A Modular Checker for Multithreaded Programs

Cormac Flanagan[1], Shaz Qadeer[1], and Sanjit A. Seshia[2*]

[1] Compaq Systems Research Center, Palo Alto, CA
[2] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA

Abstract. Designing multithreaded software systems is prone to errors due to the difficulty of reasoning about multiple interleaved threads of control operating on shared data. Static checking, with the potential to analyze the program's behavior over all execution paths and for all thread interleavings, is a powerful debugging tool. We have built a scalable and expressive static checker called Calvin for multithreaded programs. To handle realistic programs, Calvin performs modular checking of each procedure called by a thread using specifications of other procedures and other threads. The checker leverages off existing sequential program verification techniques based on automatic theorem proving. To evaluate the checker, we have applied it to several real-world programs. Our experience indicates that Calvin has a moderate annotation overhead and can catch defects in multithreaded programs, including synchronization errors and violation of data invariants.

1 Introduction

Mission-critical software systems, such as operating systems and databases, are often multithreaded. Ensuring the reliability of these systems is an important but difficult problem. Design of multithreaded software is particularly prone to errors because of subtle interactions between multiple interleaved threads of control operating on shared data. Static checking can analyze the program's behavior over all execution paths and for all thread interleavings. However, current static checking techniques do not scale to large programs.

A common way to achieve scalability is to use modularity, i.e., to analyze each component of the system separately using a specification of other components. A standard notion of modularity for sequential programs is *procedure-modular* reasoning [17], where a call site of a procedure is analyzed using a precondition/postcondition specification of that procedure. But this style of procedure-modular reasoning does not generalize to multithreaded programs [5, 15]. An orthogonal notion of modularity for multithreaded programs is *thread-modular* reasoning [14], which avoids the need to explicitly consider all possible interleavings of threads. 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. But this style of thread-modular reasoning handles a procedure call by the inherently non-scalable method of inlining the procedure body. Consequently, approaches based purely on any one of procedure-modular or thread-modular reasoning are inadequate for large programs with many procedures and many threads.

* Supported in part by a NDSEG Fellowship.

In this paper, we describe a combination of thread-modular and procedure-modular reasoning for verifying safety properties of multithreaded programs. In our methodology, the specification of each procedure 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 each procedure are parameterized by the thread identifier `tid`.

For each procedure p and for each thread `tid`, there are two proof obligations. 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`. It is sound to prove these obligations by inlining the abstractions rather than the implementations of the called procedures. Moreover, these obligations need to hold only in an environment that behaves according to the environment assumption of p. We reduce the two checks to verifying the correctness of a sequential program and present an algorithm to produce this sequential program. We leverage existing techniques for verifying sequential programs based on verification conditions and automatic theorem proving. Our approach is scalable since each procedure is verified separately using an environment assumption to model other threads and abstractions to model called procedures.

We have implemented our methodology for multithreaded Java [3] programs in a checking tool called Calvin. We have applied Calvin to several multithreaded programs, the largest of which is a 1500 line portion of the web crawler Mercator [13]. Our experience indicates that Calvin has the following useful features:

1. 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. 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 [9]. Yet, it uses the conceptually simple framework of reducing the verification of multithreaded programs to the well-studied problem of verifying sequential programs.
3. 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 concise. In addition, the necessary environment assumption annotations are simple and intuitive for programs using common synchronization idioms, such as mutexes or reader-writer locks.

Related Work. In an earlier paper [9], we presented an implementation of thread-modular reasoning for Java programs. However, a procedure call could be handled only by inlining the procedure body.

Static checkers have been built for detecting data races in multithreaded programs [2, 6, 8, 20]; however, these tools are limited to checking a small subset of the synchronization mechanisms found in systems code. Moreover, these tools cannot verify invariants or check refinement of abstractions.

Recently, a few tools for checking invariants on multithreaded programs have appeared. These tools are based on a combination of abstract interpretation and model checking. The Bandera toolkit [7] uses programmer-supplied data abstractions to translate multithreaded Java programs into the input languages of various model checkers. Yahav [21] describes a method to model check multithreaded Java programs using a 3-valued logic [19] 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. [4] present a technique for model checking a software library with an unspecified number of threads, but this method applies only when all the threads are identical and finite-state.

The compositional principle underlying our technique is assume-guarantee reasoning, of which there are several variants. We refer the reader to our earlier paper [9] for a detailed discussion; here we only discuss the closely related work of Jones [14] and Abadi and Lamport [1]. Abadi and Lamport 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. The proof rule of Jones does allow each thread in a multithreaded program to be verified separately; however the program for each thread does not have any procedure calls. Our work can be viewed as a synthesis of the two approaches, which is necessary to tackle the verification of programs that have a large number of procedures and threads.

2 The parallel language Plato

Verifying properties of multithreaded programs in a large and realistic language such as Java is quite complex. To help structure and modularize this process, our checker first translates the given Java program into a simpler intermediate language. This translation eliminates many of the complexities of the Java programming language and is outlined elsewhere [16]. In this paper, we focus on the subsequent verification of the resulting intermediate program, which we assume is expressed in the idealized language *Plato* (parallel language of atomic operations).

A Plato program P is a parallel composition $S_1 \parallel \cdots \parallel S_n$ of several statements, or *threads*. The program executes by interleaving atomic steps of its various threads. The threads interact through a shared store σ, which maps program variables to values. The sets of variables and values are left intentionally unspecified, as they are mostly orthogonal to our technical presentation. Statements in the Plato language include the empty statement **skip**, atomic operation $\{p\}X$ (described below), sequential composition $S_1; S_2$, the nondeterministic choice construct $S_1 \Box S_2$, which executes either S_1 or S_2, the iteration statement S^*, which executes S some arbitrary number of times, and procedure calls. The set *Proc* contains the procedure names and the mapping $\mathcal{B}$ provides the implementation corresponding to a procedure name. To simplify our presentation, the language does not include procedure arguments or return values.

Plato syntax

$$
\begin{array}{lll}
S \in \mathit{Stmt} ::= & \textbf{skip} & \text{no op} \\
& \mid \{p\}X & \text{atomic op} \\
& \mid S \Box S & \text{choice} \\
& \mid S ; S & \text{composition} \\
& \mid S^* & \text{iteration} \\
& \mid p() & \text{procedure call}
\end{array}
\qquad
\begin{array}{rrcl}
\mathcal{B} \in & \mathit{Defn} & = & \mathit{Proc} \rightarrow \mathit{Stmt} \\
P \in & \mathit{Program} & ::= & S_1 \parallel \cdots \parallel S_n \\
\sigma \in & \mathit{Store} & = & \mathit{Var} \rightarrow \mathit{Value} \\
X, Y \in & \mathit{Action} & \subseteq & \mathit{Store} \times \mathit{Store}
\end{array}
$$

Perhaps the most notable aspect of Plato is that it does not contain constructs for conventional primitive operations such as assignment and assertions. Instead, such primitive operations are combined into a general mechanism called an *atomic operation* $\{p\}X$, where p is a state predicate that should hold in the pre-state of the operation, and X is an *action*, or two-state predicate that describes the transition from the pre-state to the post-state.

To execute the atomic operation $\{p\}X$ from a pre-state σ, if $p(\sigma)$ does not hold, then the execution terminates in a special state **wrong** indicating that an error occurred. Otherwise an arbitrary post-store σ' is chosen that satisfies the constraint $X(\sigma, \sigma')$, and the execution of the program continues with the new store σ'. If no post-store σ' satisfies the constraint $X(\sigma, \sigma')$, then the thread blocks; execution proceeds only on the other threads.

Although an action X is a two-state predicate, it is typically written as a formula in which primed variables refer to their value in the post-store σ', and unprimed variables refer to their value in the pre-store σ. In addition, 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 $\{\mathtt{true}\}X$ to simply the action X. We also allow state predicates and actions to refer to thread identifier `tid`, a non-zero integer parameter that uniquely identifies the currently executing thread.

Using atomic operations, Plato can express many conventional constructs, including assignment, assert, assume, if, and while statements. In addition, atomic operations can also express less common constructs, such as the atomic compare-and-swap instruction `CAS(l,e,n)`, which tests if variable `l` has value `e` and swaps the values of `n` and `l` if the test passes; otherwise their values are unchanged.

Expressing conventional constructs in Plato

$$
\begin{array}{rcl}
\mathtt{x = } e & \stackrel{\text{def}}{=} & \langle \mathtt{x}' = e \rangle_{\mathtt{x}} \\
\mathtt{assert}\ e & \stackrel{\text{def}}{=} & \{e\}\langle \mathtt{true} \rangle \\
\mathtt{assume}\ e & \stackrel{\text{def}}{=} & \langle e \rangle
\end{array}
\qquad
\begin{array}{rcl}
\mathtt{if}\ (e)\ \{\ S\ \} & \stackrel{\text{def}}{=} & (\mathtt{assume}\ e; S) \Box (\mathtt{assume}\ \neg e) \\
\mathtt{while}\ (e)\ \{\ S\ \} & \stackrel{\text{def}}{=} & (\mathtt{assume}\ e; S)^*; (\mathtt{assume}\ \neg e) \\
\mathtt{CAS(l,e,n)} & \stackrel{\text{def}}{=} & \left\langle \begin{array}{l} \wedge\ \mathtt{l} \neq \mathtt{e} \Rightarrow (\mathtt{l}' = \mathtt{l} \wedge \mathtt{n}' = \mathtt{n}) \\ \wedge\ \mathtt{l} = \mathtt{e} \Rightarrow (\mathtt{l}' = \mathtt{n} \wedge \mathtt{n}' = \mathtt{l}) \end{array} \right\rangle_{\mathtt{l,n}}
\end{array}
$$

2.1 Semantics

The execution of a Plato program is defined as an interleaving of the executions of its individual, sequential threads, and is formalized as a transition system. A *sequential state* Φ is either a pair of a store and a statement, or the special state **wrong** (indicating that the execution went wrong by failing an assertion).

$$\Phi \in \mathit{SeqState} ::= (\sigma, S) \mid \mathbf{wrong}$$

In the sequential state (σ, S), the statement S identifies the code remaining to be executed, thus avoiding the need for a program counter. Given the environment $\mathcal{B}$ associating procedure names with their implementations, the semantics of an individual thread i is defined via the transition relation $\rightarrow_i$ on sequential states. We write $\mathcal{B} \vdash (\sigma, S) \rightarrow_i \Phi$ to indicate the execution of the "first instruction" in S from store σ, interpreting any occurrence of `tid` in S as i. This instruction may go wrong, yielding $\Phi = \mathbf{wrong}$, or it may terminate normally, yielding a sequential state $\Phi = (\sigma', S')$ consisting of a (possibly modified) store σ' and a statement S' that remains to be executed.

A *parallel state* Θ is either a pair of a store and a program (representing the threads being executed), or the special state **wrong**.

$$\Theta \in \mathit{ParState} ::= (\sigma, P) \mid \mathbf{wrong}$$

We write $\mathcal{B} \vdash (\sigma, P) \rightarrow_p \Theta$ to indicate the execution of a single sequential step of an arbitrarily chosen thread in P from store σ. If that sequential step terminates normally, then execution continues with the resulting post-state. If the sequential step goes wrong, then so does the entire execution. The details of the transition relations $\rightarrow_i$ and $\rightarrow_p$ are given in our technical note [11].

3 Overview of modular verification

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 start by considering an example that provides an overview and motivation of our modular verification method. The multithreaded program SimpleLock consists of two modules, `Top` and `Mutex`. The module `Top` contains two threads that manipulate a shared integer variable `x` (initially zero) protected by a mutex `m`. The module `Mutex` provides acquire and release operations on that mutex. The mutex variable `m` is either the (non-zero) 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.

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 we believe 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. In particular, the environment assumption $E_{\mathtt{tid}}$ for thread `tid` summarizes the possible effects of interleaved atomic steps of other threads. For SimpleLock, an appropriate environment assumption is:

$$E_{\mathtt{tid}} \stackrel{\text{def}}{=} \begin{array}{l} \wedge\ \mathtt{m} = \mathtt{tid} \Rightarrow \mathtt{m} = \mathtt{m}' \\ \wedge\ \mathtt{m} = \mathtt{tid} \Rightarrow \mathtt{x} = \mathtt{x}' \\ \wedge\ I \Rightarrow I' \end{array}$$

The first two conjuncts states that if thread `tid` holds the lock `m`, then other threads cannot modify either `m` or the protected variable `x`. The final conjunct states that every action preserves the invariant that whenever the lock is not held, `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 the implementations of `acquire` and `release` at their call sites. Suppose that $\mathit{InlineBody}(S)$ inlines the implementation of called procedures in a statement S. Then $\mathit{InlineBody}(\mathcal{B}(\mathtt{t1}))$ enjoys the following technical property:

> "$\mathit{InlineBody}(\mathcal{B}(\mathtt{t1}))$" is simulated by E_2^* from the set of states satisfying $\mathtt{m} = 0 \wedge \mathtt{x} = 0$ with respect to the environment assumption E_1.

The notion of simulation is formalized later in the paper. For now, the stated property intuitively means that, when executed from an initial state where both `x` and `m` are zero, each action of procedure `t1` does not go wrong and satisfies E_2, provided that each interleaved action of the other thread satisfies E_1.

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

3.2 Adding procedure-modular verification

Analyzing a large system is impossible using the simple approach sketched above of inlining procedure implementations at call sites. Instead, our checker performs a procedure-modular analysis that uses procedure specifications to model called procedures. We next tackle the question: what is the appropriate specification for the procedure `acquire` in a multithreaded program?

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

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

This specification records that `m` can be modified by the body of `acquire` and asserts that, when `acquire` terminates, `m` is equal to the current thread identifier and that `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 propose a generalized specification language 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. This code fragment $\mathcal{A}(\texttt{acquire})$ concisely specifies this behavior:

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

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 $\mathit{InlineAbs}(\mathcal{B}(\texttt{t}i))$ is 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 "$\mathit{InlineAbs}(\mathcal{B}(\texttt{t1}))$" is simulated by E_2^* from the set of states satisfying $\texttt{m} = 0 \land \texttt{x} = 0$ with respect to E_1, 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. Consider the execution of a procedure p by the current thread `tid`. We assume p is accompanied by a specification consisting of three parts: (1) an invariant $\mathcal{I}(p) \subseteq \mathit{Store}$ that must be maintained by all threads while executing p, (2) an environment assumption $\mathcal{E}(p) \in \mathit{Action}$, parameterized by `tid`, that models the behavior of threads executing concurrently with `tid`'s execution of p, and

(3) an abstraction $\mathcal{A}(p) \in \mathit{Stmt}$, also parameterized by `tid`, that summarizes the behavior of thread `tid` executing p. The abstraction $\mathcal{A}(p)$ may not contain any procedure calls.

In order for the abstraction $\mathcal{A}(p)$ to be correct, we require that the implementation $\mathcal{B}(p)$ be simulated by $\mathcal{A}(p)$ with respect to the environment assumption $\mathcal{E}(p)$. Informally, this simulation requirement holds if, assuming other threads perform actions consistent with $\mathcal{E}(p)$, 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. A relation $R \subseteq \mathit{Store} \times \mathit{Program} \times \mathit{Program}$ is a simulation relation if, whenever we have $R(\sigma,\ \ S_1 \parallel \ \ldots \ \parallel S_n,\ \ T_1 \parallel \ \ldots \ \parallel T_n)$ then the following conditions hold:

1. if $S_i = \mathbf{skip}$ then $\mathcal{B} \vdash (\sigma, T_i) \rightarrow_i^* (\sigma, \mathbf{skip})$.
2. if $\mathcal{B} \vdash (\sigma, S_i) \rightarrow_i \mathbf{wrong}$ then $\mathcal{B} \vdash (\sigma, T_i) \rightarrow_i^* \mathbf{wrong}$
3. if $\mathcal{B} \vdash (\sigma, S_i) \rightarrow_i (\sigma', S_i')$ holds then there exists a statement T_i' such that $\mathcal{B} \vdash (\sigma, T_i) \rightarrow_i^* (\sigma', T_i')$ holds and $R(\sigma',\ \ S_1 \parallel \ \ldots \ \parallel S_i' \parallel \ \ldots \ \parallel S_n,\ \ T_1 \parallel \ \ldots \ \parallel T_i' \parallel \ \ldots \ \parallel T_n)$.

A program P is *simulated* by a program Q from a set of states Σ if there exists a simulation relation R such that $R(\sigma, P, Q)$ holds for each state $\sigma \in \Sigma$. A statement B is *simulated* by a statement A with respect to an environment assumption E from a set of states Σ, if for all non-zero integers j, we have that the program $(B \parallel E^*)[\mathtt{tid} := j]$ is simulated by $(A \parallel E^*)[\mathtt{tid} := j]$ from Σ.

The implementation $\mathcal{B}(p)$ must also satisfy two other properties. While a thread `tid` executes p, every atomic operation must preserve the invariant $\mathcal{I}(p)$ and satisfy the environment assumption $\mathcal{E}(p)[\mathtt{tid} := j]$ of every thread j other than `tid`. We can check that $\mathcal{B}(p)$ is simulated by $\mathcal{A}(p)$ and also satisfies the aforementioned properties by checking that $\mathcal{B}(p)$ is simulated by a derived abstraction $\hat{\mathcal{A}}(p)$ obtained from $\mathcal{A}(p)$ as follows: for every atomic operation $\{p\}X$ in $\mathcal{A}(p)$, replace X by the action

$$X \wedge (\mathcal{I}(p) \Rightarrow \mathcal{I}'(p)) \wedge \forall j : (j \neq 0 \wedge j \neq \mathtt{tid} \Rightarrow \mathcal{E}(p)[\mathtt{tid} := j]).$$

Moreover, this simulation must hold only in an environment that preserves the invariant $\mathcal{I}(p)$. Therefore, we also define a derived environment assumption

$$\hat{\mathcal{E}}(p) \stackrel{\text{def}}{=} \mathcal{E}(p) \wedge (\mathcal{I}(p) \Rightarrow \mathcal{I}'(p)).$$

In order to check simulation for a procedure p, we first inline the derived abstractions for procedures called from $\mathcal{B}(p)$. We use $\mathit{InlineAbs} : \mathit{Stmt} \rightarrow \mathit{Stmt}$ to denote this abstraction inlining operation. We also require that for any procedure q called from p, the environment assumption of p must ensure the environment assumption of q ($\mathcal{E}(p) \Rightarrow \mathcal{E}(q)$), and the invariant of p must ensure the invariant of q ($\mathcal{I}(p) \Rightarrow \mathcal{I}(q)$). Finally, if the program starts by executing a set of concurrent procedure calls $t_1\texttt{()} \parallel \cdots \parallel t_n\texttt{()}$, then we require that the initial store satisfy the invariant $\mathcal{I}(t_i)$ of every t_i and that $\mathit{InlineAbs}(\mathcal{B}(t_i))$ ensures the environment assumption of the other threads. As formalized in the following theorem, if all these conditions hold, then the multithreaded program will not go wrong. Thus, this theorem formalizes our method for combining procedure-modular and thread-modular verification.

Theorem 1. *Let $P = t_1\texttt{()} \parallel \cdots \parallel t_n\texttt{()}$ be a parallel program. Let Init be the set of initial stores of the program. Suppose the following conditions hold.*

1. *For all procedures $p \in \mathit{Proc}$, the statement $\mathit{InlineAbs}(\mathcal{B}(p))$ is simulated by $\hat{\mathcal{A}}(p)$ from $\mathcal{I}(p)$ with respect to the environment assumption $\hat{\mathcal{E}}(p)$.*
2. *For all procedures $p, q \in \mathit{Proc}$, if p calls q then $\mathcal{E}(p) \Rightarrow \mathcal{E}(q)$ and $\mathcal{I}(p) \Rightarrow \mathcal{I}(q)$.*
3. *Init satisfies the invariant $\mathcal{I}(t_i)$ for all $i \in 1..n$.*
4. *Let G be the action $\forall j \in 1..n : (j \neq \texttt{tid} \Rightarrow \hat{\mathcal{E}}(t_j)[\texttt{tid} := j])$. For all $i \in 1..n$, the statement $\mathit{InlineAbs}(\mathcal{B}(t_i))[\texttt{tid} := i]$ is simulated by $G^*[\texttt{tid} := i]$ from Init with respect to the environment assumption $\hat{\mathcal{E}}(t_i)[\texttt{tid} := i]$.*

Then

1. *the program P is simulated by $\mathcal{A}(t_1) \parallel \ldots \parallel \mathcal{A}(t_n)$ from Init.*
2. *for any store $\sigma \in \mathit{Init}$, we have $\mathcal{B} \vdash (\sigma, P) \not\rightarrow_p^* \textbf{wrong}$ and if $\mathcal{B} \vdash (\sigma, P) \rightarrow_p^* (\sigma', P')$ then σ' satisfies $\mathcal{I}(t_i)$ for all $i \in 1..n$.*

Discharging the proof obligations in this theorem requires a method for checking simulation, which is the topic of the following section.

5 Checking simulation

In this section, we present a method for checking simulation between two statements without procedure calls. We first look at 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 $\{q\}Y$, 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 extend our method to check simulation between an implementation B and an abstraction A from a set of states Σ with respect to an environment assumption E. We assume that the abstraction A consists of n atomic operations ($\{\texttt{true}\}Y_i$ for $i \in 1..n$) interleaved with stuttering steps $\{\texttt{true}\}K$, preceded by an asserted precondition $\{\mathit{pre}\}\langle\texttt{true}\rangle$, and ending with the assumed postcondition $\{\texttt{true}\}\langle \mathit{post}\rangle$:

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

This restriction on A enables efficient simulation checking and has been sufficient for all our case studies. Our method can be generalized to arbitrary abstractions A at the cost of more 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 a *witness* variable pc ranging over $\{1, 2, \ldots, n+1\}$ to B, to indicate the operation in A that will simulate the next operation performed in B. 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:

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

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 $E^*; sim(\{p\}X, W)$. The following program extends $[\![B]\!]_A^E$ with constraints on the initial and final values of pc.

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

This program starts execution from the set of states satisfying the precondition *pre* and the initial predicate Σ 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 nonzero interpretation of `tid`, then we conclude that B is simulated by A from Σ 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. For the sake of simplicity, our checker assumes a sequentially consistent memory model and that reads and writes of primitive Java types are atomic (although neither of these assumptions is strictly consistent with Java's current memory model).

In Java, threads are objects of type `Thread`. Therefore, in our implementation the current thread identifier `tid` refers to the object corresponding to the currently executing thread. The implicit lock associated with each Java object is modeled by including in each object an additional abstract field `holder` of type `Thread`, which is either null or refers to the thread currently holding the lock.

6.1 Checker architecture

The input to Calvin is an annotated Java program. In addition to the usual field and method declarations, a Java class can contain invariants, environment assumptions, procedure abstractions, and assertions to be checked. An invariant and an environment assumption are declared once for each class. The environment assumption $\mathcal{E}(p)$ (invariant $\mathcal{I}(p)$) for a procedure p is the conjunction of the environment assumptions (invariants) in (1) the class containing p, and (2) all those classes whose methods are transitively called by p.

Calvin parses, type checks, and translates the annotated input Java program into an intermediate representation language similar to Plato. Calvin then uses the techniques of this paper, as summarized by Theorem 1, to verify the 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 provided, 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, Calvin translates it into a verification condition [12] and invokes the automatic theorem prover Simplify [18] to check the validity of this verification condition.

If the verification condition is valid, then the procedure implements its specification and the stated invariants and assertions are true. Alternatively, if the verification condition is invalid, then the theorem prover generates a counterexample, which is then post-processed into an appropriate error message in terms of the original Java program. The error message may identify an atomic step that violates one of the stated invariants, environment assumptions, or abstraction steps. The error message may also identify an assertion that could go wrong. This assertion may be explicit, as in the `SimpleLock` program, or implicit, for example, that a dereferenced pointer is never null.

The implementation of Calvin leverages extensively off the Extended Static Checker for Java [10], a powerful checking tool for sequential Java programs.

6.2 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 addition to traditional sequential optimization techniques, we have found the following two additional 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 Mercator web crawler

Mercator [13] 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: Java monitors and *readers-writer* locks.

We focused our analysis efforts on the part of Mercator's code (~1500 LOC) that uses readers-writer locks. We first provided a specification of the readers-writer lock implementation 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. The procedure `beginWrite` acquires the lock in write mode by manipulating a concrete boolean variable `hasWriter`. The annotations specifying the abstraction of `beginWrite` and the corresponding Plato code are shown below.

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

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

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 `writable_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.

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

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.

7.2 The java.util.Vector library

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

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

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]))
        ....
}
```

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`. Before the first thread calls `lastIndexOf(Object,int)`, the other thread removes all elements from `v.elementData` and resets it to an array of length 0, and sets `v.elementCount` to 0. Now the first thread tries to access `v.elementData` based on the old value of `v.elementCount` and triggers 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 [8], 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.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM TOPLAS*, 17(3):507–534, 1995.
2. A. Aiken and D. Gay. Barrier inference. In *Proc. 25th POPL*, pages 243–354, 1998.
3. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
4. T. Ball, S. Chaki, and S. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS*, pages 158–173, 2001.
5. A. Birrell, J. Guttag, J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: A formal specification. In *Proc. 11th SOSP*, pages 94–102, 1987.
6. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proc. OOPSLA*, pages 56–69, 2001.
7. 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 *Proc. 23rd ICSE*, pages 177–187, 2001.
8. C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proc. PLDI*, pages 219–232, 2000.
9. C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proc. 11th ESOP*, pages 262–277, 2002.
10. C. Flanagan, K. R. M. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, 2002.
11. C. Flanagan, S. Qadeer, and S. A. Seshia. A modular checker for multithreaded programs. Technical Note 2002-001, Compaq Systems Research Center, 2002.
12. C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proc. 28th POPL*, pages 193–205, 2001.
13. A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. In *Proc. 8th WWW Conf.*, pages 219–229, December 1999.
14. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.
15. L. Lamport. Specifying concurrent program modules. *ACM TOPLAS*, 5(2):190–222, 1983.
16. K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. Technical Note 1999-002, Compaq Systems Research Center, 1999.
17. B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.
18. C. G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, 1981.
19. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. 26th POPL*, pages 105–118, 1999.
20. N. Sterling. WARLOCK — a static data race analysis tool. In *USENIX Tech. Conf. Proc.*, pages 97–106, Winter 1993.
21. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. 28th POPL*, pages 27–40, 2001.