may go wrong more often. If the abstraction does not go wrong, then the implementation also should not go wrong and each implementation transition must be matched by a corresponding abstraction transition. When the implementation terminates the abstraction should be able to terminate as well.

We formalize the notion of simulation between (multithreaded) programs. We first define the notion of subsumption between traces. Intuitively, a trace τ is subsumed by a trace τ' if either τ' is identical to τ or τ' behaves like a prefix of τ and then goes wrong. Formally, a trace $\sigma_1 \xrightarrow{t_1} \sigma_2 \cdots \sigma_k \xrightarrow{t_k} \omega$ is *subsumed* by a trace $\sigma'_1 \xrightarrow{t'_1} \sigma'_2 \cdots \sigma'_l \xrightarrow{t'_l} \omega'$ if (1) $l \leq k$, (2) for all $1 \leq i \leq l$, we have $\sigma_i = \sigma'_i$ and $t_i = t'_i$, and (3) either $\omega' = \mathbf{wrong}$ or $l = k$ and $\omega' = \omega$. A pathset φ_1 is *simulated* by the pathset φ_2, written $\varphi_1 \sqsubseteq \varphi_2$ if every trace of φ_1 is subsumed by a trace of φ_2, and every full trace of φ_1 is subsumed by a full trace of φ_2. A program P is *simulated* by a program Q, written $P \sqsubseteq Q$, if $[\![P]\!]$ is simulated by $[\![Q]\!]$.

For any action $E \in \mathit{SpecAction}$ and a thread identifier j, let $\mathit{Fix}(E, j) \in \mathit{Action}$ be the action whose execution by a thread t mimics the execution of E by thread j. Formally, we have $\mathit{Fix}(E, j) = \{(t, (\sigma, \lambda), (\sigma', \lambda')) \mid (j, \sigma, \sigma') \in E\}$. Given a statement B, an environment assumption E, and an integer $j \in \mathit{Tid}$, let $\mathcal{P}(B,E,j)$ be the program in which the j-th thread is B and every other thread is $\mathit{Fix}(E,j)^*$.

$$\mathcal{P}(B,E,j) \stackrel{\text{def}}{=} \|((\texttt{assume tid} = j; B) \Box (\texttt{assume tid} \neq j; \mathit{Fix}(E,j)^*))$$

A statement B is *simulated* by a statement A with respect to an environment assumption E, written $B \sqsubseteq_E A$, if the program $\mathcal{P}(B,E,j)$ is simulated by the program $\mathcal{P}(A,E,j)$ for all $j \in \mathit{Tid}$.

Apart from being simulated by $\mathcal{A}(m)$, the implementation $\mathcal{B}(m)$ must also satisfy two other properties. First, every atomic operation executed by thread t during the execution of $\mathcal{B}(m)$ must preserve the invariant $\mathcal{I}(m)$. Second, the execution must satisfy the environment assumption of any thread j other than t executing in any procedure m'. The environment assumption of a procedure (for thread t) must be strong enough so that environment assumptions of all procedures (for a thread j different from t) can be verified with its aid. This requirement is undesirable because it would require a procedure to know about the details of its clients. Our methodology weakens this requirement without losing soundness and requires us to verify the environment assumptions of only those procedures that are transitively called from m. Let $\rightsquigarrow$ be the *calls* relation on the set *Proc* of procedures such that $m \rightsquigarrow l$ iff procedure m calls the procedure l. Let $\rightsquigarrow^*$ be the reflexive-transitive closure of $\rightsquigarrow$. We define a derived environment assumption for procedure m as

$$\hat{\mathcal{E}}(m) = \bigwedge_{m \rightsquigarrow^* l} \mathcal{E}(l).$$

We can check that $\mathcal{B}(m)$ is simulated by $\mathcal{A}(m)$ and also satisfies the aforementioned properties by checking that $\mathcal{B}(m)$ is simulated by a derived abstraction $\hat{\mathcal{A}}(m)$. This derived abstraction $\hat{\mathcal{A}}(m)$ is obtained from $\mathcal{A}(m)$ by replacing every atomic operation $r?Z$ in $\mathcal{A}(m)$ by $\hat{r}?\hat{Z}$ defined as follows:

$$\begin{array}{rcl}
(t,\sigma) \in \hat{r} & \stackrel{\text{def}}{=} & \wedge\ (t,\sigma) \in r \\
 & & \wedge\ (t,\sigma) \in \mathcal{I}(m) \\
(t,\sigma,\sigma') \in \hat{Z} & \stackrel{\text{def}}{=} & \wedge\ (t,\sigma,\sigma') \in Z \\
 & & \wedge\ (t,\sigma') \in \mathcal{I}(m) \\
 & & \wedge\ \forall j \in \mathit{Tid} : j \neq t \Rightarrow (j,\sigma,\sigma') \in \hat{\mathcal{E}}(m)
\end{array}$$

In order to check simulation for a procedure m, we first inline the derived abstractions for procedures called from $\mathcal{B}(m)$. We replace the call to a procedure m' in the body of m by $\mathit{PreserveLocals}(\hat{\mathcal{A}}(m'))$, where the function *PreserveLocals* is defined below. The application of this function ensures that the inlined abstraction does not change the local variables of m.

$$
\begin{aligned}
\mathit{PreserveLocals}(r) &\stackrel{\text{def}}{=} \{(t, (\sigma, \lambda)) \mid r(t, \sigma)\} \\
\mathit{PreserveLocals}(Z) &\stackrel{\text{def}}{=} \{(t, (\sigma, \lambda), (\sigma', \lambda)) \mid Z(t, \sigma, \sigma')\} \\
\mathit{PreserveLocals}(r?Z) &\stackrel{\text{def}}{=} \mathit{PreserveLocals}(r)?\mathit{PreserveLocals}(Z) \\
\mathit{PreserveLocals}(S_1; S_2) &\stackrel{\text{def}}{=} \mathit{PreserveLocals}(S_1); \mathit{PreserveLocals}(S_2) \\
\mathit{PreserveLocals}(S_1 \Box S_2) &\stackrel{\text{def}}{=} \mathit{PreserveLocals}(S_1) \Box \mathit{PreserveLocals}(S_2) \\
\mathit{PreserveLocals}(S^*) &\stackrel{\text{def}}{=} \mathit{PreserveLocals}(S)^*
\end{aligned}
$$

We use *InlineAbs* : *Stmt* $\rightarrow$ *Stmt* to denote this abstraction inlining operation. We then check that $\mathit{InlineAbs}(\mathcal{B}(m))$ is simulated by $\mathit{HavocLocals}(\hat{\mathcal{A}}(m))$ with respect to the environment assumption $\hat{\mathcal{E}}(m)$, where the function *HavocLocals* is defined below.

$$
\begin{aligned}
\mathit{HavocLocals}(r) &\stackrel{\text{def}}{=} \{(t, (\sigma, \lambda)) \mid r(t, \sigma)\} \\
\mathit{HavocLocals}(Z) &\stackrel{\text{def}}{=} \{(t, (\sigma, \lambda), (\sigma', \lambda')) \mid Z(t, \sigma, \sigma')\} \\
\mathit{HavocLocals}(r?Z) &\stackrel{\text{def}}{=} \mathit{HavocLocals}(r)?\mathit{HavocLocals}(Z) \\
\mathit{HavocLocals}(S_1; S_2) &\stackrel{\text{def}}{=} \mathit{HavocLocals}(S_1); \mathit{HavocLocals}(S_2) \\
\mathit{HavocLocals}(S_1 \Box S_2) &\stackrel{\text{def}}{=} \mathit{HavocLocals}(S_1) \Box \mathit{HavocLocals}(S_2) \\
\mathit{HavocLocals}(S^*) &\stackrel{\text{def}}{=} \mathit{HavocLocals}(S)^*
\end{aligned}
$$

Note that the recursive definition of the two functions *PreserveLocals* and *HavocLocals* differs only in the case of actions. While $\mathit{HavocLocals}(Z)$ allows arbitrary updates to the local variables, $\mathit{PreserveLocals}(Z)$ leaves the local variables unchanged. The following theorem formalizes our modular verification methodology.

Theorem 1 *For each procedure $m \in \mathit{Proc}$, let its body $\mathcal{B}(m) \subseteq \mathit{Stmt}$, abstraction $\mathcal{A}(m) \subseteq \mathit{SpecStmt}$, environment assumption $\mathcal{E}(m) \subseteq \mathit{SpecAction}$, and invariant $\mathcal{I}(m) \subseteq \mathit{SpecPredicate}$ be given. Let $P = \| l\texttt{()}$ be a parallel program. Suppose for all procedures $m \in \mathit{Proc}$, the statement $\mathit{InlineAbs}(\mathcal{B}(m))$ is simulated by $\mathit{HavocLocals}(\hat{\mathcal{A}}(m))$ with respect to the environment assumption $\hat{\mathcal{E}}(m)$. Then the following are true.*

(1) P is simulated by $Q = \|\mathit{HavocLocals}(\hat{\mathcal{A}}(l))$.
*(2) If $\sigma \in \mathcal{I}(l)$, $\mathit{HavocLocals}(\mathcal{A}(l))$ is simulated by $\texttt{true}^*$ with respect to $\hat{\mathcal{E}}(l)$, and $\sigma \xrightarrow{t_1} \cdots \xrightarrow{t_k} \omega$ is a trace of P, then $\omega \neq$ **wrong** and $\omega \in \mathcal{I}(l)$.*

By verifying simulation for each procedure, the modular verification theorem allows us to conclude two results. First, the program $P = \| l\texttt{()}$ is simulated by a program Q in which every thread executes the derived specification of l. Second, if the specification of l is simulated by $\texttt{true}^*$ (a statement in which no atomic operation goes wrong) with respect to its derived assumption, then the execution of every atomic operation in the specification of l by a thread t satisfies the environment assumption of every procedure transitively called from l for every thread other than t. This fact allows us to conclude that the parallel program Q will not go wrong if it begins execution in a global store satisfying $\mathcal{I}(l)$.

The proof of this theorem is given in Appendix A. Discharging the proof obligations in this theorem requires a method for checking simulation between two statements without procedure calls, which is the topic of the following section.

The modular verification methodology advocated in this section is designed to decompose the problem of verifying a large multithreaded program into a set of smaller and more manageable problems, one for each procedure. The verification obligation for a procedure depends on the call tree of the entire program. Hence, a module might have to be re-verified if changes in the implementation of another module results in a modification of the call tree.

5 Checking simulation

We first consider the simpler problem of checking that the atomic operation $p?X$ is simulated by $q?Y$. This simulation holds if (1) whenever $p?X$ goes wrong, then $q?Y$ also goes wrong, i.e., $\neg p \Rightarrow \neg q$, and (2) whenever $p?X$ performs a transition, $q?Y$ can perform a corresponding transition or may go wrong, i.e., $p \wedge X \Rightarrow \neg q \vee Y$. The conjunction of these two conditions can be simplified to $(q \Rightarrow p) \wedge (q \wedge X \Rightarrow Y)$.

The following atomic operation $\mathit{sim}(p?X, q?Y)$ checks simulation between the atomic operations $p?X$ and $q?Y$; it goes wrong from states for which $p?X$ is not simulated by $q?Y$, blocks in states where $q?Y$ goes wrong, and otherwise behaves like $p?X$. The definition uses the notation $\forall \mathit{Var}'$ to quantify over all primed (post-state) variables.

$$\mathit{sim}(p?X, q?Y) \quad \stackrel{\text{def}}{=} \quad ((q \Rightarrow p) \wedge (\forall \mathit{Var}'.\ q \wedge X \Rightarrow Y))?(q \wedge X)$$

We now extend our method to check simulation between an implementation B and an abstraction A with respect to an environment assumption E. Let I be the invariant associated with the implementation B; e.g., if B is $\mathit{InlineAbs}(\mathcal{B}(m))$ for some procedure m, then I is $\mathcal{I}(m)$. We assume that the abstraction A consists of n atomic operations $I?Y_1, I?Y_2, \ldots, I?Y_n$ interleaved with stuttering steps $I?K$, preceded by an asserted precondition $\mathit{pre}?\langle\texttt{true}\rangle$, and ending with the assumed postcondition $\texttt{true}?\langle post\rangle$:

$$\begin{array}{lll} A & \stackrel{\text{def}}{=} & pre?\langle\texttt{true}\rangle; \\ & & (I?K^*; I?Y_1);\ \ldots;\ (I?K^*; I?Y_n); \\ & & I?K^*; \texttt{true}?\langle post\rangle \end{array}$$

This restriction on A enables efficient simulation checking and has been sufficient for all our case studies. Our method may be extended to more general abstractions A at the cost of additional complexity.

Our method translates B, A, and E into a sequential program such that if that program does not go wrong, then B is simulated by A with respect to E. We need to check that whenever B performs an atomic operation, the statement A performs a corresponding operation. In order to perform this check, the programmer needs to add an *auxiliary* variable pc ranging over $\{1, 2, \ldots, n+1\}$ to B, so that each atomic operation in B updates pc as well as the original program variables. The value of pc indicates the operation in A that will simulate the next operation performed in B. The variable pc is initialized to 1. An atomic operation in B can either leave pc unchanged or increment it by 1. If the operation leaves pc unchanged, then the corresponding operation in A is K. If the operation changes pc from i to $i+1$, then the corresponding operation in A is Y_i. Thus, each atomic operation in B needs to be simulated by the following atomic operation:

$$W \stackrel{\text{def}}{=} I?(\bigvee_{i=1}^{n} (pc = i \wedge pc' = i+1 \wedge Y_i) \vee (pc = pc' \wedge K))$$

Using the above method, we generate the sequential program $[\![B]\!]_A^E$ which performs the simulation check at each atomic action, and also precedes each atomic action with the iterated environment assumption that models the interleaved execution of other threads. Thus, the program $[\![B]\!]_A^E$ is obtained by replacing every atomic operation $p?X$ in the program B with the code $PreserveLocals(E^*); sim(p?X, W)$. The following program extends $[\![B]\!]_A^E$ with constraints on the initial and final values of pc.

$$\texttt{assume}\ pre \wedge pc = 1; [\![B]\!]_A^E; PreserveLocals(E^*); \texttt{assert}\ post \wedge pc = n+1$$

This program starts execution from the set of states satisfying the precondition pre and asserts the postcondition $post$ at the end. Note that this sequential program is parameterized by the thread identifier `tid`. If this program cannot go wrong for any non-zero value of `tid`, then we conclude that B is simulated by A with respect to E. We leverage existing sequential analysis techniques (based on verification conditions and automatic theorem proving) for this purpose.

6 Implementation

We have implemented our modular verification method for multithreaded Java programs in an automatic checking tool called Calvin. This section provides an overview of Calvin, including a description of its annotation language and various performance optimizations that we have implemented.

6.1 Checker architecture

The Calvin checker takes as input a Java program, together with annotations describing candidate environment assumptions, procedure abstractions, invariants, and asserted correctness properties, and outputs warnings and error messages indicating if any of these properties are violated. Calvin starts by parsing the input program to produce abstract syntax trees (ASTs). After type checking, these abstract syntax trees are translated into an intermediate representation language that can express Plato syntax [31]. The translation of annotations into Plato syntax is described in Section 6.3.

Calvin then uses the techniques of this paper, as summarized by Theorem 1, to verify this intermediate representation of the program. To verify that each procedure p satisfies its specification, Calvin first inlines the abstraction of any procedure call from p. (If the abstraction is not available, then the implementation is inlined instead.) Next, Calvin uses the simulation checking technique of the previous section to generate a sequential "simulation checking" program S.

To check the correctness of S, a verification condition is generated according to the following translation,[1] which is based on Dijkstra's weakest precondition translation [15].

$$\begin{aligned}
vc(p?X, Q) &= p \wedge \forall \vec{x'}.\ A(\vec{x}, \vec{x'}) \implies Q[\vec{x} := \vec{x'}] \\
&\quad \text{where } \vec{x} \text{ denotes the variables modified by } A \\
vc(x := e, Q) &= Q[x := e] \\
vc(S_1; S_2, Q) &= vc(S_1, vc(S_2, Q)) \\
vc(S_1 \Box S_2, Q) &= vc(S_1, Q) \vee vc(S_2, Q) \\
vc(S^*, Q) &= vc(\mathtt{skip}\Box(S; (\mathtt{skip}\Box S)), Q)
\end{aligned}$$

[1] Note that this translation may generate exponentially-large verification conditions. To avoid this problem, Calvin actually uses a semantically-equivalent translation that generates compact verification conditions, as described in an earlier paper [24]. A detailed description of that translation is outside the scope of this paper.

This translation can handle arbitrary atomic operations, but uses a specialized translation for particular atomic operations such as assignments. Following ESC/Java, Calvin provides two options for translating loops. One option is for the programmer to explicitly provide a loop invariant. A second, more convenient option, which we used in our experiments, is simply to unroll each loop a small number of times, as shown in the above translation. Although unsound, this approach has proved adequate in practice to detect a range of defects using both ESC/Java and Calvin.

The generated verification condition is then fed into the theorem prover Simplify [38,14]. This theorem prover is fully-automatic and requires no interaction with the programmer. It may, however, fail to terminate, in which case Calvin reports a time-out after five minutes. If the theorem prover detects that the verification condition is invalid, then it generates a counterexample, which is then post-processed into an appropriate error message in terms of the original Java program. Typically, the error message either identifies an atomic step that may violate one of the stated invariants, environment assumptions, or abstraction steps, or the error message may identify an assertion that could go wrong. This assertion may either be explicit, as in the example programs of Section 3, or implicit, such as, for example, that a dereferenced pointer is never null. Conversely, if the theorem prover verifies the validity of the verification condition, then Calvin concludes that the procedure implements its specification and that the stated invariants and assertions are true.

The implementation of Calvin leverages extensively off the Extended Static Checker for Java, which is a powerful checking tool for sequential Java programs. For more information regarding ESC/Java, we refer the interested reader to a recent paper [22].

6.2 Handling Java threads and monitors

In our implementation, thread identifiers are either references to objects of type `java.lang.Thread` or a special value `main` (different from all object references) that refers to the program's initial thread. Thread creation is modeled by introducing an abstract instance field[2] `start` into the `java.lang.Thread` class. When a thread is created, this field is initialized to false. When a created thread is forked, this field is set to true. The following assume statement is implicit at the beginning of the main method:

$$\texttt{assume tid} = \texttt{main}$$

[2] An abstract variable is one that is used only for specification purposes and is not originally present in the implementation.

The following assume statement is implicit at the beginning of the run method in any runnable class:

$$\texttt{assume tid} = \texttt{this} \wedge \texttt{tid.start}$$

The implicit lock associated with each Java object is modeled by including in each object an additional abstract field `holder` of type `java.lang.Thread`, which is either null or refers to the thread currently holding the lock. The Java synchronization statement `synchronized(x)` $\{$ `S` $\}$ is desugared into

$$\begin{array}{l} \langle \texttt{x.holder} = \texttt{null} \wedge \texttt{x.holder}' = \texttt{tid} \rangle_{\texttt{x.holder}}; \\ \;\texttt{S} \;; \\ \langle \texttt{x.holder}' = \texttt{null} \rangle_{\texttt{x.holder}} \end{array}$$

For the sake of simplicity, our checker assumes a sequentially consistent memory model and that reads and writes of primitive Java types are atomic.

6.3 Annotation language

This section describes the source annotations for each procedure p. The annotation `env_assumption` provides environment assumptions. Each class may have multiple such annotations, each of which provides an action (that may refer to `tid`). The environment assumption of a class is the conjunction of all these actions. The environment assumption $\mathcal{E}(p)$ of a method p is the conjunction of the environment assumption of the class containing p and of all classes whose methods are transitively called by p.

The annotation `global_invariant` provides invariants. Each class may have multiple such annotations, with each annotation providing a predicate. The invariant of a class is the conjunction of the predicates in all these annotations. The invariant of a method p is the invariant of the class containing p.

The abstraction of a method p is specified using the following notation:

$$\begin{array}{l} \texttt{requires } \mathit{pre} \\ \texttt{modifies } c \\ \texttt{action: also\_modifies } v_1 \texttt{ ensures } e_1 \\ \ldots \\ \texttt{action: also\_modifies } v_n \texttt{ ensures } e_n \\ \texttt{ensures } \mathit{post} \end{array}$$

where $c, v_1, \ldots, v_n$ are sets of variables, *pre* is a single-store predicate, and $e_1, \ldots, e_n, \mathit{post}$ are actions.

From the above notation, we construct the abstraction statement $\mathcal{A}(p)$ as follows:

(1) We construct the following guarantee G based on the assumption that actions of p should not violate the environment assumptions of p for other threads.

$$G \stackrel{\text{def}}{=} \forall \texttt{Thread}\ j : (j \neq \mathit{null} \wedge j \neq \texttt{tid}) \Rightarrow \mathcal{E}(p)[\texttt{tid} := j]$$

(2) If I is the invariant of p, we combine the various annotations into the following abstraction statement $\mathcal{A}(p)$:

$$\begin{array}{l} \mathit{pre}?\langle\texttt{true}\rangle; \\ I?\langle G \wedge I'\rangle_c{}^*; I?\langle e_1 \wedge G \wedge I'\rangle_{c\cup v_1}; \\ \ldots \\ I?\langle G \wedge I'\rangle_c{}^*; I?\langle e_n \wedge G \wedge I'\rangle_{c\cup v_n}; \\ I?\langle G \wedge I'\rangle_c{}^*; \\ \texttt{true}?\langle \mathit{post}\rangle \end{array}$$

The stuttering steps should satisfy G and only modify variables in c. Each `action:` block in the annotations corresponds to an atomic operation in the abstraction; this atomic operation can modify variables in c and v_i, it should satisfy both e_i and the guarantee G, and the `requires` action *pre* is asserted to hold initially. Finally, every step is required to maintain the invariance of I.

Comparing $\mathcal{A}(p)$ with the notation in Section 5, we see that Y_i is $\langle e_1 \wedge G \wedge I'\rangle_{c\cup v_1}$ and K is $\langle G \wedge I'\rangle_c$.

6.4 Optimizations

Calvin reduces simulation checking to the correctness of the sequential "simulation checking" program. The simulation checking program is often significantly larger than the original procedure implementation, due in part to the iterated environment assumption inserted before each atomic operation. To reduce verification time, Calvin simplifies the program before attempting to verify it. In particular, we have found the following two optimizations particularly useful for simplifying the simulation checking program:

- In all our case studies, the environment assumptions were reflexive and transitive. Therefore, our checker optimizes the iterated environment assumption E^* to the single action E after using the automatic theorem prover to verify that E is indeed reflexive and transitive.
- The environment assumption of a procedure can typically be decomposed into a conjunction of actions mentioning disjoint sets of variables, and any

two such actions commute. Moreover, assuming the original assumption is reflexive and transitive, each of these actions is also reflexive and transitive. Consider an atomic operation that accesses a single shared variable `v`. An environment assertion is inserted before this atomic operation, but all actions in the environment assumption that do not mention `v` can be commuted to the right of this operation, where they merge with the environment assumption associated with the next atomic operation. Thus, we only need to precede each atomic operation with the actions that mention the shared variable being accessed.

7 Applications

7.1 The Apprentice challenge problem

Moore and Porter [37] introduced the Apprentice example as a challenge problem for multithreaded software analysis tools. The Apprentice example contains three classes: `Container`, `Job` and `Apprentice` (see Figure 7). The class `Container` has an integer field `counter`. The class `Job`, which extends `Thread`, has a field `objref` pointing to a `Container` object. The class `Apprentice` contains the `main` routine.

After k iterations of the loop in `main`, there are $k + 1$ concurrently executing threads consisting of one main thread and k instances of `Job`. We would like to prove that in any concurrent execution the field `counter` of any instance of `Container` takes a sequence of non-decreasing values.[3] This property is stated by the following annotation in the `Container` class.

```
/*@ env_assumption \old(counter) <= counter */
```

Note that this property could be violated in several ways. A thread t executing the method t`.run` reads t`.objref` thrice during one iteration of the loop:

(1) to obtain the monitor on the object pointed to by t`.objref`,
(2) to read t`.objref.counter`, and
(3) to write t`.objref.counter`.

If another thread modifies t`.objref` from o_1 to o_2 between the second and third reads, then the value written by thread t into o_2`.counter` may be less than its previous value. Moreover, even if other threads do not modify t`.objref`, they might increment t`.objref.counter` more than once between the read

[3] Calvin treats the `int` type as unbounded unlike the 32-bit semantics in Java.

Figure 7: The Apprentice challenge

```
class Container {    int counter;    }

class Job extends Thread {
    Container objref;

    public final void run() {
        for (;;) {
            synchronized(objref) { objref.counter = objref.counter + 1; }
        }
    }
}

class Apprentice {
    public static void main(String[] args) {
        Container container = new Container();
        for (;;) {
            Job job = new Job();
            job.objref = container;
            job.start();
        }
    }
}
```

and the write of t.`objref.counter`. This interference might again cause a similar violation.

The environment assumption stated above is not strong enough to analyze each thread separately in Calvin. We also need to specify the conditions under which the environment of a thread can modify the fields `counter` and `objref`. We add the annotation

```
/*@ unwritable_by_env_if holder == tid */
```

to the field `counter` to indicate that for any instance o of `Container`, if thread t holds the monitor on o then the environment of t may not modify o.`counter`. Thus, `unwritable_by_env_if` annotations provide a simple and concise way of writing environment assumptions. For example, the `unwritable_by_env_if` annotation shown above on the field `counter` is semantically equivalent to the following annotation:

```
/*@ env_assumption holder == tid ==> counter == \old(counter) */
```

We also add the annotation

```
/*@ unwritable_by_env_if tid == main || objref != null */
```

to the field `objref`. In this annotation, `main` refers to the main thread. This annotation specifies that for any instance o of `Job`, the environment of `main` must not modify o`.objref`. In addition, even `main` must not modify o`.objref` if o`.objref` is different from `null`. Using these annotations, Calvin is successfully able to verify the original environment assumption together with the environment assumptions induced by these annotations.

We now introduce a bug in the Apprentice example as suggested by Moore and Porter.

```
public static void main(String[] args) {
    Container container = new Container();
    Container bogus = new Container();
    for (;;) {
        Job job = new Job();
        job.objref = container;
        job.start();
        job.objref = bogus;
    }
}
```

In this buggy implementation, the thread `main` mutates `job.objref` again after `job` has started. As mentioned above, such behavior might cause the `counter` field of some `Container` object to decrease.

Calvin produces the following warning for the modified Apprentice example:

```
Apprentice.java:29: Warning: Write of variable when not allowed
        job.objref = bogus;

Associated declaration is "Apprentice.java", line 9, col 8:
    /*@ unwritable_by_env_if (tid == main || objref != null) */
```

This warning indicates that `main` violates the requirement that `job.objref` should not be modified once it has been initialized.

7.2 The Mercator web crawler

Mercator [26] is a web crawler which is part of Altavista's Search Engine 3 product. It is multithreaded and written entirely in Java. Mercator spawns a number of *worker* threads to perform the web crawl and write the results to shared data structures in memory and on disk. To help recover from failures, Mercator also spawns a *background* thread that writes a snapshot of its state to disk at regular intervals. Synchronization between these threads is achieved using two kinds of locks: mutual exclusion locks and *readers-writer* locks.

We focused our analysis efforts on the part of Mercator's code (about 1500 LOC) that uses readers-writer locks. We first provided a specification of the readers-writer lock implementation (class `ReadersWriterLock`) in terms of two abstract variables—`writer`, a reference to a `Thread` object, and `readers`, a set of references to `Thread` objects. If a thread owns the lock in write mode then `writer` contains a reference to that thread and `readers` is empty, otherwise `writer` is `null` and `readers` is the set of references to all threads that own the lock in read mode.

Consider the procedure `beginWrite` that acquires the lock in write mode by setting a program variable `hasWriter` of type `boolean`. The specification of `beginWrite` and the corresponding Plato code are shown below.

Figure 8: Specifying readers-writer lock

```
/*@
requires holder == tid
modifies hasWriter
action:
  also_modifies writer
  ensures writer == null
       && writer' == tid
*/
public void beginWrite() {
 ...
}
```

$$\begin{array}{l}
\mathtt{holder} = \mathtt{tid}?\langle \mathtt{true} \rangle; \\
\mathtt{true}?\langle \mathtt{true} \rangle_{\mathtt{hasWriter}}{}^{*}; \\
\mathtt{true}?\left\langle \begin{array}{l} \mathtt{writer} = \mathtt{null} \\ \wedge\ \mathtt{writer}' = \mathtt{tid} \end{array} \right\rangle_{\{\mathtt{hasWriter},\mathtt{writer}\}}; \\
\mathtt{true}?\langle \mathtt{true} \rangle_{\mathtt{hasWriter}}{}^{*}
\end{array}$$

The next step was to annotate and check the clients of `ReadersWriterLock` to ensure that they follow the synchronization discipline for accessing shared data. The part of Mercator that we analyzed uses two readers-writer locks—`L1` and `L2`. We use the following `unwritable_by_env_if` annotation to state that before modifying the variable `tbl`, the background thread should always acquire lock `L1` in write mode, but a worker thread need only acquire the mutex on lock object `L2`.

```
/*@ unwritable_by_env_if   (tid == backgroundThread && L1.writer == tid)
                         || (tid instanceof Worker && L2.holder == tid) */
private long[][]tbl; // the in-memory table
```

We also provided specifications of public methods that can access the shared data and used inlining to avoid annotating non-public methods.

Overall, we needed to insert 55 annotations into the source code. The majority of these annotations (21) were needed to specify and prove the implementation of readers-writer locks. However, once the readers-writer class is specified, its specification can be re-used when checking many clients of this class.

Interface annotations (apart from those in `ReadersWriterLock`) numbered 16, and largely consisted of constraints on the type of thread that could call a method, and about locks that needed to be held on entry to a method.

We did not find any bugs in the part of Mercator that we analyzed; however, we injected bugs of our own, and Calvin located those. In spite of inlining all non-public methods, the analysis took less than 10 minutes for all except one public method. The exception was a method of 293 lines (after inlining non-public method calls), on which the theorem prover ran overnight to report no errors.

7.3 The java.util.Vector library

We ran Calvin on the class `java.util.Vector` (about 400 LOC) from JDKv1.2. There are two shared fields: an integer `elementCount`, which contains the number of elements in the vector, and an array `elementData`, which stores the elements. These variables are protected by the lock on the `Vector` object.

```
/*@ unwritable_by_env_if this.holder == tid */
protected int elementCount;
/*@ unwritable_by_env_if this.holder == tid */
protected Object elementData[];

/*@ global_invariant 0 <= elementCount && elementCount <= elementData.length */
/*@ global_invariant elementData != null */
```

Based on the specifications, Calvin detected a race condition illustrated in the following excerpt.

```
public int lastIndexOf(Object elem) {
    return lastIndexOf(elem, elementCount-1); // RACE!
}
public synchronized int lastIndexOf(Object elem, int index) {
        ....
        for (int i = index; i >= 0; i--)
          if (elem.equals(elementData[i]))
        ....
}
....
synchronized void trimToSize() { ... }
synchronized boolean removeAllElements() { ... }
```

Suppose there are two threads manipulating a `Vector` object `v`. The first thread calls `v.lastIndexOf(Object)`, which reads `v.elementCount` without acquiring the lock on `v`. Now suppose that before the first thread calls `lastIndexOf(Object,int)`, the second thread calls `v.removeAllElements()`, which sets `v.elementCount` to 0, and then calls `trimToSize()`, which resets `v.elementData` to be an array of length 0. Then, when the first thread tries to access `v.elementData` based on the old value of `v.elementCount`, it will trigger an array out-of-bounds exception. An erroneous fix for this race condition is as follows:

```
public int lastIndexOf(Object elem) {
    int count;
    synchronized(this) { count = elementCount-1; }
    return lastIndexOf(elem, count);
}
```

Even though the lock is held when `elementCount` is accessed, the original defect still remains. RCC/Java [19], a static race detection tool, caught the original defect in the `Vector` class, but will not catch the defect in the modified code. Calvin, on the other hand, still reports this error as what it is: a potential array out-of-bounds error. The defect can be correctly fixed by declaring `lastIndexOf(Object)` to be `synchronized`.

8 Related work

A variety of static and dynamic checkers have been built for detecting data races in multithreaded programs [4,10,44,41,22]; however, these tools are limited to checking a subset of the synchronization mechanisms found in systems code. For example, RCC/Java [19,20] is an annotation-based checker for Java that uses a type system to identify data races. While this tool is successful at finding errors in large programs, the inability to specify subtle synchronization patterns results in false alarms. Moreover, these tools cannot verify invariants or check refinement of abstractions. The methods proposed by Engler et al. [17,18] for checking and inferring simple rules on code behavior are scalable and surprisingly effective, but cannot check general invariants.

Several tools verify invariants on multithreaded programs using a combination of abstract interpretation and model checking. The Bandera toolkit [16] uses programmer-supplied data abstractions to translate multithreaded Java programs into the input languages of various model checkers. Yahav [46] describes a method to model check multithreaded Java programs using a 3-valued logic [40] to abstract the store. Since these tools explicitly consider all interleavings of the multiple threads, they have difficulty scaling to large programs. Ball et al. [8] present a technique for model checking a software library with an unspecified number of threads that are identical and finite-state. Bruening [11] has built a dynamic assertion checker based on state-space exploration for multithreaded Java programs. His tool concurrently runs an Eraser-like [42] race detector to ensure the absence of races, which guarantees that `synchronized` code blocks can be considered atomic. Stoller [45] provides a generalization of Bruening's method to allow model checking of programs with either message-passing or shared-memory communication. Both of these approaches focus on mutex-based synchronization and operate on the concrete program without any abstraction.

The compositional principle underlying our technique is assume-guarantee reasoning, of which there are several variants. One of the earliest assume-guarantee proof rules was developed by Misra and Chandy [35] for message-passing systems, and later refined by others (e.g., [29,39,36]). However, their message-passing formulation is not directly applicable to shared-memory software.

The most closely related previous work is that by Jones [28] and by Abadi and Lamport [1]. Jones [28,27] gave a proof rule for multithreaded shared-memory programs and used it to manually refine an assume-guarantee specification down to a program. This proof rule of Jones allows each thread in a multithreaded program to be verified separately, but the program for each thread does not have any procedure calls. We have extended Jones' work to allow the proof obligations for each thread to be checked mechanically by an automatic theorem prover, and our extension also handles procedure calls. The use of assume-guarantee reasoning to analyze multithreaded Java programs has also been explored by Erika Abraham *et al* [3,2]. Their approach is based on an extension of Hoare-style triples, and so requires assertions at each program point.

Stark [43] also presented a rule for shared-memory programs to deduce that a conjunction of assume-guarantee specifications hold on a system provided each specification holds individually, but his work did not allow the decomposition of the implementation. Compositional techniques similar to assume-guarantee reasoning have been used to perform refinement in the setting of action systems as well [7].

Abadi and Lamport [1] consider a composition of components, where each component modifies a separate part of the store. Their system is general enough to model a multithreaded program since a component can model a collection of threads operating on shared state and signaling among components can model procedure calls. However, their proof rule does not allow each thread in a component to be verified separately. Collette and Knapp [13] extend Abadi and Lamport's approach to the more operational setting of Unity specifications [12]. Alur and Henzinger [5] and McMillan [34] have presented assume-guarantee proof rules for hardware components.

In recent work [25], we have begun to explore an extension to the abstraction mechanism presented here. We augment simulation-based abstraction with the notion of reduction, which was first introduced by Lipton [32]. Reduction permits us to identify sequences of steps in a procedure that are guaranteed to execute without interference. Such "atomic" sequences can be summarized by a single step in procedure specifications, thereby making specifications more concise in some cases.

9 Conclusions

We have presented a new methodology for modular verification of multithreaded programs, based on combining the twin principles of thread-modular reasoning and procedure-modular reasoning. Our experience with Calvin, an implementation of this methodology for multithreaded Java programs, shows that it is scalable and sufficiently expressive to check interesting properties of real-world multithreaded systems code.

Acknowledgments Cormac Flanagan and Stephen Freund were partly supported by the National Science Foundation under Grants CCR-0341179 and CCR-0341387.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[2] E. Abrahm, F. S. de Boer, W. P. de Roever, and M. Steffen. A compositional operational semantics for javamt. In N. Dershowitz, editor, *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 290–303. Springer, 2003.

[3] E. Abrahm, F. S. de Boer, W. P. de Roever, and M. Steffen. Inductive proof outlines for monitors in java. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems, 6th IFIP WG 6.1 International Conference, FMOODS 2003*, volume 2884 of *Lecture Notes in Computer Science*, pages 155–169. Springer, 2003.

[4] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th Symposium on Principles of Programming Languages*, pages 243–354, 1998.

[5] R. Alur and T. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[6] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[7] R.-J. Back and J. von Wright. Compositional action system refinement. *Formal Aspects of Computing*, 15(2–3):103–117, 2003.

[8] T. Ball, S. Chaki, and S. Rajamani. Parameterized verification of multithreaded software libraries. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, April 2001.

[9] A. Birrell, J. Guttag, J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: A formal specification. Research Report 20, DEC Systems Research Center, 130 Lytton Ave, Palo Alto, CA 94301, USA, August 1987.

[10] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of 16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 56–69, Tampa Bay, FL, October 2001.

[11] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, Massachusetts Institute of Technology, 1999.

[12] K. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, 1988.

[13] P. Collette and E. Knapp. Logical foundations for compositional verification and development of concurrent programs in Unity. In *Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science 936, pages 353–367. Springer-Verlag, 1995.

[14] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.

[15] E. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[16] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, 2001.

[17] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation(OSDI)*, October 2000.

[18] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[19] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[20] C. Flanagan and S. N. Freund. Detecting race conditions in large programs. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 90–96, June 2001.

[21] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proceedings of European Symposium on Programming*, pages 262–277, April 2002.

[22] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.

[23] C. Flanagan, S. Qadeer, and S. Seshia. A modular checker for multithreaded programs. In *CAV 02: Computer Aided Verification*, July 2002.

[24] C. Flanagan and J. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th Symposium on Principles of Programming Languages*, pages 193–205. ACM, Jan. 2001.

[25] S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. In *Workshop on Formal Techniques for Java-like Programs*, 2003. An extended version has been submitted to a special issue of the *Journal of Object Technology* dedicated to papers from this workshop.

[26] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. In *Proceedings of World Wide Web conference*, pages 219–229, December 1999.

[27] C. Jones. Specification and design of (parallel) programs. In R. Mason, editor, *Information Processing*, pages 321–332. Elsevier Science Publishers B. V. (North-Holland), 1983.

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

[29] B. Jonsson. On decomposing and refining specifications of distributed systems. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Lecture Notes in Computer Science 430, pages 361–385. Springer-Verlag, 1989.

[30] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[31] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999.

[32] R. Lipton. Reduction: A method of proving properties of parallel programs. In *Communications of the ACM*, volume 18:12, pages 717–721, 1975.

[33] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[34] K. McMillan. A compositional rule for hardware design refinement. In O. Grumberg, editor, *CAV 97: Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 24–35. Springer-Verlag, 1997.

[35] J. Misra and K. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[36] A. Mokkedem and D. Mery. On using a composition principle to design parallel programs. In *Algebraic Methodology and Software Technology*, pages 315–324, 1993.

[37] J. S. Moore and G. Porter. The apprentice challenge. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):193–216, 2002.

[38] C. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, 1981.

[39] P. Pandya and M. Joseph. P-A logic: A compositional proof system for distributed programs. *Distributed Computing*, 5(1), 1991.

[40] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[41] S. Savage, M. Burrows, C. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[42] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[43] E. Stark. A proof technique for rely/guarantee properties. In *Proceedings of the 5th Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 206, pages 369–391. Springer-Verlag, 1985.

[44] N. Sterling. WARLOCK — a static data race analysis tool. In *USENIX Technical Conference Proceedings*, pages 97–106, Winter 1993.

[45] S. D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proceedings of the 7th International SPIN Workshop on Model Checking and Software Verification*, Lecture Notes in Computer Science 1885, pages 224–244. Springer-Verlag, 2000.

[46] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proceedings of the 28th Symposium on Principles of Programming Languages*, pages 27–40, January 2001.

A Proof of modular verification theorem

Lemma 1 *If the statement* $l()$ *is simulated by the statement* $HavocLocals(\hat{\mathcal{A}}(l))$ *with respect to* $\hat{\mathcal{E}}(l)$*, then the program* $\|l()$ *is simulated by the program* $\|HavocLocals(\hat{\mathcal{A}}(l))$*.*

Proof Let

$$
\begin{aligned}
P &\stackrel{\text{def}}{=} \|l() \\
Q &\stackrel{\text{def}}{=} \|HavocLocals(\hat{\mathcal{A}}(l)) \\
P_j &\stackrel{\text{def}}{=} \mathcal{P}(l(), \hat{\mathcal{E}}(l), j) \\
Q_j &\stackrel{\text{def}}{=} \mathcal{P}(HavocLocals(\hat{\mathcal{A}}(l)), \hat{\mathcal{E}}(l), j)
\end{aligned}
$$

We prove that if τ is a trace of P, then there is a trace τ' of Q such that (1) τ is subsumed by τ', and (2) if τ' does not go wrong, then τ is a trace of P_j for all $1 \le j \le n$. The proof is by induction on the length of τ.

- *Base Case:* Let $\tau = \omega$. This trivial trace clearly satisfies the desired property.
- *Induction Step:* Suppose τ corresponds to a run r_a of P, where

$$r_a = (\sigma_0, z_0^a) \xrightarrow{|t_1, a_1|} (\sigma_1, z_1^a) \cdots (\sigma_{k-1}, z_{k-1}^a) \xrightarrow{|t_k, a_k|} (\sigma_k, z_k^a) \xrightarrow{|j, a|} (\omega_a, z^a)$$

Let r be the prefix of r_a that excludes the last transition. By the induction hypothesis, there is a run r_d of Q given by

$$r_d = (\sigma_0, z_0^d) \xrightarrow{|t_1, d_1|} (\sigma_1, z_1^d) \cdots (\sigma_{l-1}, z_{l-1}^d) \xrightarrow{|t_l, d_l|} (\omega_d, z^d)$$

such that $trace(r_d)$ subsumes $trace(r)$.

If $\omega_d = \textbf{wrong}$, then $trace(r_d)$ also subsumes $trace(r_a) = \tau$ and we are done.

Otherwise $\omega_d = \sigma_k \neq \textbf{wrong}$, $l = k$, and there is a run r_b of P_j given by

$$r_b = (\sigma_0, z_0^b) \xrightarrow{|t_1, b_1|} (\sigma_1, z_1^b) \cdots (\sigma_{k-1}, z_{k-1}^b) \xrightarrow{|t_k, b_k|} (\sigma_k, z_k^b)$$

First, we prove that τ is subsumed by a trace of Q. A run r_{ab} of P_j can be obtained from r_a and r_b by replacing actions of thread j in r_b by corresponding actions of thread j in r_a and adding the last action of thread j in r_a to the end of r_b. This run r_{ab} has the property that $trace(r_{ab}) = trace(r_a) = \tau$. Since P_j is simulated by Q_j, there is a run of Q_j given by

$$r_c = (\sigma_0, z_0^c) \xrightarrow{|t_1, c_1|} (\sigma_1, z_1^c) \cdots (\sigma_{m-1}, z_{m-1}^c) \xrightarrow{|t_m, c_m|} (\sigma_m, z_m^c) \xrightarrow{|j, c|} (\omega_c, z^c)$$

such that $trace(r_c)$ subsumes $trace(r_{ab}) = \tau$. A run r_{cd} of Q can be obtained from r_c and r_d by replacing actions of thread j in r_d by corresponding actions of thread j in r_c. If $m = k$, we also append the last action of thread j in r_c to r_d. This run r_{cd} has the property that $trace(r_{cd}) = trace(r_c)$ and therefore it subsumes τ.

We now prove that if $\omega_c \neq \textbf{wrong}$, then τ is a trace of P_i for all $i \in Tid$. If $\omega_c \neq \textbf{wrong}$, then $m = k$ and $\omega_c = \omega_a$ and $trace(r_a) = trace(r_c) = \tau$. Thus we get that τ is a trace of P_j. Now, pick $i \in Tid$ such that $i \neq j$. By the induction hypothesis, there is a run r_e of P_i given by

$$r_e = (\sigma_0, z_0^e) \xrightarrow{|t_1,e_1|} (\sigma_1, z_1^e) \cdots (\sigma_{k-1}, z_{k-1}^e) \xrightarrow{|t_k,e_k|} (\sigma_k, z_k^e)$$

We have shown that there is a transition of Q of the form $\sigma_k \xrightarrow{|j,d|} \omega_a$. From the definition of Q, the atomic operation d is of the form $\hat{p}?\hat{X}$ where $\hat{p} \implies \mathcal{I}(l)$ and $\hat{X} \implies (\forall i \in Tid : i \neq \mathtt{tid} \Rightarrow \hat{\mathcal{E}}(l))$. Thus, if $\omega_a \neq \textbf{wrong}$, then $\hat{\mathcal{E}}(l)(j, \sigma_k, \omega_a)$ holds. Therefore, the run r_e of P_i can be extended to

$$(\sigma_0, z_0^e) \xrightarrow{|t_1,e_1|} (\sigma_1, z_1^e) \cdots (\sigma_{k-1}, z_{k-1}^e) \xrightarrow{|t_k,e_k|} (\sigma_k, z_k^e) \xrightarrow{|j,\hat{\mathcal{E}}(l)|} (\omega_a, z^e)$$

and we get that τ is a trace of P_i.

□

Lemma 2 *If a statement S is simulated by a statement T with respect to environment assumption E and E' implies E, then S is simulated by T with respect to E'.*

Proof

Fix $j \in Tid$ and let

$$\begin{aligned}
P_j &\stackrel{\text{def}}{=} \mathcal{P}(S, E, j) \\
Q_j &\stackrel{\text{def}}{=} \mathcal{P}(T, E, j) \\
P'_j &\stackrel{\text{def}}{=} \mathcal{P}(S, E', j) \\
Q'_j &\stackrel{\text{def}}{=} \mathcal{P}(T, E', j)
\end{aligned}$$

Consider a run $r = (\sigma_0, z_0) \xrightarrow{|t_1,a_1|} (\sigma_1, z_1) \ldots \xrightarrow{|t_m,a_m|} (\omega, z)$ of P'_j, for arbitrary j. Consider all transitions $\sigma_{i-1} \xrightarrow{|t_i,a_i|} \sigma_i$ in r where $t_i \neq j$. For each such transition, $E'(j, \sigma_{i-1}, \sigma_i)$ holds. Since, E' implies E, $E(j, \sigma_{i-1}, \sigma_i)$ holds. Therefore, r is a run of P_j.

Since $P_j \sqsubseteq Q_j$, there exists a run $r' = (\sigma_0, z'_0) \xrightarrow{|t_1,b_1|} (\sigma_1, z'_1) \ldots \xrightarrow{|t_n,b_n|} (\omega', z')$ of Q_j such that $trace(r')$ subsumes $trace(r)$. Consider any transition $\sigma_{i-1} \xrightarrow{|t_i,b_i|}$

σ_i in r' where $t_i \neq j$. Since $\mathit{trace}(r') = \mathit{trace}(r)$, both $E(j, \sigma_{i-1}, \sigma_i)$ and $E'(j, \sigma_{i-1}, \sigma_i)$ hold. Therefore, r' is also a run of Q'_j.

Thus, we get $P'_j \sqsubseteq Q'_j$ for all $j \in \mathit{Tid}$ and thereby $S \sqsubseteq_{E'} T$.

□

We introduce some additional notation for the remainder of this appendix. Let $\mathcal{P}^d(B, E, j)$ be the parallel program in which the j-th thread executes B with the depth of its stack bounded by d and every other thread executes $E^*[\mathtt{tid} := j]$. We write $B \sqsubseteq^d_E A$ to indicate that the program $\mathcal{P}^d(B, E, j)$ is simulated by the program $\mathcal{P}^d(A, E, j)$ for all $j \in \mathit{Tid}$.

Let $\bar{u}$ be a path that is the concatenation of n paths $\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_n$. Let $r_1, r_2, \ldots, r_{n-1}$ be full runs of $\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_{n-1}$ respectively, and let r_n be a run of $\bar{u}_n$, such that the last state in r_i is the first state of r_{i+1} for $1 \leq i < n$. Then, we denote the corresponding run r of $\bar{u}$ by $r_1; r_2; \ldots; r_n$.

Lemma 3 *Suppose for all $m \in \mathit{Proc}$, $\mathit{InlineAbs}(\mathcal{B}(m))$ is simulated by the statement $\mathit{HavocLocals}(\hat{\mathcal{A}}(m))$ with respect to the environment assumption $\hat{\mathcal{E}}(m)$. Then for all $d \in \mathbb{N}$, statements S, and environment assumptions E such that $E \Rightarrow \hat{\mathcal{E}}(l)$ whenever l is called by S, we have $S \sqsubseteq^d_E \mathit{InlineAbs}(S)$.*

Proof We proceed by induction over the depth d of the stack.

- *Base case:* Suppose $d = 0$. By the definition of $[\![S]\!]^0$ and $\mathit{InlineAbs}(S)$, we get $[\![S]\!]^0 \subseteq [\![\mathit{InlineAbs}(S)]\!]$. Therefore $S \sqsubseteq^0_E \mathit{InlineAbs}(S)$.
- *Induction step:* Suppose $d \geq 1$. We proceed by induction over the structure of S. Fix an E such that $E \Rightarrow \hat{\mathcal{E}}(m)$ whenever m is called by S. Also, fix $j \in \mathit{Tid}$.
  - $\underline{(S = a):}$ Then, $\mathit{InlineAbs}(S) = a$. Therefore, $[\![S]\!]^d = [\![\mathit{InlineAbs}(S)]\!]$, and so, $S \sqsubseteq^d_E \mathit{InlineAbs}(S)$.
  - $\underline{(S = S_1; S_2):}$ Consider a run r of $\mathcal{P}^d(S, E, j)$. There are two possible cases: (1) r is a run of $\mathcal{P}^d(S_1, E, j)$, or (2) $r = r_1; r_2$, r_1 is a full run of $\mathcal{P}^d(S_1, E, j)$, and r_2 is a run of $\mathcal{P}^d(S_2, E, j)$.

    Case 1. By the induction hypothesis, we have $S_1 \sqsubseteq^d_E \mathit{InlineAbs}(S_1)$. Therefore, there is a run r' of $\mathcal{P}(\mathit{InlineAbs}(S_1), E, j)$ such that $\mathit{trace}(r)$ is subsumed by $\mathit{trace}(r')$. Since r' is a run of $\mathcal{P}(\mathit{InlineAbs}(S_1), E, j)$, it is also a run of the program $\mathcal{P}(\mathit{InlineAbs}(S_1); \mathit{InlineAbs}(S_2), E, j)$.

    Case 2. By the induction hypothesis, we have that $S_1 \sqsubseteq^d_E \mathit{InlineAbs}(S_1)$ and $S_2 \sqsubseteq^d_E \mathit{InlineAbs}(S_2)$. Then there is a full run r'_1 of $\mathcal{P}(\mathit{InlineAbs}(S_1), E, j)$ such that $\mathit{trace}(r_1)$ is subsumed by $\mathit{trace}(r'_1)$. If r'_1 goes wrong, then r'_1 is also a run of $\mathcal{P}(\mathit{InlineAbs}(S_1); \mathit{InlineAbs}(S_2), E, j)$ and we are done.

    Otherwise $\mathit{trace}(r_1) = \mathit{trace}(r'_1)$. Further, there is also a run r'_2 of

$\mathcal{P}(\mathit{InlineAbs}(S_2), E, j)$ such that $\mathit{trace}(r_2)$ is subsumed by $\mathit{trace}(r_2')$. Let $r' = r_1'; r_2'$. Then, we get that $\mathit{trace}(r)$ is subsumed by $\mathit{trace}(r')$ and r' is a run of $\mathcal{P}(\mathit{InlineAbs}(S_1); \mathit{InlineAbs}(S_2), E, j)$.

Since $\mathit{InlineAbs}(S_1; S_2) = \mathit{InlineAbs}(S_1); \mathit{InlineAbs}(S_2)$, in both cases we get that r' is a run of $\mathcal{P}(\mathit{InlineAbs}(S_1; S_2), E, j)$.

- $\underline{(S = S_1 \Box S_2):}$ Consider a run r of $\mathcal{P}^d(S, E, j)$. Either r is a run of $\mathcal{P}^d(S_1, E, j)$ or r is a run of $\mathcal{P}^d(S_2, E, j)$. By the induction hypothesis, we get $S_1 \sqsubseteq_E^d \mathit{InlineAbs}(S_1)$ and $S_2 \sqsubseteq_E^d \mathit{InlineAbs}(S_2)$. If r is a run of $\mathcal{P}^d(S_1, E, j)$, then there is a run r' of $\mathcal{P}(\mathit{InlineAbs}(S_1), E, j)$ such that $\mathit{trace}(r)$ is subsumed by $\mathit{trace}(r')$. If r is a run of $\mathcal{P}^d(S_2, E, j)$, then there is a run r' of $\mathcal{P}(\mathit{InlineAbs}(S_2), E, j)$ such that $\mathit{trace}(r)$ is subsumed by $\mathit{trace}(r')$. Thus, there is a run r' of $\mathcal{P}(\mathit{InlineAbs}(S_1) \Box \mathit{InlineAbs}(S_2), E, j)$ such that $\mathit{trace}(r)$ is subsumed by $\mathit{trace}(r')$. Since we also know that $\mathit{InlineAbs}(S_1 \Box S_2) = \mathit{InlineAbs}(S_1) \Box \mathit{InlineAbs}(S_2)$, we get r' is a run of $\mathcal{P}(\mathit{InlineAbs}(S_1 \Box S_2), E, j)$.

- $\underline{(S = {S_1}^*):}$ Consider a run r of $\mathcal{P}^d(S, E, j)$. Then, for some $x > 0$, there are runs $r_1, r_2, \ldots, r_x$ with the following properties: (1) $r = r_1; r_2; \ldots; r_x$, (2) for all $0 < i < x$, r_i is a full run of $\mathcal{P}^d(S_1, E, j)$, and (3) r_x is a run of $\mathcal{P}^d(S_1, E, j)$.

  By the induction hypothesis, we have $S_1 \sqsubseteq_E^d \mathit{InlineAbs}(S_1)$. Therefore, for all $0 < i < x$, there is a full run r_i' of $\mathcal{P}(\mathit{InlineAbs}(S_1), E, j)$ such that $\mathit{trace}(r_i)$ is subsumed by $\mathit{trace}(r_i')$. Moreover, there is a run r_x' of $\mathcal{P}(\mathit{InlineAbs}(S_1), E, j)$ such that $\mathit{trace}(r_x)$ is subsumed by $\mathit{trace}(r_x')$.

  Case 1. At least one of r_i' $(1 \leq i \leq x)$ goes wrong. Let j be the least i that goes wrong. Let $r' = r_1'; \ldots; r_j'$. Then r' is a run of $\mathcal{P}(\mathit{InlineAbs}(S_1)^*, E, j)$ and $\mathit{trace}(r')$ subsumes $\mathit{trace}(r)$.

  Case 2. No run r_i' $(1 \leq i \leq x)$ goes wrong. Let $r' = r_1'; \ldots; r_x'$. Then r' is a run of $\mathcal{P}(\mathit{InlineAbs}(S_1)^*, E, j)$ and $\mathit{trace}(r') = \mathit{trace}(r)$.

  In both case, we get a run r' of $\mathcal{P}(\mathit{InlineAbs}(S_1)^*, E, j)$ such that $\mathit{trace}(r')$ subsumes $\mathit{trace}(r)$. Since $\mathit{InlineAbs}({S_1}^*) = \mathit{InlineAbs}(S_1)^*$, we get that r' is a run of $\mathcal{P}(\mathit{InlineAbs}({S_1}^*), E, j)$.

- $\underline{(S = m()):}$ Since the statement $m()$ calls the procedure m, we have $E \Rightarrow \hat{\mathcal{E}}(m)$. Moreover, $\hat{\mathcal{E}}(m) \Rightarrow \hat{\mathcal{E}}(l)$ whenever l is called by m. Therefore $E \Rightarrow \hat{\mathcal{E}}(l)$ whenever l is called by m. From the induction hypothesis, we get $\mathcal{B}(m) \sqsubseteq_E^{d-1} \mathit{InlineAbs}(\mathcal{B}(m))$. We also have the premise that $\mathit{InlineAbs}(\mathcal{B}(m)) \sqsubseteq_{\hat{\mathcal{E}}(m)} \mathit{HavocLocals}(\hat{\mathcal{A}}(m))$ Since $E \Rightarrow \hat{\mathcal{E}}(m)$, we use Lemma 2 to get $\mathit{InlineAbs}(\mathcal{B}(m)) \sqsubseteq_E \mathit{HavocLocals}(\hat{\mathcal{A}}(m))$. It follows that $\mathcal{B}(m) \sqsubseteq_E^{d-1} \mathit{HavocLocals}(\hat{\mathcal{A}}(m))$. Now, note the following two identities:
  1. $[\![S]\!]^d = \{\mathit{Push}\}; [\![\mathcal{B}(m)]\!]^{d-1}; \{\mathit{Pop}\}$.
  2. $\mathit{InlineAbs}(S) = \mathit{PreserveLocals}(\hat{\mathcal{A}}(m))$.

  Note further that the two programs $\mathcal{P}^{d-1}(\mathit{HavocLocals}(\hat{\mathcal{A}}(m)), E, j)$ and $\mathcal{P}^{d-1}(\mathit{PreserveLocals}(\hat{\mathcal{A}}(m)), E, j)$ have identical sets of traces (for all j),

and *Push* and *Pop* only modify local state. Therefore, we conclude that $S \sqsubseteq_E^d$ *InlineAbs*(S).

$\square$

Restatement of Theorem 1 *For each procedure $m \in Proc$, let its body $\mathcal{B}(m) \subseteq Stmt$, abstraction $\mathcal{A}(m) \subseteq SpecStmt$, environment assumption $\mathcal{E}(m) \subseteq SpecAction$, and invariant $\mathcal{I}(m) \subseteq SpecPredicate$ be given. Let $P = \| l()$ be a parallel program. Suppose for all procedures $m \in Proc$, the statement $InlineAbs(\mathcal{B}(m))$ is simulated by $HavocLocals(\hat{\mathcal{A}}(m))$ with respect to the environment assumption $\hat{\mathcal{E}}(m)$. Then the following are true.*

(1) P is simulated by $Q = \| HavocLocals(\hat{\mathcal{A}}(l))$.
*(2) If $\sigma \in \mathcal{I}(l)$, $HavocLocals(\mathcal{A}(l))$ is simulated by $\texttt{true}^*$ with respect to $\hat{\mathcal{E}}(l)$, and $\sigma \xrightarrow{t_1} \cdots \xrightarrow{t_k} \omega$ is a trace of P, then $\omega \neq$ **wrong** and $\omega \in \mathcal{I}(l)$.*

Proof We consider each part of the theorem in turn.

- *Part 1:* By Lemma 3, we get $l() \sqsubseteq_{\hat{\mathcal{E}}(l)}^d InlineAbs(l())$ for all $d \geq 0$. Therefore $l() \sqsubseteq_{\hat{\mathcal{E}}(l)} InlineAbs(l())$. Since $InlineAbs(l()) = PreserveLocals(\hat{\mathcal{A}}(l))$ we know that $l() \sqsubseteq_{\hat{\mathcal{E}}(l)} PreserveLocals(\hat{\mathcal{A}}(l))$. Additionally, the two programs $\mathcal{P}(HavocLocals(\hat{\mathcal{A}}(l)), \hat{\mathcal{E}}(l), j)$ and $\mathcal{P}(PreserveLocals(\hat{\mathcal{A}}(l)), \hat{\mathcal{E}}(l), j)$ have identical sets of traces, for all j. Therefore, $l() \sqsubseteq_{\hat{\mathcal{E}}(l)} HavocLocals(\hat{\mathcal{A}}(l))$. By Lemma 1, we can conclude that P is simulated by Q.
- *Part 2:* By induction on the length m of a run r of P.
  - *Base case:* For $m = 0$, $\sigma_0 \in \mathcal{I}(l)$, and hence the trivial run r does not end in **wrong**.
  - *Induction step:* Let $m > 0$ and let r be the run

$$(\sigma_0, z_0) \xrightarrow{|t_1, a_1|} (\sigma_1, z_1) \ldots (\sigma_{n-1}, z_{n-1}) \xrightarrow{|t_n, a_n|} (\omega, z)$$

where $\sigma_0 \in \mathcal{I}(l)$. By the induction hypothesis, we have that $\sigma_0, \ldots, \sigma_{n-1} \in \mathcal{I}(l)$. Since $P \sqsubseteq Q$, there is a run r' of Q such that $trace(r')$ subsumes $trace(r)$. Let r' be the run

$$(\sigma_0, z_0') \xrightarrow{|t_1, b_1|} (\sigma_1, z_1') \ldots (\sigma_{m-1}, z_{m-1}') \xrightarrow{|t_m, b_m|} (\omega', z')$$

where for each k, b_k is $\hat{p_k}?\hat{X}_k$ where $\hat{p_k} = p_k \wedge \mathcal{I}(l)$ and $\hat{X}_k = X_k \wedge \mathcal{I}'(l) \wedge (\forall i \in Tid : i \neq \texttt{tid} \Rightarrow \hat{\mathcal{E}}(l))$.

Now since b_k is of the above form, $trace(r')$ is also a trace of the program $\mathcal{P}(\mathcal{A}(l), \hat{\mathcal{E}}(l), t_m)$, as elaborated below:

(1) For each state transition $\sigma_{k-1} \xrightarrow{|t_m, b_k|} \sigma_k$, since $\sigma_{k-1} \in \mathcal{I}(l)$, we conclude that $\sigma_{k-1} \xrightarrow{|t_m, p_k?X_k|} \sigma_k$.

(2) For each state transition $\sigma_{k-1} \xrightarrow{|t, b_k|} \sigma_k$ where $t \neq t_m$, $\hat{\mathcal{E}}(l)(t, \sigma_{k-1}, \sigma_k)$

holds.

Furthermore, since $\mathcal{A}(l)$ is simulated by $\mathtt{true}^*$, we conclude that $trace(r')$ is a trace of $\mathcal{P}(\mathtt{true}^*, \hat{\mathcal{E}}(l), t_m)$, which means that $\omega' \neq \textbf{wrong}$. Therefore $n = m$ and $\omega' = \omega$. From the structure of b_m and the fact that $\sigma_{m-1} \in \mathcal{I}(l)$ and $\omega \neq \textbf{wrong}$, we conclude that $\omega \in \mathcal{I}(l)$.

□