SRC Technical Note
2001 - 003

November 19, 2001

Thread-Modular Verification For Shared-Memory Programs

Cormac Flanagan   Stephen N. Freund   Shaz Qadeer

COMPAQ

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.compaq.com/SRC/

Abstract

Ensuring the reliability of multithreaded software systems is difficult due to the interaction between threads. This paper describes the design and implementation of a static checker for such systems. To avoid considering all possible thread interleavings, the checker uses assume-guarantee reasoning, and relies on the programmer to specify an *environment assumption* that constrains the interaction between threads. Using this environment assumption, the checker reduces the verification of the original multithreaded program to the verification of several sequential programs, one for each thread. These sequential programs are subsequently analyzed using extended static checking techniques (based on verification conditions and automatic theorem proving). Experience indicates that the checker is capable of handling a range of synchronization disciplines. In addition, the required environment assumptions are simple and intuitive for common synchronization idioms.

1 Introduction

Ensuring the reliability of critical software systems is an important but extremely difficult task. A number of useful tools and techniques have been developed for reasoning about sequential systems. Unfortunately, these sequential analysis tools are not applicable to many critical software systems because such systems are often multithreaded. The presence of multiple threads significantly complicates the analysis because of the potential for interference between threads; each atomic step of a thread can influence the subsequent behavior of other threads.

For multithreaded programs, more complex analysis techniques are necessary. The classical assertional approach [Ash75, OG76, Lam77, Lam88] requires control predicates at each program point to specify the reachable program states, but the annotation burden for using this approach is high. Some promising tools [CDH$^+$00, Yah01] use model checking and abstract interpretation to infer the reachable state set automatically, but the need to consider all possible thread interleavings may hinder scaling these tools to large programs.

A more modular and scalable approach is assume-guarantee reasoning, in which each component is verified separately using a specification of the other components [MC81, Jon83a]. Several researchers have presented assume-guarantee proof rules (see Section 2), and some verification tools that support assume-guarantee reasoning on hardware have recently appeared [McM97, AHM$^+$98]. However, tools for assume-guarantee reasoning on realistic software systems do not exist.

In this paper, we describe the design and implementation of a static checker for multithreaded programs, based on an assume-guarantee decomposition. This checker is targeted to the verification of actual implementations of software systems, as opposed to logical models or abstractions of these systems. The checker relies on the programmer to specify, for each thread, an *environment assumption* that models the interference caused by other threads. This environment assumption is an action, or two-store relation, that constrains the updates to the shared store by interleaved atomic steps of other threads. The atomic steps of each thread are also required to satisfy a corresponding guarantee condition that implies the assumption of every other thread.

Using these assumptions and guarantees, our checker translates each thread into a sequential program that models the behavior of that thread precisely and uses the environment assumption to model the behavior of other threads. Thus, our assume-guarantee decomposition reduces the verification of a program with n threads to the verification of n sequential programs. This *thread-modular* decomposition allows our tool to leverage extended static checking techniques [DLNS98] (based on verification conditions and automatic theorem proving) to check the resulting sequential programs.

We have implemented our checker for multithreaded programs written in the Java programming language [AG96], and we have successfully applied this checker to a number of programs. These programs use a variety of synchronization mechanisms, ranging from simple mutual exclusion locks to more complex idioms found in systems code, including a subtle synchronization idiom used in the distributed file system Frangipani [TML97].

Experience with this implementation indicates that our analysis has the following useful features:

1. It naturally scales to programs with many threads since each thread is analyzed separately.
2. For programs using common synchronization idioms, such as mutexes or reader-writer locks, the necessary annotations are simple and intuitive.
3. Control predicates can be expressed in our analysis by explicating the program counter of each thread as an auxiliary variable. Therefore, theoretically our method is as expressive as the Owicki-Gries method. However, for many common cases, such as those appearing in Section 6, our method requires significantly fewer annotations.

The remainder of the paper proceeds as follows. The following section describes related work on assume-guarantee reasoning and other tools for detecting synchronization errors. Section 3 introduces *Plato*, an idealized language for parallel programs that we use as the basis for our development. Section 4 provides a formal definition of thread-modular verification. Section 5 applies thread-modular reasoning to the problem of invariant verification. Section 6 describes our implementation and its application to a number of example programs. We conclude in Section 7.

2 Background

One of the earliest assume-guarantee proof rules was developed by Misra and Chandy [MC81] for message-passing systems, and later refined by others (see, for example, [Jon89, MM93]). However, their message-passing formulation is not directly applicable to shared-memory software.

Jones [Jon83a, Jon83b] gave a proof rule for multithreaded shared-memory programs and used it to manually refine an assume-guarantee specification down to a program. We extend his work to allow the proof obligations for each thread to be checked mechanically by an automatic theorem prover. Stark [Sta85] 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.

Abadi and Lamport [AL95] view the composition of components as a conjunction of temporal logic formulas [Lam94] describing them, and they present a rule to decompose such systems. Since threads modifying shared variables cannot be viewed as components in their framework, their work is not directly applicable to our problem. Collette and Knapp [CK95] extended the rule of Abadi and Lamport to the more operational setting of Unity [CM88] specifications.

Alur and Henzinger [AH96] and McMillan [McM97] present assume-guarantee proof rules for hardware components. A number of other compositional proof rules not based on assume-guarantee reasoning have also been proposed, such as [BKP84, CM88, MP95].

Yahav [Yah01] describes a method to model check multithreaded programs using a 3-valued logic [SRW99, LAS00] to abstract the store. This technique can verify interesting properties of small programs. Păsăreanu et al. [PDH99] also describe a model checking tool for compositional checking of finite-state message-passing systems. Ábráham-Mumm and deBoer [AMdB00] sketch a logic for verifying multi-threaded Java programs indirectly via a translation to communicating sequential programs.

A number of tools have been developed for identifying specific synchronization errors in multithreaded programs. These approaches are less general than thread-modular verification and use specific analysis techniques to locate specific errors, such as data races and deadlocks. For example, RCC/Java [FF00] is an annotation-based checker for Java that uses a type system to identify data races [FA99]. While this tool is successful at finding errors in large programs, the inability to specify subtle synchronization patterns results in many false alarms [FF01]. ESC/Java [LSS99], Warlock [Ste93], and the dynamic testing tool Eraser [SBN+97] are other tools in this category, and are discussed in an earlier paper [FF00].

3 The Parallel Language Plato

We present thread-modular verification in terms of the idealized language *Plato* (parallel language of atomic operations). A Plato program P is a parallel composition $S_1 \mid \cdots \mid S_n$ 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 development.

Statements in the Plato language include the empty statement **skip**, sequential composition $S_1; S_2$, the nondeterministic choice construct $S_1 \Box S_2$, which executes either S_1 or S_2, and the iteration statement S^*, which executes S some arbitrary number of times.

Plato syntax

S	$\in$	*Stmt*	::=	**skip**	no operation	P	$\in$	*Program*	::=	$S_1 \mid \cdots \mid S_n$
			|	$X \downarrow Y$	atomic operation	σ	$\in$	*Store*	$=$	*Var* $\to$ *Value*
			|	$S \Box S$	nondeterministic choice	X, Y	$\in$	*Action*	$\subseteq$	*Store* $\times$ *Store*
			|	$S; S$	composition					
			|	S^*	nondeterministic iteration					

Perhaps the most notable aspect of Plato is that it does not contain constructs for conventional primitive operations such as assignment and lock acquire and release operations. Instead, such primitive operations are combined into a general mechanism called an *atomic operation* $X \downarrow Y$, where X and Y are *actions*, or two-store predicates. The action X is a constraint on the transition from the pre-store σ to the post-store σ', and Y is an assertion about this transition.

To execute the atomic operation $X \downarrow Y$, an arbitrary post-store σ' is chosen that satisfies the constraint $X(\sigma, \sigma')$. There are two possible outcomes:

1. If the assertion $Y(\sigma, \sigma')$ holds, then the atomic operation terminates normally, and the execution of the program continues with the new store σ'.

2. If the assertion $Y(\sigma, \sigma')$ does not hold, then the execution *goes wrong*.

If no post-store σ' satisfies the constraint $X(\sigma, \sigma')$, then the thread is blocked, and the execution can proceed only on the other threads.

In an atomic operation, we write each action 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 $V \subseteq Var$, 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. We abbreviate the common case $\langle X \rangle_\emptyset$ to $\langle X \rangle$ and also abbreviate $\langle X \rangle_{\{a\}}$ to $\langle X \rangle_a$.

Atomic operations can express many conventional primitives, such as assignment, assert, and assume statements (see below). Atomic operations can also express other primitives, in particular lock acquire and release operations. We assume that each lock is represented by a variable and that each thread has a unique nonzero thread identifier. If a thread holds a lock, then the lock variable contains the corresponding thread identifier; if the lock is not held, then the variable contains zero. Under this representation, acquire and release operations for lock `mx` and thread i are shown below. Finally, Plato can also express traditional control constructs, such as `if` and `while` statements.

Expressing conventional constructs in Plato

$$
\begin{array}{rcl@{\qquad}rcl}
\texttt{x := }e & \stackrel{\text{def}}{=} & \langle \texttt{x' = }e\rangle_{\texttt{x}} \downarrow \texttt{true} & \texttt{acq(mx)} & \stackrel{\text{def}}{=} & \langle \texttt{mx = 0} \wedge \texttt{mx' = i}\rangle_{\texttt{mx}} \downarrow \texttt{true} \\
\texttt{assert }e & \stackrel{\text{def}}{=} & \langle \texttt{true}\rangle \downarrow e & \texttt{rel(mx)} & \stackrel{\text{def}}{=} & \langle \texttt{mx' = 0}\rangle_{\texttt{mx}} \downarrow (\texttt{mx = i}) \\
\texttt{assume }e & \stackrel{\text{def}}{=} & \langle e\rangle \downarrow \texttt{true} & \texttt{if }(e)\ \{\ S\ \} & \stackrel{\text{def}}{=} & (\texttt{assume }e; S)\Box(\texttt{assume }\neg e) \\
 & & & \texttt{while }(e)\ \{\ S\ \} & \stackrel{\text{def}}{=} & (\texttt{assume }e; S)^*; (\texttt{assume }\neg e)
\end{array}
$$

3.1 Formal Semantics

The execution of a program is defined as an interleaving of the executions of its individual, sequential threads. 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). The semantics of individual threads is defined via the transition relation $\Phi \rightarrow_s \Phi$, defined in the figure below.

A *parallel state* Θ is either a pair of a store and a program (representing the threads being executed), or the special state **wrong**. The transition relation $\Theta \rightarrow_p \Theta$ on parallel states executes a single sequential step of an arbitrarily chosen thread. 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.

Formal semantics of Plato

$$
\Phi \in \mathit{SeqState} ::= (\sigma, S) \mid \textbf{wrong} \qquad\qquad \Theta \in \mathit{ParState} ::= (\sigma, P) \mid \textbf{wrong}
$$

[ACTION OK]
$$\frac{X(\sigma,\sigma') \qquad Y(\sigma,\sigma')}{(\sigma, X \downarrow Y) \rightarrow_s (\sigma', \textbf{skip})}$$

[ACTION WRONG]
$$\frac{X(\sigma,\sigma') \qquad \neg Y(\sigma,\sigma')}{(\sigma, X \downarrow Y) \rightarrow_s \textbf{wrong}}$$

[CHOICE]
$$\frac{i \in \{1,2\}}{(\sigma, S_1 \Box S_2) \rightarrow_s (\sigma, S_i)}$$

[LOOP DONE]
$$\frac{}{(\sigma, S^*) \rightarrow_s (\sigma, \textbf{skip})}$$

[LOOP UNROLL]
$$\frac{}{(\sigma, S^*) \rightarrow_s (\sigma, S; S^*)}$$

[ASSOC]
$$\frac{}{(\sigma, (S_1; S_2); S_3) \rightarrow_s (\sigma, S_1; (S_2; S_3))}$$

[SEQ STEP]
$$\frac{(\sigma, S_1) \rightarrow_s (\sigma', S_1')}{(\sigma, S_1; S_2) \rightarrow_s (\sigma', S_1'; S_2)}$$

[SEQ SKIP]
$$\frac{}{(\sigma, \textbf{skip}; S) \rightarrow_s (\sigma, S)}$$

[SEQ WRONG]
$$\frac{(\sigma, S_1) \rightarrow_s \textbf{wrong}}{(\sigma, S_1; S_2) \rightarrow_s \textbf{wrong}}$$

[PARALLEL]
$$\frac{(\sigma, S_i) \rightarrow_s (\sigma', S_i')}{\begin{array}{l}(\sigma, S_1 \mid \cdots \mid S_i \mid \cdots \mid S_n) \\ \quad \rightarrow_p (\sigma', S_1 \mid \cdots \mid S_i' \mid \cdots \mid S_n)\end{array}}$$

[PARALLEL WRONG]
$$\frac{(\sigma, S_i) \rightarrow_s \textbf{wrong}}{(\sigma, S_1 \mid \cdots \mid S_i \mid \cdots \mid S_n) \rightarrow_p \textbf{wrong}}$$

4 Thread-Modular Verification

We reason about a parallel program $P = S_1 \mid \cdots \mid S_n$ by reasoning about each thread in P separately. For each thread i, we specify two actions — an environment assumption A_i and a guarantee G_i. The assumption of a thread is a specification of what transitions may be performed by other threads in the program. The guarantee of a thread is required to hold on every action performed by the thread itself. To ensure the correctness of the assumptions, we require that the guarantee of each thread be stronger than the assumption of every other thread. In addition, to accommodate effect-free transitions, we require each assumption and guarantee to be reflexive. The precise statement of these requirements is as follows:

1. A_i and G_i are reflexive for all $i \in 1..n$.
2. $G_i \subseteq A_j$ for all $i, j \in 1..n$ such that $i \neq j$.

If these requirements are satisfied, then $\langle A_1, G_1\rangle, \ldots, \langle A_n, G_n\rangle$ is an *assume-guarantee decomposition* for P.

We next define the translation $[\![S]\!]_G^A$ of a statement S with respect to an assumption A and a guarantee G. This translation verifies that each atomic operation of S satisfies the guarantee G. In addition, the translation inserts the iterated environment assumption A^* as appropriate to model atomic steps of other threads.

$$[\![\bullet]\!]_\bullet^\bullet : \mathit{Stmt} \times \mathit{Action} \times \mathit{Action} \to \mathit{Stmt}$$

$$\begin{array}{rcl}
[\![\mathbf{skip}]\!]_G^A & = & A^* \\
[\![X \downarrow Y]\!]_G^A & = & A^*; X \downarrow (Y \wedge G); A^* \\
[\![S_1 \Box S_2]\!]_G^A & = & A^*; ([\![S_1]\!]_G^A \Box [\![S_2]\!]_G^A) \\
[\![S_1; S_2]\!]_G^A & = & [\![S_1]\!]_G^A; [\![S_2]\!]_G^A \\
[\![S^*]\!]_G^A & = & A^*; (([\![S]\!]_G^A; A^*)^*; A^*)
\end{array}$$

We use this translation and the assume-guarantee decomposition to abstract each thread i of the parallel program P into the sequential program $[\![S_i]\!]_{G_i}^{A_i}$, called the *i-abstraction* of P. For any thread i, if A_i models the environment of thread i and the sequential i-abstraction of P does not go wrong, then we conclude that the corresponding thread S_i in P does not go wrong and also satisfies the guarantee G_i. Thus, if none of the i-abstractions go wrong, then none of the threads in P go wrong. This property is formalized by the following theorem; its correctness proof avoids circular reasoning by using induction over time. (An extended report containing the proof of theorems in this paper is in available at `http://www.research.compaq.com/SRC/personal/freund/tmv-draft.ps`.)

Theorem 1 (Thread-Modular Verification) *Let $P = S_1 \mid \cdots \mid S_n$ be a parallel program with assume-guarantee decomposition $\langle A_1, G_1\rangle, \ldots, \langle A_n, G_n\rangle$. For all $\sigma \in Store$, if $\forall i \in 1..n.\ (\sigma, [\![S_i]\!]_{G_i}^{A_i}) \not\to_s^*$ **wrong**, then $(\sigma, P) \not\to_p^*$ **wrong**.*

This theorem allows us to decompose the analysis of a parallel program $S_1 \mid \cdots \mid S_n$ into analyses of individual threads by providing an assume-guarantee decomposition $\langle A_1, G_1\rangle, \ldots, \langle A_n, G_n\rangle$. In practice, we only require the programmer to specify reflexive assumptions $A_1, \ldots, A_n$, and we derive the corresponding reflexive guarantees by

$$G_i \quad = \quad (\forall j \in 1..n.\ j \neq i \Rightarrow A_j).$$

For all examples we have considered, the natural assumptions are transitive in addition to being reflexive. This allows us to optimize the iterations A_i^* in each i-abstraction to simply the action A_i. In addition, the n environment assumptions $A_1, \ldots, A_n$ for a program with n threads can typically be conveniently expressed as a single action parameterized by thread identifier, as shown below.

4.1 Example

To illustrate Theorem 1, consider the following program SimpleLock. The program manipulates two shared variables, an integer `x` and a lock `mx`. To synchronize accesses to `x`, each thread acquires the lock `mx` before manipulating `x`. The correctness condition we would like to verify is that $\texttt{Thread}_1$ never goes wrong by failing the assertion `x > 1`.

SimpleLock program, desugared $\texttt{Thread}_1$, and $[\![\texttt{Thread}_1]\!]_{G_1}^{A_1}$

$\texttt{Thread}_1$:	$\texttt{Thread}_2$:	Desugared $\texttt{Thread}_1$:	$[\![\texttt{Thread}_1]\!]_{G_1}^{A_1}$:
`acq(mx);`	`acq(mx);`	$\langle \texttt{mx = 0} \wedge \texttt{mx' = 1} \rangle_{\texttt{mx}}$;	A_1 ; $\langle \texttt{mx = 0} \wedge \texttt{mx' = 1} \rangle_{\texttt{mx}} \downarrow G_1$;
`x := x * x;`	`x := 0;`	$\langle \texttt{x' = x * x} \rangle_{\texttt{x}}$;	A_1 ; $\langle \texttt{x' = x * x} \rangle_{\texttt{x}} \downarrow G_1$;
`x := x + 2;`	`rel(mx);`	$\langle \texttt{x' = x + 2} \rangle_{\texttt{x}}$;	A_1 ; $\langle \texttt{x' = x + 2} \rangle_{\texttt{x}} \downarrow G_1$;
`assert x > 1;`		$\langle \texttt{true} \rangle \downarrow (\texttt{x > 1})$;	A_1 ; $\langle \texttt{true} \rangle \downarrow (\texttt{x > 1} \wedge G_1)$;
`rel(mx);`		$\langle \texttt{mx' = 0} \rangle_{\texttt{mx}} \downarrow (\texttt{mx = 1})$;	A_1 ; $\langle \texttt{mx' = 0} \rangle_{\texttt{mx}} \downarrow (\texttt{mx = 1} \wedge G_1)$;
			A_1

The synchronization discipline in this program is that if a thread holds the lock `mx`, then the other thread cannot modify either the variable `x` or the lock variable `mx`. This discipline is formalized by the following environment assumption for thread identifier $i \in 1..2$:

$$A_i \quad = \quad (\texttt{mx} = i \Rightarrow \texttt{mx}' = i \wedge \texttt{x}' = \texttt{x})$$

The corresponding guarantees are $G_1 = A_2$ and $G_2 = A_1$. Since A_1 is reflexive and transitive, we can optimize both A_1^* and $A_1^* ; A_1^*$ to A_1 in the 1-abstraction of SimpleLock, shown above.

Verifying the two i-abstractions of SimpleLock is straightforward, using existing analysis techniques for sequential programs. In particular, our checker uses extended static checking to verify that the two sequential i-abstractions of SimpleLock do not go wrong. Thus, the hypotheses of Theorem 1 are satisfied, and we conclude that the parallel program SimpleLock does not fail its assertion.

5 Invariant Verification

In the previous section, we showed that the SimpleLock program does not fail its assertion. In many cases, we would also like to show that a program preserves certain data invariants. This section extends thread-modular verification to check data invariants on a parallel program $P = S_1 \mid \ldots \mid S_n$. We use $\mathit{Init} \subseteq \mathit{Store}$ to describe the possible initial states of P, and we say that a set of states I is an *invariant* of P with respect to Init if for each $\sigma \in \mathit{Init}$, if $(\sigma, P) \rightarrow_p^* (\sigma', P')$, then $\sigma' \in I$.

To show that I is an invariant of P, it suffices to show that I holds initially (i.e., $\mathit{Init} \subseteq I$), and that I is preserved by each transition of P. We prove the latter property using thread-modular verification, where the guarantee G_i of each thread satisfies the property

$$G_i \quad \Rightarrow \quad (I \Rightarrow I').$$

In this formula, the predicate I denotes the action where I holds in the pre-state, and the post-state is unconstrained; similarly, I' denotes the action where the pre-state is unconstrained, and I holds in the post-state. Thus, $I \Rightarrow I'$ is the action stating that I is preserved.

The following theorem formalizes the application of thread-modular reasoning to invariant verification.

Theorem 2 (Invariant Verification) *Let $P = S_1 \mid \cdots \mid S_n$ be a parallel program with assume-guarantee decomposition $\langle A_1, G_1\rangle, \ldots, \langle A_n, G_n\rangle$, and let Init and I be sets of stores. Suppose:*

1. *$\mathit{Init} \subseteq I$*
2. *$\forall i \in 1..n.\ G_i \Rightarrow (I \Rightarrow I')$*
3. *$\forall i \in 1..n.\ \forall \sigma \in \mathit{Init}.\ (\sigma, [\![S_i]\!]^{A_i}_{G_i}) \not\rightarrow^*_s$* **wrong**

Then I is an invariant of P with respect to Init.

In practice, we apply this theorem by requiring the programmer to supply the invariant I and the parameterized environment assumption A_i. We derive the corresponding parameterized guarantee:

$$G_i \quad = \quad (\forall j \in 1..n.\ j \neq i \Rightarrow A_j) \wedge (I \Rightarrow I')$$

The guarantee states that each atomic step of a thread satisfies the assumptions of the other threads and also preserves the invariant. Since each step preserves the invariant, we can strengthen the environment assumption to:

$$B_i \quad = \quad A_i \wedge (I \Rightarrow I')$$

The resulting assume-guarantee decomposition $\langle B_1, G_1\rangle, \ldots, \langle B_n, G_n\rangle$ is then used in the application of Theorem 2. The first condition of that theorem, that $\mathit{Init} \subseteq I$, can be checked using a theorem prover [Nel81]. The second condition, that $\forall i \in 1..n.\ G_i \Rightarrow (I \Rightarrow I')$, follows directly from the definition of G_i. The final condition (similar to the condition of Theorem 1), that each sequential i-abstraction $[\![S_i]\!]^{B_i}_{G_i}$ does not go wrong from any initial store in *Init*, can be verified using extended static checking. The following section describes our implementation of a checker for parallel programs that supports thread modular and invariant verification.

6 Implementation and Applications

We have implemented an automatic checker for parallel, shared-memory programs. This checker takes as input a Java program, together with annotations describing appropriate environment assumptions, invariants, and asserted correctness properties. The input program is first translated into an intermediate representation language similar to Plato, and then the techniques of this paper are applied to generate an i-abstraction, which is parameterized by the thread identifier i. The checker optimizes this i-abstraction using a number of techniques. In particular, the checker uses an automatic theorem prover to verify that the environment assumption A_i is reflexive and transitive and then optimizes each iterated environment assumption A_i^* to A_i.

Once optimized, the i-abstraction is then converted into a verification condition [Dij75, FS01]. When generating this verification condition, procedure calls are handled by inlining, and loops are translated either using a programmer-supplied loop invariant, or in an unsound, but useful, manner by unrolling loops some finite number of times [LSS99]. The automatic theorem prover Simplify [Nel81], is invoked to check the validity of this verification condition.

If the verification condition is valid, then the parameterized i-abstraction does not go wrong, and hence the original Java program preserves the stated invariants and assertions. 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. Typically, the error message either identifies an atomic step that may violate one of the stated invariants or environment assumptions, or identifies an assertion that may go wrong. This assertion may either be explicit, as in the example programs, or may be an implicit assertion, for example, that a dereferenced pointer is never null.

The implementation of our checker 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 related documents [DLNS98, LSS99, Ext01].

In the next three subsections, we describe the application of our checker to parallel programs using various kinds of synchronization. Due to space restrictions, these examples are necessarily small, but our checker has also been applied to significantly larger programs. In each of the presented examples, we state the necessary annotations: the assumptions A_i for each thread i and the invariant I to be proved. Given these annotations, our checker can automatically verify each of the example programs. For consistency with our earlier development, these programs are presented using Plato syntax.

6.1 Dekker's Mutual Exclusion Algorithm

Our first example is Dekker's algorithm, a classic algorithm for mutual exclusion that uses subtle synchronization.

Dekker's mutual exclusion algorithm

```
Variables:            Thread1:                        Thread2:
  boolean a1;           while (true) {                  while (true) {
  boolean a2;             a1 := true;                     a2 := true;
  boolean cs1;            cs1 := ¬a2;                     cs2 := ¬a1;
  boolean cs2;            if (cs1) {                      if (cs2) {
                            // critical section              // critical section
Initially:                  cs1 := false;                   cs2 := false;
  ¬cs1 ∧ ¬cs2             }                               }
                          a1 := false;                    a2 := false;
                        }                               }
```

The algorithm uses two boolean variables $\mathtt{a}_1$ and $\mathtt{a}_2$. We introduce two variables $\mathtt{cs}_1$ and $\mathtt{cs}_2$, where $\mathtt{cs}_i$ is true if thread i is in its critical section. Each $\mathtt{Thread}_i$ expects that the other thread will not modify $\mathtt{a}_i$ and $\mathtt{cs}_i$. We formalize this expectation as the assumption:

$$A_i \quad = \quad (\mathtt{a}_i = \mathtt{a}'_i \wedge \mathtt{cs}_i = \mathtt{cs}'_i)$$

We would like to verify that the algorithm achieves mutual exclusion, which is expressed as the invariant $\neg(\mathtt{cs}_1 \wedge \mathtt{cs}_2)$. Unfortunately, this invariant cannot be verified directly. The final step is to strengthen the invariant to

$$I \quad = \quad \neg(\mathtt{cs}_1 \wedge \mathtt{cs}_2) \wedge (\mathtt{cs}_1 \Rightarrow \mathtt{a}_1) \wedge (\mathtt{cs}_2 \Rightarrow \mathtt{a}_2).$$

Using the assumptions A_1 and A_2 and the strengthened invariant I, our checker verifies that Dekker's algorithm achieves mutual exclusion.

In this example, the environment assumptions are quite simple. The subtlety of the algorithm is reflected in the invariant which had to be strengthened by two conjuncts. In general, the complexity of the assertions needed by our checker reflects the complexity of the synchronization patterns used in program being checked.

6.2 Reader-Writer Locks

The next example applies thread-modular reasoning to a *reader-writer lock*, which can be held in two different modes, *read mode* and *write mode*. Read mode is non-exclusive, and multiple threads may hold the lock in that mode. On the other hand, holding the lock in write mode means that no other threads hold the lock in either mode. Acquire operations block when these guarantees cannot be satisfied.

We implement a reader-writer lock using two variables: an integer `w`, which identifies the thread holding the lock in write mode (or 0 if no such thread exists), and an integer set `r`, which contains the identifiers of all threads holding the lock in read mode. The following atomic operations express acquire and release in read and write mode for thread i:

$$\begin{array}{rcl}
\texttt{acq_write}(\texttt{w},\texttt{r}) & \stackrel{\text{def}}{=} & \langle \texttt{w = 0} \wedge \texttt{r} = \emptyset \wedge \texttt{w}' = i\rangle_{\texttt{w}} \\
\texttt{acq_read}(\texttt{w},\texttt{r}) & \stackrel{\text{def}}{=} & \langle \texttt{w = 0} \wedge \texttt{r}' = \texttt{r} \cup \{i\}\rangle_{\texttt{r}} \\
\texttt{rel_write}(\texttt{w},\texttt{r}) & \stackrel{\text{def}}{=} & \langle \texttt{w}' = \texttt{0}\rangle_{\texttt{w}} \downarrow (\texttt{w} = i) \\
\texttt{rel_read}(\texttt{w},\texttt{r}) & \stackrel{\text{def}}{=} & \langle \texttt{r}' = \texttt{r} \setminus \{i\}\rangle_{\texttt{r}} \downarrow (i \in \texttt{r})
\end{array}$$

For a thread to acquire the lock in write mode, there must be no writer and no readers. Similarly, to acquire the lock in read mode, there must be no writer, but there may be other readers, and the result of the acquire operation is to put the thread identifier into the set `r`. The release operations are straightforward. All of these lock operations respect the following data invariant *RWI* and the environment assumption RWA_i:

$$\begin{array}{rcl}
RWI & = & (\texttt{r} = \emptyset \vee \texttt{w} = \texttt{0}) \\
RWA_i & = & (\texttt{w} = i \Leftrightarrow \texttt{w}' = i) \;\wedge\; (i \in \texttt{r} \Leftrightarrow i \in \texttt{r}')
\end{array}$$

We illustrate the analysis of reader-writer locks by verifying the following program, in which the variable `x` is guarded by the reader-writer lock. $\texttt{Thread}_2$ asserts that the value of `x` is stable while the lock is held in read mode, even though $\texttt{Thread}_1$ mutates `x` while the lock is held in write mode.

Reader-writer lock example

Variables:	$\texttt{Thread}_1$:	$\texttt{Thread}_2$:
`int w, x, y;`	`acq_write(w, r);`	`acq_read(w, r);`
`int_set r;`	`x := 3;`	`y := x;`
	`rel_write(w, r);`	`assert y = x;`
Initially:		`rel_read(w, r);`
$\texttt{w = 0} \wedge \texttt{r} = \emptyset$;		

The appropriate environment assumption for this program

$$A_i \quad = \quad RWA_i \wedge (i \in \texttt{r} \Rightarrow \texttt{x} = \texttt{x}') \wedge (i = 2 \Rightarrow \texttt{y} = \texttt{y}')$$

states that (1) each thread i can assume the reader-writer assumption RWA_i, (2) if thread i holds the lock in read mode, then `x` cannot be changed by another thread, and (3) the variable `y` is modified only by $\texttt{Thread}_2$. This environment assumption, together with the data invariant RWI, is sufficient to verify this program using our checker.

Although the reader-writer lock is more complex than the mutual-exclusion lock described earlier, the additional complexity of the reader-writer lock is localized to the annotations RWA_i and RWI that specify the lock implementation. Given these annotations, it is encouraging to note that the additional annotations required to verify reader-writer lock clients are still straightforward.

6.3 Time-Varying Mutex Synchronization

We now present a more complex example to show the power of our checker. The example is derived from a synchronization idiom found in the Frangipani file system [TML97].

For each file, Frangipani keeps a data structure called an *inode* that contains pointers to disk blocks that hold the file data. Each block has a busy bit indicating whether the block has been allocated to an inode. Since the file system is multithreaded, these data structures are guarded by mutexes. In particular, distinct mutexes protect each inode and each busy bit. However, the mutex protecting a disk block depends on the block's allocation status. If a block is unallocated (its busy bit is false), the mutex for its busy bit protects it. If the block is allocated (its busy bit is true), the mutex for the owning inode protects it. The following figure shows a highly simplified version of this situation.

Time-varying mutex program

```
Variables:               Thread1:                    Thread2:
  int block;               acq(m_inode);               acq(m_busy);
  boolean busy;            if (¬inode) {               if (¬busy) {
  boolean inode;             acq(m_busy);                block := 0;
  int m_inode;               busy := true;               assert block = 0;
  int m_busy;                rel(m_busy);              }
                             inode := true;            rel(m_busy);
Initially:                 }
  inode = busy             block := 1;
                           assert block = 1;
                           rel(m_inode);
```

The program contains a single disk block, represented by the integer variable `block`, and uses a single bit `busy` to store the block's allocation status. There is a single inode whose contents have been abstracted to a bit indicating whether the inode has allocated the block. The two mutexes `m_inode` and `m_busy` protect the variables `inode` and `busy`, respectively.

The program contains two threads. $\texttt{Thread}_1$ acquires the mutex `m_inode`, allocates the block if it is not allocated already, and sets `block` to 1. Since $\texttt{Thread}_1$ is holding the lock on the inode that has allocated the block, the thread has exclusive access to the block contents. Thus, the subsequent assertion that the block value remains 1 should never fail.

$\texttt{Thread}_2$ acquires the mutex `m_busy`. If `busy` is false, the thread sets `block` to 0 and asserts that the value of block is 0. Since $\texttt{Thread}_2$ holds the lock on `busy` when the block is unallocated, the thread should have exclusive access to `block`, and the assertion should never fail.

We now describe annotations necessary to prove that the assertions always hold. First, the lock m_inode protects inode, and the lock m_busy protects busy:

$$
\begin{aligned}
J_i &= (\texttt{m\_inode = } i \Rightarrow (\texttt{m\_inode}' = i \wedge \texttt{inode' = inode})) \wedge \\
&\quad (\texttt{m\_busy = } i \Rightarrow (\texttt{m\_busy}' = i \wedge \texttt{busy' = busy}))
\end{aligned}
$$

In addition, if busy is true, then block is protected by m_inode; otherwise, block is protected by m_busy:

$$
\begin{aligned}
K_i &= (\texttt{busy} \wedge \texttt{m\_inode = } i \Rightarrow \texttt{block = block}') \wedge \\
&\quad (\neg\texttt{busy} \wedge \texttt{m\_busy = } i \Rightarrow \texttt{block = block}')
\end{aligned}
$$

Finally, the busy bit must be set when the inode has allocated the block. Moreover, the busy bit can be reset only by the thread that holds the lock on the inode. We formalize these requirements as the invariant I and the assumption L_i respectively.

$$
\begin{aligned}
I &= (\texttt{m\_inode = 0} \wedge \texttt{inode}) \Rightarrow \texttt{busy} \\
L_i &= (\texttt{m\_inode = } i \wedge \texttt{busy}) \Rightarrow \texttt{busy}'
\end{aligned}
$$

With these definitions, the complete environment assumption for each thread i is:

$$A_i = J_i \wedge K_i \wedge L_i$$

Given A_i and I, our checker is able to verify that the assertions in this program never fail.

This example illustrates the expressiveness of our checker. By comparison, previous tools for detecting synchronization errors [Ste93, SBN+97, FF00] have been mostly limited to finding races in programs that only use simple mutexes (and, in some cases, reader-writer locks). However, operating systems and other large-scale systems tend to use a variety of additional synchronization mechanisms, some of which we have described in the last few sections. Other synchronization idioms include binary and counting semaphores, producer-consumer synchronization, fork-join parallelism, and wait-free non-blocking algorithms. Our experience to date indicates that our checker has the potential to handle many of these synchronization disciplines.

7 Conclusions

The ability to reason about the correctness of large, multithreaded programs is essential to ensure the reliability of such systems. One natural strategy for decomposing such verification problems is procedure-modular verification, which has enjoyed widespread use in a variety of program analysis techniques for many years. Instead of reasoning about a call-site by inlining the corresponding procedure body, procedure-modular verification uses some specification of that procedure, for example, a type signature or a precondition/postcondition pair.

A second, complementary decomposition strategy is assume-guarantee decomposition [Jon83a], which avoids the need to consider all possible interleavings of the various threads explicitly. Instead, each thread is analyzed separately, with an environment assumption providing a specification of the behavior of the other program threads.

This paper presents an automatic checker for multithreaded programs, based on an assume-guarantee decomposition. The checker relies on the programmer to provide annotations describing the environment assumption of each thread. A potential concern with any annotation-based analysis technique is the overhead of providing such annotations. Our experience applying our checker to a number of example programs indicates that this annotation overhead is moderate. In particular, for many common synchronization idioms, the necessary environment assumptions are simple and intuitive. The environment assumption may also function as useful documentation for multithreaded programs, providing benefits similar to (formal or informal) procedure specifications.

We believe that verification of large, multithreaded programs requires the combination of both thread-modular and procedure-modular reasoning. However, specifying a procedure in a multithreaded program is not straightforward. In particular, because other threads can observe intermediate states of the procedure's computation, a procedure cannot be considered to execute atomically and cannot be specified as a simple precondition/postcondition pair. Combining thread-modular and procedure-modular reasoning appropriately is an important area for future work. We hope that this paper provides a suitable foundation for further development in this area.

References

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[AHM+98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 521–525. Springer-Verlag, 1998.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[AMdB00] E. Ábráham Mumm and F. S. de Boer. Proof-outlines for threads in java. In *CONCUR 2000: Theories of Concurrency*, Lecture Notes in Computer Science 1877. Springer-Verlag, 2000.

[Ash75] E.A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, January 1975.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal-logic specifications. In *Proceedings of the 16th Annual Symposium on Theory of Computing*, pages 51–63. ACM Press, 1984.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[CK95] 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.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, 1988.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DLNS98] D. L. Detlefs, K. R. M. Leino, C. G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[Ext01] Extended Static Checking web page, Compaq Systems Research Center. `http://www.research.compaq.com/SRC/esc/`, July 2001.

[FA99] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of European Symposium on Programming*, pages 91–108, March 1999.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[FF01] C. Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 90–96, June 2001.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.

[Jon83a] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[Jon83b] C.B. Jones. Specification and design of (parallel) programs. In R. Mason, editor, *Information Processing*, pages 321–332. Elsevier Science Publishers B. V. (North-Holland), 1983.

[Jon89] B. Jonsson. On decomposing and refining specifications of distributed systems. In J.W. 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.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[Lam88] L. Lamport. Control predicates are better than dummy variables. *ACM Transactions on Programming Languages and Systems*, 10(2):267–281, April 1988.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proceedings of the Static Analysis Symposium*, pages 280–301, 2000.

[LSS99] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[McM97] K.L. 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.

[MM93] A. Mokkedem and D. Mery. On using a composition principle to design parallel programs. In *Algebraic Methodology and Software Technology*, pages 315–324, 1993.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[Nel81] C. G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, 1981.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[PDH99] C.S. Păsăreanu, M.B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *Theoretical and Practical Aspects of SPIN Model Checking*, Lecture Notes in Computer Science 1680, 1999.

[SBN+97] S. Savage, M. Burrows, C.G. Nelson, P. Sobalvarro, and T.A. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[SRW99] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Conference Record of the Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[Sta85] E.W. 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.

[Ste93] N. Sterling. WARLOCK — a static data race analysis tool. In *USENIX Technical Conference Proceedings*, pages 97–106, Winter 1993.

[TML97] C.A. Thekkath, T. Mann, and E.K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, October 1997.

[Yah01] 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 Proofs

This appendix contains the proofs of the Thread-Modular and Invariant Verification Theorems. We begin with a lemma relating transitions of a thread S and its translation $[\![S]\!]_G^A$ for actions A and G. The lemma states that a transition of S may be simulated by 0 or more steps of $[\![S]\!]_G^A$, provided the transition satisfies G. If it does not satisfy G, then $[\![S]\!]_G^A$ goes wrong.

Lemma 1 (Translation Simulation) *Let $A, G \in Action$ where G is reflexive.*

1.1 *If $(\sigma_1, S_1) \rightarrow_s (\sigma_2, S_2)$ and $G(\sigma_1, \sigma_2)$, then $(\sigma_1, [\![S_1]\!]^A_G) \rightarrow^*_s (\sigma_2, [\![S_2]\!]^A_G)$.*

1.2 *If $(\sigma_1, S_1) \rightarrow_s$ **wrong**, then $(\sigma_1, [\![S_1]\!]^A_G) \rightarrow^*_s$ **wrong**.*

1.3 *If $(\sigma_1, S_1) \rightarrow_s (\sigma_2, S_2)$ and $\neg G(\sigma_1, \sigma_2)$, then $(\sigma_1, [\![S_1]\!]^A_G) \rightarrow^*_s$ **wrong**.*

Proof of 1.1 The proof is by structural induction on the derivation of $(\sigma_1, S_1) \rightarrow_s (\sigma_2, S_2)$. We show three representative cases:

Case 1 [ACTION OK]: In this case, $S_1 = X \downarrow Y$, $X(\sigma_1, \sigma_2)$, $Y(\sigma_1, \sigma_2)$, and $S_2 = \mathbf{skip}$. Therefore,

$$\begin{array}{rcll} (\sigma_1, [\![S_1]\!]^A_G) & = & (\sigma_1, A^*; X \downarrow (Y \wedge G); A^*) & \\ & \rightarrow^*_s & (\sigma_1, \mathbf{skip}; X \downarrow (Y \wedge G); A^*) & \text{by [LOOP DONE] and [SEQ STEP]} \\ & \rightarrow_s & (\sigma_1, X \downarrow (Y \wedge G); A^*) & \text{by [SEQ SKIP]} \\ & \rightarrow_s & (\sigma_2, A^*) & \text{by [ACTION OK], [SEQ STEP], and [SEQ SKIP]} \\ & = & (\sigma_2, [\![\mathbf{skip}]\!]^A_G) & \end{array}$$

Case 2 [CHOICE]: In this case, $\sigma_1 = \sigma_2$, $S_1 = T_1 \Box T_2$, and $S_2 = T_i$ where $i \in \{1, 2\}$. Therefore,

$$\begin{array}{rcll} (\sigma_1, [\![S_1]\!]^A_G) & = & (\sigma_1, A^*; ([\![T_1]\!]^A_G \Box [\![T_2]\!]^A_G)) & \\ & \rightarrow_s & (\sigma_1, [\![T_1]\!]^A_G \Box [\![T_2]\!]^A_G) & \text{by [LOOP DONE], [SEQ STEP], and [SEQ SKIP]} \\ & \rightarrow_s & (\sigma_2, [\![T_i]\!]^A_G) & \text{by [CHOICE]} \\ & = & (\sigma_2, [\![S_2]\!]^A_G) & \end{array}$$

Case 3 [SEQ STEP]: In this case, $S_1 = T_1; T_2$, $(\sigma_1, T_1) \rightarrow_s (\sigma_2, U_1)$, and $S_2 = U_1; T_2$. By induction, $(\sigma_1, [\![T_1]\!]^A_G) \rightarrow^*_s (\sigma_2, [\![U_1]\!]^A_G)$. Therefore,

$$\begin{array}{rcll} (\sigma_1, [\![S_1]\!]^A_G) & = & (\sigma_1, [\![T_1]\!]^A_G; [\![T_2]\!]^A_G) & \\ & \rightarrow^*_s & (\sigma_2, [\![U_1]\!]^A_G; [\![T_2]\!]^A_G) & \text{by [SEQ STEP]}^* \\ & = & (\sigma_2, [\![S_2]\!]^A_G) & \end{array}$$

where [SEQ STEP]* denotes possibly many applications of the rule [SEQ STEP].

Case 4 [ASSOC]: In this case, $\sigma_1 = \sigma_2$, $S_1 = (T_1; T_2); T_3$, and $S_2 = T_1; (T_2; T_3)$. Therefore,

$$\begin{array}{rcll} (\sigma_1, [\![S_1]\!]^A_G) & = & (\sigma_1, ([\![T_1]\!]^A_G; [\![T_2]\!]^A_G); [\![T_3]\!]^A_G) & \\ & \rightarrow_s & (\sigma_1, [\![T_1]\!]^A_G; ([\![T_2]\!]^A_G; [\![T_3]\!]^A_G)) & \text{by [ASSOC]} \\ & = & (\sigma_2, [\![S_2]\!]^A_G) & \end{array}$$

The remaining cases are similar. □

Proof of 1.2 The proof is by structural induction on the derivation of $(\sigma_1, S_1) \rightarrow_s$ **wrong**. There are two cases:

Case 1 [ACTION WRONG]: In this case, $S_1 = X \downarrow Y$, $X(\sigma_1, \sigma_2)$, and $\neg Y(\sigma_1, \sigma_2)$. This implies that $(\sigma_1, X \downarrow (Y \wedge G)) \to_s$ **wrong** by rule [ACTION WRONG]. Therefore,

$$\begin{array}{rcll} (\sigma_1, [\![S_1]\!]^A_G) & = & (\sigma_1, A^*; X \downarrow (Y \wedge G); A^*) & \\ & \to^*_s & (\sigma_1, \textbf{skip}; X \downarrow (Y \wedge G); A^*) & \text{by [LOOP DONE] and [SEQ STEP]} \\ & \to_s & (\sigma_1, X \downarrow (Y \wedge G); A^*) & \text{by [SEQ SKIP]} \\ & \to_s & \textbf{wrong} & \text{by [SEQ WRONG]} \end{array}$$

Case 2 [SEQ WRONG]: In this case, $S_1 = T_1; T_2$ and $(\sigma_1, T_1) \to_s$ **wrong**. By induction, $(\sigma_1, [\![T_1]\!]^A_G) \to^*_s$ **wrong**. Therefore,

$$\begin{array}{rcll} (\sigma_1, [\![S_1]\!]^A_G) & = & (\sigma_1, A^*; ([\![T_1]\!]^A_G; [\![T_2]\!]^A_G)) & \\ & \to^*_s & (\sigma_1, \textbf{skip}; [\![T_1]\!]^A_G; [\![T_2]\!]^A_G) & \text{by [LOOP DONE] and [SEQ STEP]} \\ & \to_s & (\sigma_1, [\![T_1]\!]^A_G; [\![T_2]\!]^A_G) & \text{by [SEQ SKIP]} \\ & \to^*_s & \textbf{wrong} & \text{by [SEQ STEP]}^* \text{ and [SEQ WRONG]} \end{array}$$

□

Proof of 1.3 The proof is by structural induction on the derivation of $(\sigma_1, S_1) \to_s (\sigma_2, S_2)$. There are three cases:

Case 1 [ACTION OK]: In this case, $S_1 = X \downarrow Y$, $X(\sigma_1, \sigma_2)$, $Y(\sigma_1, \sigma_2)$, and $\neg G(\sigma_1, \sigma_2)$. This implies that $(\sigma_1, X \downarrow (Y \wedge G)) \to_s$ **wrong** by rule [ACTION WRONG]. Therefore,

$$\begin{array}{rcll} (\sigma_1, [\![S_1]\!]^A_G) & = & (\sigma_1, A^*; X \downarrow (Y \wedge G); A^*) & \\ & \to^*_s & (\sigma_1, \textbf{skip}; X \downarrow (Y \wedge G); A^*) & \text{by [LOOP DONE] and [SEQ STEP]} \\ & \to_s & (\sigma_1, X \downarrow (Y \wedge G); A^*) & \text{by [SEQ SKIP]} \\ & \to_s & \textbf{wrong} & \text{by [SEQ WRONG]} \end{array}$$

Case 2 [SEQ STEP]: In this case, $S_1 = T_1; T_2$ and $(\sigma_1, T_1) \to_s (\sigma_2, U_1)$, and $S_2 = U_1; T_2$. By induction, $(\sigma_1, [\![T_1]\!]^A_G) \to^*_s$ **wrong**. Therefore,

$$\begin{array}{rcll} (\sigma_1, [\![S_1]\!]^A_G) & = & (\sigma_1, [\![T_1]\!]^A_G)[\![T_2]\!]^{A^*}_G & \\ & \to^*_s & \textbf{wrong} & \text{by [SEQ WRONG]}^* \end{array}$$

where [SEQ WRONG]* refers to a derived rule in which the first statement of a sequential composition goes wrong after possibly many steps.

Case 3 [SEQ SKIP], [ASSOC], [CHOICE], [LOOP DONE], [LOOP UNROLL]: None of these transitions update the store. Hence, $\sigma_2 = \sigma_1$. From the hypotheses of the lemma, G is reflexive, which contradicts the hypothesis $\neg G(\sigma_1, \sigma_2)$.

□

We next show that the translation begins with a yield point at which the abstraction of the other threads may make an arbitrary number of steps.

Lemma 2 *Let $A, G \in$ Action where G is reflexive. Given any store σ and statement S, $(\sigma, [\![S]\!]^A_G) \rightarrow^*_s (\sigma, A; [\![S]\!]^A_G)$.*

Proof Proof is by structual induction on S. Two representative cases are shown:

Case 1 $S = S_1 \Box S_2$: In this case, $[\![S]\!]^A_G = A^*; ([\![S_1]\!]^A_G \Box [\![S_2]\!]^A_G)$. Therefore,

$$\begin{aligned}
(\sigma, [\![S]\!]^A_G) &\rightarrow_s (\sigma, (A; A^*); ([\![S_1]\!]^A_G \Box [\![S_2]\!]^A_G)) \quad \text{by [LOOP UNROLL]} \\
&\rightarrow_s (\sigma, A; (A^*; ([\![S_1]\!]^A_G \Box [\![S_2]\!]^A_G))) \quad \text{by [ASSOC]} \\
&= (\sigma, A; [\![S]\!]^A_G)
\end{aligned}$$

Case 2 $S = S_1; S_2$: By induction, $(\sigma, [\![S_1]\!]^A_G) \rightarrow^*_s (\sigma, A; [\![S_1]\!]^A_G)$. Therefore,

$$\begin{aligned}
(\sigma, [\![S]\!]^A_G) &= (\sigma, [\![S_1]\!]^A_G; [\![S_2]\!]^A_G) \\
&\rightarrow^*_s (\sigma, (A; [\![S_1]\!]^A_G); [\![S_2]\!]^A_G) \quad \text{by [SEQ STEP]}^* \\
&\rightarrow_s (\sigma, A; ([\![S_1]\!]^A_G; [\![S_2]\!]^A_G)) \quad \text{by [ASSOC]} \\
&= (\sigma, A; [\![S]\!]^A_G)
\end{aligned}$$

□

Before stating the next lemma, we define the following translation on parallel states. This function simplifies the notation for constructing an i-abstraction from a parallel program in a given state.

$$\begin{aligned}
[\![\bullet]\!]_i : \mathit{ParState} &\rightarrow \mathit{SeqState} \\
[\![(\sigma, S_1 \mid \cdots \mid S_n)]\!]_i &= (\sigma, [\![S_i]\!]^{G_i}_{A_i}) \\
[\![\mathbf{wrong}]\!]_i &= \mathbf{wrong}
\end{aligned}$$

The assume-guarantee decomposition to use in the translation is implicit from the context. We may now show that any step taken by a parallel thread either can be simulated by every i-abstraction of the program or causes at least one i-abstraction to go wrong.

Lemma 3 *Let $P = S_1 \mid \cdots \mid S_n$ be a parallel program with assume-guarantee decomposition $\langle A_1, G_1\rangle, \ldots, \langle A_n, G_n\rangle$. Let $\Theta_1 = (\sigma_1, P)$ for some store $\sigma_1 \in$ Store. If $\Theta_1 \rightarrow_p \Theta_2$, then either:*

1. $\forall j \in 1..n.\ [\![\Theta_1]\!]_j \rightarrow^*_s [\![\Theta_2]\!]_j$
2. $\exists j \in 1..n.\ [\![\Theta_1]\!]_j \rightarrow^*_s \mathbf{wrong}$

Proof The proof is by case analysis on the rule used to derive $\Theta_1 \rightarrow_p \Theta_2$:

Case 1 [PARALLEL]: In this case, let

$$\Theta_2 = (\sigma_2, S_1 \mid \cdots \mid T_i \mid \cdots \mid S_n)$$

where $(\sigma_1, S_i) \rightarrow_s (\sigma_2, T_i)$. There are two cases:

Case 1.1 $G_i(\sigma_1, \sigma_2)$: By Lemma 1.1, $[\![\Theta_1]\!]_i \rightarrow_s^* [\![\Theta_2]\!]_i$. Also for $j \neq i$, the definition of $[\![\Theta_1]\!]_j$ yields

$$\begin{array}{rll} [\![\Theta_1]\!]_j & \rightarrow_s^* (\sigma_1, A_j; [\![S_j]\!]_{G_j}^{A_j}) & \text{by Lemma 2} \\ & \rightarrow_s (\sigma_2, [\![S_j]\!]_{G_j}^{A_j}) & \text{because } G_i(\sigma_1, \sigma_2) \text{ and } G_i \subseteq A_j \\ & = [\![\Theta_2]\!]_j & \end{array}$$

Thus, $\forall j \in 1..n.\ [\![\Theta_1]\!]_j \rightarrow_s^* [\![\Theta_2]\!]_j$.

Case 1.2 $\neg G_i(\sigma_1, \sigma_2)$: Lemma 1.3 indicates that $[\![\Theta]\!]_i = (\sigma_1, [\![S_i]\!]_{G_i}^{A_i}) \rightarrow_s^* \textbf{wrong}$.

Case 2 [PARALLEL WRONG]: $(\sigma_1, S_i) \rightarrow_s \textbf{wrong}$ for some i. By Lemma 1.2, $[\![\Theta_1]\!]_i = (\sigma_1, [\![S_i]\!]_{G_i}^{A_i}) \rightarrow_s^* \textbf{wrong}$.

□

We now extend the previous lemma to parallel execution sequences of any length.

Lemma 4 *Let $P = S_1 \mid \cdots \mid S_n$ be a parallel program with assume-guarantee decomposition $\langle A_1, G_1\rangle, \ldots, \langle A_n, G_n\rangle$. Let $\Theta_1 = (\sigma_1, P)$ for some store $\sigma_1 \in Store$. If $\Theta_1 \rightarrow_p^* \Theta_2$, then either:*

1. $\forall j \in 1..n.\ [\![\Theta_1]\!]_j \rightarrow_s^* [\![\Theta_2]\!]_j$
2. $\exists j \in 1..n.\ [\![\Theta_1]\!]_j \rightarrow_s^* \textbf{wrong}$

Proof Proof is by induction on k, the length of the derivation of $\Theta_1 \rightarrow_p^* \Theta_2$:

Case 1 $k = 0$: In this case, $\Theta_2 = \Theta_1$ and $\forall j \in 1..n.\ [\![\Theta_1]\!]_j \rightarrow_s^* [\![\Theta_2]\!]_j$ is trivially satisfied.

Case 2 $k > 0$: In this case, $\Theta_1 \rightarrow_p^{k-1} \Theta_3 \rightarrow_p \Theta_2$ where, by induction, one of the following two cases holds:

Case 2.1 $\forall j \in 1..n.\ [\![\Theta_1]\!]_j \rightarrow_s^* [\![\Theta_3]\!]_j$: Lemma 3 applied to $\Theta_3 \rightarrow_p \Theta_2$ indicates that one of the following holds:

1. $\forall j \in 1..n.\ [\![\Theta_3]\!]_j \rightarrow_s^* [\![\Theta_2]\!]_j$: Consider any $j \in 1..n$. In this case, we have $[\![\Theta_1]\!]_j \rightarrow_s^* [\![\Theta_2]\!]_j$.
2. $\exists j \in 1..n.\ [\![\Theta_3]\!]_j \rightarrow_s^* \textbf{wrong}$: We know that $[\![\Theta_1]\!]_j \rightarrow_s^* [\![\Theta_3]\!]_j \rightarrow_s^* \textbf{wrong}$. Thus, part 2 of the lemma is proven.

Case 2.2 $\exists j \in 1..n.\ [\![\Theta_1]\!]_j \rightarrow_s^* \textbf{wrong}$: This implies the second condition in the statement of the lemma is satisfied.

□

The Thread-Modular Verification Theorem from Section 4 follows directly from this lemma.

Theorem 1 (Thread-Modular Verification) *Let $P = S_1 \mid \cdots \mid S_n$ be a parallel program with assume-guarantee decomposition $\langle A_1, G_1\rangle, \ldots, \langle A_n, G_n\rangle$. For all $\sigma \in Store$, if $\forall i \in 1..n.\ (\sigma, [\![S_i]\!]_{G_i}^{A_i}) \not\rightarrow_s^*$* **wrong***, then $(\sigma, P) \not\rightarrow_p^*$* **wrong**.

Proof This theorem is a direct consequence of Lemma 4. □

The machinery developed to prove the previous theorem may also be used to prove the Invariant Verification Theorem, restated below.

Theorem 2 (Invariant Verification) *Let $P = S_1 \mid \cdots \mid S_n$ be a parallel program with assume-guarantee decomposition $\langle A_1, G_1\rangle, \ldots, \langle A_n, G_n\rangle$, and let Init and I be sets of stores. Suppose:*

1. $\mathit{Init} \subseteq I$
2. $\forall i \in 1..n.\ G_i \Rightarrow (I \Rightarrow I')$
3. $\forall i \in 1..n.\ \forall \sigma \in \mathit{Init}.\ (\sigma, [\![S_i]\!]_{G_i}^{A_i}) \not\rightarrow_s^* \textbf{wrong}$

Then I is an invariant of P with respect to Init.

Proof To show that I is an invariant of P with respect to *Init*, we show that for all $\sigma_1 \in \mathit{Init}$, if $(\sigma_1, P) \rightarrow^* (\sigma_2, P_2)$, then $\sigma_2 \in I$. The proof is by induction on the length k of the derivation of $(\sigma_1, P) \rightarrow_p^* (\sigma_2, P_2)$.

Case 1 $k = 0$: In this case, $\sigma_2 = \sigma_1 \in \mathit{Init}$, and $\sigma_2 \in I$ by condition 1.

Case 2 $k > 0$: In this case, $(\sigma_1, P) \rightarrow_p^{k-1} (\sigma_3, P_3) \rightarrow_p (\sigma_2, P_2)$, where the inductive hypothesis ensures that $\sigma_3 \in I$. Let

$$\begin{array}{rcl} P_3 & = & T_1 \mid \cdots \mid T_i \mid \cdots \mid T_n \\ P_2 & = & T_1 \mid \cdots \mid U_i \mid \cdots \mid T_n \end{array}$$

where $(\sigma_3, T_i) \rightarrow_s (\sigma_2, U_i)$. By Lemma 4 and condition 3, it must be the case that

$$\begin{array}{rl} [\![(\sigma_1, P)]\!]_i & \rightarrow_s^* [\![(\sigma_3, P_3)]\!]_i \\ & = (\sigma_3, [\![T_i]\!]_{G_i}^{A_i}) \\ & \not\rightarrow_s^* \textbf{wrong} \end{array}$$

Given this fact and the transition $(\sigma_3, T_i) \rightarrow_s (\sigma_2, U_i)$, Lemma 1.3 shows that $G(\sigma_3, \sigma_2)$. This, combined with $\sigma_3 \in I$ and condition 2, implies that $I(\sigma_2)$.

□