Checking Concise Specifications For Multithreaded Software

Williams College Technical Note 01-2002, Dec. 2002 (revised Dec. 2003)

Stephen N. Freund
Computer Science Department
Williams College
Williamstown, MA 01267

Shaz Qadeer
Microsoft Research
One Microsoft Way
Redmond, WA 98052

Abstract

Ensuring the reliability of multithreaded software systems is difficult due to the potential for subtle interactions between threads. We present a new modular verification technique to check concise specifications of large multithreaded programs. Our analysis scales to systems with large numbers of procedures and threads. We achieve thread-modular analysis by annotating each shared variable by an *access predicate* that summarizes the condition under which a thread may access that variable. We achieve procedure-modular analysis by annotating each procedure with a specification related to its implementation by an abstraction relation combining the notions of *simulation* and *reduction*. We have implemented our analysis in Calvin-R, a static checker for multithreaded Java programs.

1 Introduction

Software verification is an important and difficult problem. Over the past few decades, a variety of techniques based on dataflow analysis, theorem proving, and model checking have emerged for the analysis of sequential software. However, these techniques have not yet enabled verification of large, multithreaded software systems. Since concurrency is an insidious source of programming errors, multithreaded programs would benefit significantly from automated error-detection tools. The need for such tools will continue to grow as multithreaded software becomes even more widespread, expanding from the domain of low-level systems software (operating systems and databases) to most programs written in high-level languages like Java and C#. In this paper, we present a new modular verification technique for multithreaded Java programs.

Modularity is the key to scaling program analyses to large software systems. For sequential programs, modular analysis is achieved through pre- and post-conditions for procedures. However, due to interaction among the threads, pre- and post-conditions are insufficient for modular verification of multithreaded programs. Jones [24] proposed the first proof rule for modular verification of multithreaded programs. The proof rule of Jones required, in addition to pre- and post-conditions, a rely-guarantee specification for each procedure to capture the interaction among the threads. Both the rely and the guarantee specifications are actions (binary relations on the shared store). The guarantee specification is a requirement on the updates performed by the thread executing the procedure, and the rely specification is a requirement on the updates performed by the other threads.

In previous work [18], we extended Jones' method by generalizing the guarantee of a procedure from a single action to a program constructed from actions. This *guarantee program* has the property that every sequence of atomic updates to shared variables in the implementation is matched by a sequence of atomic updates in the specification. The implementation and guarantee of a procedure must be related via *simulation*. Simulation provides *data abstraction* for multithreaded programs by supporting an abstract description of each action in the sequence of actions performed during the execution of a procedure. In particular, simulation allows steps that modify only local variables to be abstracted away. This generalization is crucial for allowing modular specification and verification of a multithreaded library independently from the clients of the library.

Unfortunately, simulation requires every step in a procedure's implementation that updates a shared variable to be matched by a step in its specification. Consider, for example, a multithreaded program in which the shared variable `count` is protected by the mutex `m`. The following procedure increments `count` by one.

```
void increment() {
    acquire(m);
    int j = count;
    j++;
    count = j;
    release(m);
}
```

A specification that simulates the implementation must have at least three steps corresponding to acquiring `m`, updating `count`, and releasing `m`. Consequently, such a specification is no more concise or intuitive than the implementation. Although the programmer's intuition is that the execution of `increment` by a thread appears to happen "in one step", simulation does not by itself allow us to prove this.

We introduce a new and more expressive criterion for relating the implementation of a procedure to its specification. The new relation augments simulation with the notion of reduction, which was first introduced by Lipton [26]. The notion of reduction is based on commuting operations performed by different threads when they do not interfere with each other. An operation that commutes to the right of a succeeding operation by a different thread is a *right mover*, and an operation that commutes to the left of a preceding operation by a different thread is a *left mover*. For example, the operation `acquire(m)` is a right mover, and the operation `release(m)` is a left mover. Moreover, since all threads access `count` only while holding the mutex `m`, the read from `count` and write to `count` are both right and left movers since no other thread can concurrently access `count`.

Any execution sequence in which a thread performs a sequence of right movers followed by a single atomic operation followed by a sequence of left movers can be viewed as occurring "in one step". The execution of `increment` by a thread has this property. To check that `increment` atomically increments `count` by one, we first apply reduction and then check simulation only on the reduced sequence. Thus, reduction allows control abstraction for multithreaded programs akin to pre- and post-conditions for sequential programs.

Recent work by Flanagan, Freund, and Qadeer [16, 13] suggests that the large majority of methods in multithreaded programs are reducible to a single atomic step. For these methods, Calvin-R specifications are no more complex than specifications under the assumption that the program is single-threaded. Thus, we expect that for many methods, the overhead of using Calvin-R will not be much higher than that of existing sequential checkers.

The abstraction mechanism in Calvin-R permits many non-atomic methods to be handled in an equally straight-forward way. As an example, consider the following method that models allocation of resources. The method searches for a false bit in the `bits` array and allocates that "resource" by setting that bit to true. The resource `bits[`i`]` is protected by the lock `l[`i`]`, for each i.

```
public static int alloc() {
    for (int i = 0; i < NBLOCKS; i++) {
        acquire(l[i]);
        if (!bits[i]) {
            bits[i] = true;
            return i;
        }
        release(l[i]);
    }
    return -1;
}
```

Clearly, this method does not reduce to one atomic sequence because it may acquire and release a series of locks before finding a false bit. However, by taking advantage of both reduction and simulation, we are able to specify the functional behavior of `alloc` as a single operation that selects and sets a false bit from the array (or returns -1 if no free resource is found). We develop such a specification and show how Calvin-R checks it in Section 2. We are not aware of any other automated checking tool that uses a combination of simulation and reduction to check abstraction.

In order to apply reduction to code sequences that access shared variables, the locking discipline for these shared variables must be specified by the programmer. Although mutexes are the most common synchronization discipline, a variety of other mechanisms are used in practice [31, 14]. To capture these idioms, we introduce *access predicates*, a general mechanism for describing a wide variety of synchronization mechanisms including mutexes, readers-writer locks, data-dependent locking, etc. The access predicate for a variable expresses the condition under which a thread may access it. Our verification technique checks the code of each thread assuming that the environment (containing other threads) behaves according to the access predicates.

We have implemented our analysis in the Calvin-R checker, an extension of the Calvin checker for multithreaded Java programs [14, 18]. Our tool modularly checks that each method in a program satisfies the access predicates and is abstracted by its specification. For each check, Calvin-R constructs a sequential program capturing the necessary correctness requirements and verifies that this program does not go wrong using existing verification techniques for sequential programs. Specifically, we employ verification conditions [9, 19] and the Simplify automatic theorem prover [28].

As a first case study, we used Calvin-R to check properties of Daisy, a simple multithreaded NFS file system designed to test Calvin-R. The file system synchronization mechanisms are similar in complexity to those found in other file systems. However, the data structures and algorithms in Daisy are relatively simple, allowing us to implement it in approximately 1200 lines of Java code. We have verified that all procedures in Daisy satisfy the access predicates, showing that the code adheres to the specified synchronization discipline. In addition, we specified and checked functional requirements for the most complex procedures in Daisy.

2 Overview

We demonstrate our specification and verification system with two example classes. The first implements a counter, and the second implements block allocation in a simple file system. In both cases, we state concise specifications for the code and describe how our analysis checks them.

2.1 Counter

The following class implements a counter:

Figure 1: Counter

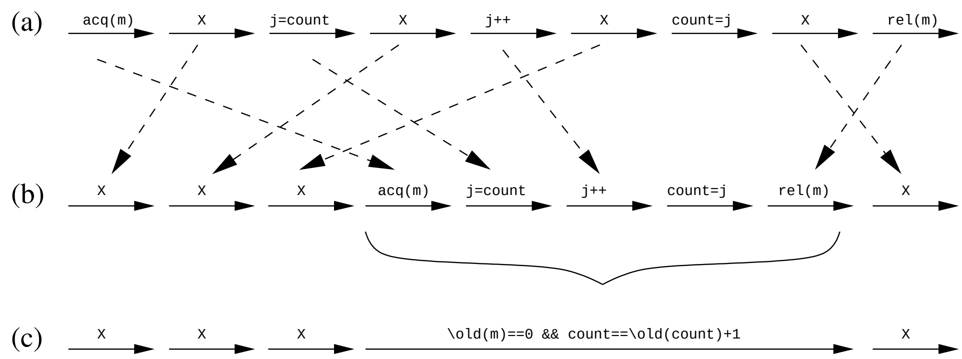
```
class Counter {
  int m /*@ accessible_if m == 0 || m == tid */;
  int count /*@ accessible_if m == 0 || m == tid */;

  /*@ performs action (count) {
               \old(m) == 0 && count == \old(count) + 1 } */
  void increment() {
      acquire(m); int j = count; j++; count = j; release(m);
  }
}
```

The `increment` method adds one to `count`. Concurrent calls to `increment` are serialized using the mutex lock `m`, which is acquired at the beginning and released at the end of `increment`. We model the mutex as an integer variable whose value is the identifier of the thread holding the mutex, or 0 if it is not held. The atomic operation `acquire(m)` blocks until `m` is 0 and then sets `m` to the identifier of the currently executing thread (`tid`), and the atomic operation `release(m)` sets `m` back to 0. We use `acquire` and `release` in place of the built-in Java synchronization operations only to simplify the technical development of our analysis.

The `performs` annotation specifies method behavior. According to `increment`'s specification, the method behaves as if at some point during its execution, it performs a single atomic action that (1) modifies only the variable `count`, which is indicated with the modifies clause `(count)`; and (2) blocks until the value of `m` is 0 and then increments `count` by 1. The values `\old(m)` and `\old(count)` refer to the variable values in the pre-state of this action.

The `accessible_if` annotations indicate to our checker the access predicates for the variables. The access predicates for `m` and `count`, denoted $A_{\texttt{m}}$ and $A_{\texttt{count}}$, express the requirement that a thread t may access `m` and `count` only if $\texttt{m} = 0$ or $\texttt{m} = t$. It is worth noting that the access predicates for the shared variables do not preclude

Figure 2: Checking abstraction for `increment`



data races. As we have pointed out earlier [16], absence of data races is neither necessary nor sufficient for atomicity.

The method `increment` can be called (possibly concurrently) by any number of threads. To ensure the specification of `increment` is valid for any calling thread, Calvin-R checks the method for an arbitrary thread t with the assumption that other threads operating concurrently with thread t access `m` and `count` according to the access predicates. For each execution trace of the method in such an environment, the tool checks both that thread t satisfies the access predicates and that the execution is abstracted by an execution of the specification of `increment` in the same environment. The first check can be easily performed using the technique of thread-modular verification [24, 14, 18].

To check abstraction, Calvin-R *reduces* the given execution to another execution that ends in the same final store and in which the operations performed by thread t (in `increment`) happen atomically without any interleaved actions by the environment. After reduction, the tool checks that the single atomic action in the specification of `increment` *simulates* the composition of the consecutively occurring actions of thread t.

To reduce an execution trace, the tool shows that the operations by thread t in the execution form a sequence of zero or more right-commuting operations (*right movers*) followed by a single operation followed by a sequence of zero or more left-commuting operations (*left movers*).

An operation of t is a right mover if, immediately after its execution, no other thread can access a variable accessed by that operation. Since the environment operations behave according to the access predicates, we can derive the condition to verify that an operation is right-mover from the access predicates. For example, if the operation by thread t accesses variable `m`, the condition $E_{\mathtt{m}}(t)$ must be shown in the post-store:

$$\begin{array}{rcl} E_{\mathtt{m}}(t) & = & \forall j \in \mathit{Tid}.\; j \neq t \Rightarrow \neg A_{\mathtt{m}}(j) \\ & = & \forall j \in \mathit{Tid}.\; j \neq t \Rightarrow \neg(\mathtt{m} = 0 \vee \mathtt{m} = j) \\ & = & (\mathtt{m} = t) \end{array}$$

Thus, to prove that an operation by thread t accessing `m` is a right mover, we must show that thread t holds `m` in the post-store of the operation. Intuitively, the predicate $E_{\mathtt{m}}(t)$ is the condition under which thread t has *exclusive access* to `m`.[1] The condition for an operation accessing `count` is identical. We may commute a right mover with the operation following it because we are guaranteed that the two operations access disjoint sets of variables.

Similarly, an operation is a left mover if we can prove that `m` is held by thread t in the pre-store. For `increment`, we can show that `acquire(m)` and all subsequent operations until `release(m)` are right movers and `release(m)` is a left mover. Therefore, the code in `increment` is reducible to a single action.

Figure 2(a) shows an execution of `increment` by thread t interleaved with arbitrary actions of other threads. Actions from other threads are labeled `X`, and for simplicity we insert only a single such action between consecutive actions of t. Execution (b) shows the reduced execution.

The simulation check is straight-forward once the execution has been reduced, as shown in (c).

Note that the specification of `increment` shown above is comparable in complexity to the post-condition specification for `seq_increment`, a version of `increment` designed for sequential programs:

```
/*@ modifies count; ensures count == \old(count) + 1 */
void seq_increment() { int t = count; t++; count = t; }
```

Our recent work suggests that a large fraction of methods in multithreaded systems are written to be free of interference and may be considered to execute atomically [13]. In these cases, Calvin-R's abstraction relation enables us to specify a method's behavior in a fashion no more complex than specifying a method's sequential behavior.

2.2 Block allocation in Daisy

To further illustrate the importance of using both reduction and simulation for proving succinct procedure specifications, we present in Figure 3 the code for block allocation from a simple file system implemented as an initial case study for Calvin-R.

The `alloc` method searches for a free file system block by finding a false bit in the `bits` array. The flag `bits[j]` indicates whether

[1] Alternatively, we could have specified the exclusive access predicate and derived the access predicate from it.

Figure 3: Allocator

```
class Allocator {
  /*@ accessible_if[j] l[j] == 0 || l[j] == tid */
  static int l[NBLOCKS];
  /*@ accessible_if[j] l[j] == 0 || l[j] == tid */
  static boolean bits[NBLOCKS];

  /*@ performs action "NoBlocks": () { \result == -1 }
        [] action "Allocated": (bits[\result], l[\result]) {
             0 <= \result && \result < NBLOCKS &&
             \old(l[\result]) == 0 && l[\result] == tid &&
             !\old(bits[\result]) && bits[\result]
      }
  */
  public static int alloc() {
      for (int i = 0; i < NBLOCKS; i++) {
          acquire(l[i]);
          if (!bits[i]) {
              bits[i] = true;
              //@ witness "Allocated";
              return i;
          }
          release(l[i]);
      }
      //@ witness "NoBlocks";
      return -1;
  }

  /*@ requires l[i] == tid
      performs action "Free": (bits[i], l[i])
                      { l[i] == 0 && !bits[i] }
  */
  public static void free(int i) {
      bits[i] = false;
      release(l[i]);
      //@ witness "Free";
  }
}
```

the j-th disk block is currently in use. When `alloc` identifies a free block, it allocates the block by setting the appropriate bit to true and returns the index of the block with the lock corresponding to it still held. (The caller will release the lock after it has finished initializing the data structures for the new block.) The `alloc` method returns -1 if it fails to find a free block.

The mutex `l[j]` guards the bit `bits[j]`. The `accessible_if` annotation on `bits` is parameterized by `j` to indicate this relationship for all `j`. Locking at such a fine granularity is a standard technique for improving throughput in commercial file systems. However, it is also a major source of errors and demands substantial debugging effort.

The `free` method takes a block index `i` as an argument and requires that the the mutex `l[i]` be held on entry to the method. It frees block `i` by setting `bits[i]` to false and returns after releasing `l[i]`.

The `performs` annotation for `alloc` is intuitive and mirrors the two possible outcomes of executing `alloc`. The specification is a choice between two atomic actions. In the first action, no free block is found and -1 is returned. The special variable `\result` refers to the value returned by a method. In the second action, the return value is the index of an unused block. This action blocks until the mutex protecting the allocation bit of the block is zero, and it then updates the bit from false to true. The `witness` annotations in the code indicate program points where simulation steps in the specification may occur. We use the explicit witness to guide the simulation check by indicating the "commit points" in the implementation of the atomic actions in the specification.[2] The specification for the `free` method is similar.

As in the `increment` example, Calvin-R checks that the implementation of `alloc` is abstracted by its specification for an arbitrary thread t in an environment that respects the access predicates. However, the checking of `alloc` is significantly more complicated than that of `increment`. The `alloc` method has an unbounded number of execution sequences, each consisting of 0 or more *acquire-test-release* subsequences, followed by either a return of -1 or an *acquire-test-set* and a return of a non-negative index. Such executions are not reducible to a single atomic action. Therefore, our checker decomposes the sequence of actions performed by thread t into subsequences that are each reducible to a single action, as shown in Figure 4(a) and (b) for one possible execution.

Calvin-R deduces that each *acquire-test-release* subsequence is reducible to a single atomic action, and further checks that each of these actions is simulated by `skip`, an action that leaves every variable unchanged. If there is no final *acquire-test-set* sequence, then Calvin-R further deduces that the last implementation action returns -1 and is therefore simulated by the action `"NoBlocks"` of the specification. If there is a final *acquire-test-set* sequence followed by the return of a non-negative index, Calvin-R reduces it to a single atomic action and checks that it is simulated by the action `"Allocated"` of the specification. In both cases, by first using reduction and then simulation, Calvin-R abstracts the execution to a (possibly empty) sequence of `skip` action followed by an action from the specification.[3] Figure 4 illustrates one possible execution of `alloc`. Execution (b) shows the reduced execution of (a), and (c) and (d) demonstrate the simulation. We divide the simulation into two steps to show that simulation involves composing a sequence of actions into a single action, as well as generalizing an action.

Although `alloc` uses fine-grained synchronization, our method allows us to prove a concise and intuitive specification that is similar in complexity to the specification of `alloc` assuming single-threaded execution.

3 Plato language

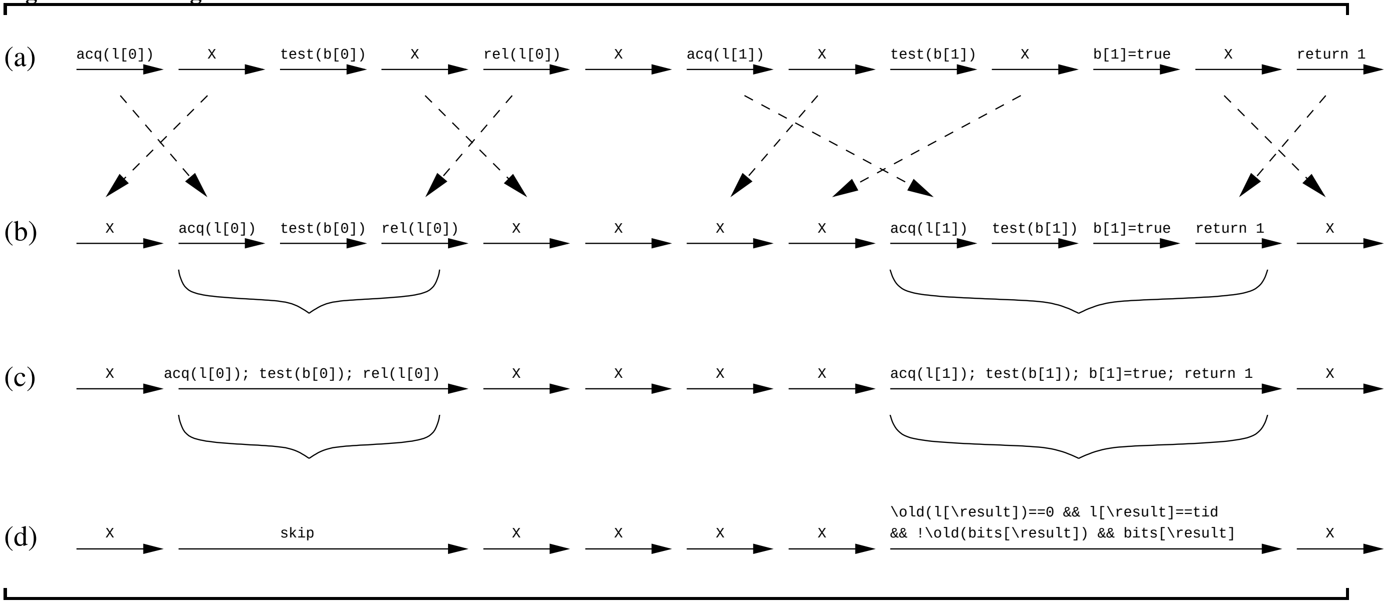
In order to avoid the complexity of reasoning about programs written in a large language like Java, Calvin-R translates concurrent programs into a minimal intermediate language called Plato (parallel language of atomic operations) [14, 18]. The rest of this paper focuses on verification of programs once they have been translated into Plato.

A Plato program P is the parallel composition of one or more sequential statements. When P is executed, the statements are run in separate threads. Each thread is assigned a unique identifier (a positive integer), and a scheduler interleaves steps taken by the threads nondeterministically. Threads operate on a shared store σ, which maps variable names to values, but we leave the exact representation of variables and values unspecified because they are orthogonal to the key ideas of Plato. A sequential statement may be an atomic operation (described below), a sequential composition $S_1; S_2$, a nondeterministic choice $S_1 \Box S_2$ that executes either S_1 or S_2, an

[2] In general, it is difficult to infer the correspondence between implementation steps and specification steps automatically, but it is trivial for the programmer to specify the correspondence.

[3] The `performs` specification implicitly allows arbitrary number of `skip` actions at any control point.

Figure 4: Checking abstraction for `alloc`



iteration statement S^* that executes S an arbitrary (zero or more) number of times, or a procedure call $m()$. For the remainder of this paper, we assume a fixed function $\mathcal{B}$ mapping procedure names to statements.

Figure 5: Plato syntax

σ	$\in$	*Store*	$=$	$Var \to Value$	
s, t	$\in$	*Tid*	$=$	$\{1, 2, 3, \ldots\}$	
p, q	$\in$	*Predicate*	$\subseteq$	$Tid \times Store$	
X, Y	$\in$	*Action*	$\subseteq$	$Tid \times Store \times Store$	
l, m	$\in$	*Proc*			
$\mathcal{B}$	$\in$	*Defn*	$=$	$Proc \to Stmt$	
P, Q	$\in$	*Program*	$::=$	$S_1 \parallel \ldots \parallel S_n$	
S, T, U	$\in$	*Stmt*	$::=$	a	atomic op
			$\mid$	$S_1 ; S_2$	composition
			$\mid$	$S_1 \Box S_2$	choice
			$\mid$	S^*	iteration
			$\mid$	$m()$	procedure call
a, b, c	$\in$	*AtomicOp*	$::=$	$p?X$	

An atomic operation has the form $p?X$, where the predicate p and the action X are parameterized by the identifier of the thread executing $p?X$. The predicate p is an assertion about the pre-store of the operation, and the action X is a predicate over two stores describing the effect of the operation. When a thread with identifier t executes the atomic operation $p?X$ in store σ, there are three possible outcomes:

1. If $p(t, \sigma)$ holds, the program moves into a post-store σ' such that $X(t, \sigma, \sigma')$.
2. If $p(t, \sigma)$ holds but no such σ' exists, the thread blocks.
3. If $p(t, \sigma)$ is false, execution terminates in the state $\mathbf{wrong}$ to indicate that an error has occurred.

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

For any action X and set of variables $V \subseteq Var$, 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. Several abbreviations and Plato encodings for common programming constructs and locks appear in the figure below:

Figure 6: Conventional constructs in Plato

$\langle X \rangle$	$=$	$\langle X \rangle_\emptyset$
X	$=$	$\mathtt{true}?X$
`skip`	$\stackrel{\text{def}}{=}$	$\langle \mathtt{true} \rangle$
`havoc`	$\stackrel{\text{def}}{=}$	$\langle \mathtt{true} \rangle_{Var} = \mathtt{true}$
`x = `e	$\stackrel{\text{def}}{=}$	$\langle \mathtt{x}' = e \rangle_\mathtt{x}$
`assert `e	$\stackrel{\text{def}}{=}$	$e?\langle \mathtt{true} \rangle$
`assume `e	$\stackrel{\text{def}}{=}$	$\langle e \rangle$
`if `(e)` { `S` }`	$\stackrel{\text{def}}{=}$	$(\mathtt{assume}\ e; S) \Box (\mathtt{assume}\ \neg e)$
`while `(e)` { `S` }`	$\stackrel{\text{def}}{=}$	$(\mathtt{assume}\ e; S)^* ; (\mathtt{assume}\ \neg e)$
`acquire(mx)`	$\stackrel{\text{def}}{=}$	$\langle \mathtt{mx} = 0 \wedge \mathtt{mx}' = \mathtt{tid} \rangle_\mathtt{mx}$
`release(mx)`	$\stackrel{\text{def}}{=}$	$\langle \mathtt{mx}' = 0 \rangle_\mathtt{mx}$

Although Plato does not support dynamic thread creation directly, this feature can be modeled in the language easily.

3.1 Semantics

The Plato semantics is based on execution paths. In order to construct execution paths, we first translate a statement S to the set $[\![S]\!]$ of sequences of atomic operations, as shown below. The sequence $\overline{a};\overline{b}$ is the concatenation of sequences $\overline{a}$ and $\overline{b}$. The set $\varphi_1 ; \varphi_2$ is the set of all sequences obtained by the concatenation of a sequence from φ_1 and a sequence from φ_2. Note that we are overloading the operator ";" to mean both sequential composition of statements and concatenation of sequences and sets of sequences. The set φ^* is the Kleene closure of the set φ. Finally, the set $\overline{u}_1 \otimes \ldots \otimes \overline{u}_n$ contains all interleavings of the paths $\overline{u}_1, \ldots, \overline{u}_n$, and $\psi_1 \otimes \ldots \otimes \psi_n$ is the union of all pathsets obtained by taking the interleavings of a path from each ψ_i for $1 \leq i \leq n$.

Figure 7: Program paths

$$
\begin{array}{rcl}
\overline{a}, \overline{b} & \in & \mathit{Seq} \quad = \quad a_1; \ldots; a_n \\
\varphi & \in & \mathit{SeqSet} \\
u, w & \in & \mathit{Step} \quad = \quad |t, a| \\
\overline{u}, \overline{w} & \in & \mathit{Path} \quad = \quad u_1; \ldots; u_n \\
\psi & \in & \mathit{PathSet}
\end{array}
$$

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

$$
\begin{array}{rcl}
[\![\bullet]\!] & : & \mathit{Stmt} \to \mathit{SeqSet} \\
[\![S]\!] & = & \bigcup_{d \geq 0} [\![S]\!]^d
\end{array}
$$

$$
\begin{array}{rcl}
[\![\bullet]\!] & : & \mathit{Program} \to \mathit{PathSet} \\
[\![S_1 \parallel \ldots \parallel S_n]\!] & = & |1, S_1| \otimes \ldots \otimes |n, S_n|
\end{array}
$$

The behavior of a *thread* $|t, S|$, where t is a thread identifier and S is a statement, is captured by the set $\{|t, \overline{a}| \mid \overline{a} \in [\![S]\!]\}$ of paths. The behavior of a program $P = S_1 \parallel \ldots \parallel S_n$ is captured by the set of paths $[\![P]\!]$ obtained by interleaving the threads $|1, S_1|, \ldots, |n, S_n|$ in all possible ways.

The *transition relation* $\bullet \xrightarrow{\bullet} \bullet \subseteq \mathit{State} \times \mathit{Step} \times \mathit{State}$ maps a state and a step to a state, which is either another store or $\mathbf{wrong}$:

$$\omega \quad \in \quad \mathit{State} \;=\; \mathit{Store} \mid \mathbf{wrong}$$

Figure 8: Transition relation

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

$$
\begin{array}{lll}
\sigma \xrightarrow{u} \sigma' & & \text{if } p(t,\sigma) \text{ and } X(t,\sigma,\sigma') \\
\sigma \xrightarrow{u} \mathbf{wrong} & & \text{if } \neg p(t,\sigma) \\
\mathbf{wrong} \xrightarrow{u} \mathbf{wrong} & &
\end{array}
$$

The transition from $\mathbf{wrong}$ to $\mathbf{wrong}$ simplifies the presentation of our abstraction relation in the next section.

Let $\overline{u} = u_1; \ldots; u_n$ be a path. If $\omega_1 \neq \mathbf{wrong}$ and $\omega_i \xrightarrow{u_i} \omega_{i+1}$ for all i such that $1 \leq i \leq n$, then $\tau = \omega_1 \xrightarrow{u_1} \ldots \xrightarrow{u_{n-1}} \omega_n \xrightarrow{u_n} \omega_{n+1}$ is an *execution* of $\overline{u}$, and each transition $\omega_i \xrightarrow{u_i} \omega_{i+1}$ is a *reachable transition* of $\overline{u}$. The execution τ is $|t, S|$-*complete* if the projection of $\overline{u}$ to steps taken by thread t is a path of $|t, S|$. To simplify our notation, we use the abbreviation $\omega \xrightarrow{|t,\overline{u}|} \omega'$ to describe the execution of path $\overline{u}$ starting in state ω. If $\overline{u}$ is the empty sequence, then $\omega = \omega'$ and $\omega \xrightarrow{|t,\overline{u}|} \omega'$ represents ω. If $\overline{u} = \overline{u}_1; u$, then $\omega \xrightarrow{\overline{u}} \omega'$ represents $\omega \xrightarrow{\overline{u}_1} \omega'' \xrightarrow{u} \omega'$ for some ω''.

If $\overline{u}$ is a path of program P, then τ is also an execution of P. Similarly, if $\overline{u}$ is a path of thread $|t, S|$, then τ is an execution of $|t, S|$. Program P *goes wrong* if an execution of P ends in $\mathbf{wrong}$. Thread $|t, S|$ *goes wrong* if an execution of $|t, S|$ ends in $\mathbf{wrong}$.

4 Concise specifications

We now formalize access predicates and the abstraction mechanism introduced in Section 2.

4.1 Access predicates

Access predicates provide a concise mechanism for expressing a variety of synchronization disciplines. Each variable in a program has an access predicate. The access predicate for variable `x` is denoted by $A_{\mathtt{x}}$. If a thread t performs an operation a accessing the variable `x` from a store σ, then $A_{\mathtt{x}}(t, \sigma)$ must be true.[4]

The set of variables accessed by an atomic operation a is denoted by $\mathit{Access}(a)$. Although we could use a semantic notion of access, it suffices to define $\mathit{Access}(Z)$ for any action, predicate, or atomic operation Z to be the smallest set containing all variables modified or mentioned by Z and closed under the following rule for all `x` and `y`:

If $\mathtt{y} \in \mathit{Access}(Z)$ and $\mathtt{x} \in \mathit{Access}(A_{\mathtt{y}})$, then $\mathtt{x} \in \mathit{Access}(Z)$.

We extend access predicates to any atomic operation a:

$$A_a \quad \stackrel{\text{def}}{=} \quad \bigwedge_{\mathtt{x} \in \mathit{Access}(a)} A_{\mathtt{x}}.$$

A transition $\sigma \xrightarrow{|t,a|} \omega$ *satisfies* the access predicate $A_{\mathtt{x}}$ if $A_{\mathtt{x}}(t, \sigma)$ is true whenever $\mathtt{x} \in \mathit{Access}(a)$, and a transition is *nice* if it satisfies the access predicates for all variables. A program P is *nice* if every reachable transition $\sigma \xrightarrow{|t,a|} \omega$ of P is nice.

Later, we present a modular analysis to check that a program is nice by checking that each individual thread is nice. A thread is nice if it is never the first thread of a program to violate the access predicates. Formally, a thread $|t, S|$ is *nice* if no $|t, S|$-complete execution has some transition $\sigma \xrightarrow{|t,a|} \omega$ as the first transition that is not nice. A program is nice if all of its threads are nice.

Access predicates provide a convenient way to identify potential race conditions in a program. A race condition arises if two threads simultaneously attempt to access a particular variable. Formally, the program P has a *race* on the variable `x` if P has executions $\sigma \xrightarrow{\overline{u}} \sigma' \xrightarrow{|t,a|} \omega_1$ and $\sigma \xrightarrow{\overline{u}} \sigma' \xrightarrow{|e,b|} \omega_2$ such that $t \neq e$, $\mathtt{x} \in \mathit{Access}(a)$, and $\mathtt{x} \in \mathit{Access}(b)$. To prove a program has no races on `x`, we choose $A_{\mathtt{x}}$ with the property

$$\forall \sigma.\ \forall i, j.\ A_{\mathtt{x}}(i, \sigma) \wedge A_{\mathtt{x}}(j, \sigma) \Rightarrow i = j.$$

We then use the modular analysis of this paper to check that every reachable transition of P satisfies $A_{\mathtt{x}}$. If the check succeeds, there is no race on variable `x` in P.

4.2 Abstraction relation

In order to verify properties of large programs, we shall verify each procedure separately, using only specifications of other procedures. Specifications, like the code they describe, are Plato statements. However, the specification is typically much smaller than the implementation.

A procedure must satisfy its specification regardless of the thread calling it. To scale to programs with a large number of threads, we

[4] For simplicity, we do not distinguish between read and write accesses. However, we can easily extend our system to have separate read-access and write-access predicates.

check a procedure for each calling thread separately. In practice, however, procedures seldom behave differently for different calling threads, and we can check a procedure for a single arbitrary thread and conclude that the specification is valid for all possible calling threads.

It is difficult to write a concise specification for a procedure if the environment of the thread executing that procedure can perform arbitrary operations. Access predicates provide a useful mechanism for restricting the environment; we only define abstraction for executions in which all transitions executed by the environment are nice. We call an execution in which all transitions are nice a *nice execution*. In order to define the abstraction relation, we first define two transformations on nice executions.

1. Given $\tau = \omega_0 \xrightarrow{\overline{u}} \omega_1 \xrightarrow{|t,a|} \omega_2 \xrightarrow{|e,b|} \omega_3 \xrightarrow{\overline{w}} \omega$, where $t \neq e$ and $\omega_1 \xrightarrow{|e,b|} \omega_2' \xrightarrow{|t,a|} \omega_3$, we can apply the *reduce* transformation to the execution τ to obtain $\omega_0 \xrightarrow{\overline{u}} \omega_1 \xrightarrow{|e,b|} \omega_2' \xrightarrow{|t,a|} \omega_3 \xrightarrow{\overline{w}} \omega$.
2. Given $\tau = \omega_0 \xrightarrow{\overline{u}} \omega_1 \xrightarrow{|t,\overline{a}|} \omega_2 \xrightarrow{\overline{w}} \omega$ and $\omega_1 \xrightarrow{|t,\overline{a}'|} \omega_2$, we can apply the *t-simulate* transformation to the execution τ to obtain $\omega_0 \xrightarrow{\overline{u}} \omega_1 \xrightarrow{|t,\overline{a}'|} \omega_2 \xrightarrow{\overline{w}} \omega$.

We abstract a nice execution in which thread t is executing the implementation of a procedure by repeatedly applying these two transformations until we get another nice execution in which thread t is executing the specification of that procedure.

More precisely, thread $|t,S|$ is abstracted by the thread $|t,T|$, denoted by $|t,S| \preceq |t,T|$, if for every $|t,S|$-complete and nice execution τ_0, there is a series of nice executions $\tau_1,\ldots,\tau_n$ such that (1) τ_n is a $|t,T|$-complete execution, and (2) for all $0 \le i < n$, the execution τ_{i+1} is obtained from τ_i by either a reduce or a t-simulate transformation. The abstraction relation $\preceq$ preserves the important soundness property that if a program P goes wrong, then the program obtained by abstracting each thread in P will also go wrong.

THEOREM 4.1. *Suppose $|i,S_i|$ and $|i,T_i|$ are nice threads and $|i,S_i| \preceq |i,T_i|$ for all $1 \le i \le n$. If $S_1 \parallel \ldots \parallel S_n$ goes wrong, then $T_1 \parallel \ldots \parallel T_n$ goes wrong.*

The proofs for Theorem 4.1 and the other theorems appear in the appendix.

4.3 Checking procedure specifications

We denote the specification of a procedure m by $\mathcal{S}(m)$, which is a statement without procedure calls. We analyze each procedure separately by inlining the specifications of called procedures. Let the function $\mathit{InlineSpec} : \mathit{Stmt} \to \mathit{Stmt}$ inline the specifications of called procedures in a statement. The following theorem states the soundness of our modular verification methodology. To state the theorem concisely, we extend abstraction to nice statements. A statement S is *nice* if $|t,S|$ is nice for all $t \in \mathit{Tid}$. The statement S is abstracted by the statement T, denoted $S \preceq T$, if $|t,S| \preceq |t,T|$ for all $t \in \mathit{Tid}$.

THEOREM 4.2. *Let P be the program $l_1() \parallel \ldots \parallel l_n()$ and Q be the program $\mathcal{S}(l_1) \parallel \ldots \parallel \mathcal{S}(l_n)$. Suppose $\mathit{InlineSpec}(\mathcal{B}(m))$ and $\mathcal{S}(m)$ are nice statements and $\mathit{InlineSpec}(\mathcal{B}(m)) \preceq \mathcal{S}(m)$ for every procedure $m \in \mathit{Proc}$. If P goes wrong, then Q goes wrong.*

In summary, we verify separately that each procedure is abstracted by its specification after procedure calls have been replaced with specifications of the called procedures. This modular verification enables us to reason about the behavior of the original program by reasoning about its specification.

5 Sequential checkers

We have built a practical verification tool based on Theorem 4.2. Our tool uses an automated theorem prover to check that the statements $\mathit{InlineSpec}(\mathcal{B}(m))$ and $\mathcal{S}(m)$ are nice and $\mathit{InlineSpec}(\mathcal{B}(m)) \preceq \mathcal{S}(m)$ for each procedure m. Each check involves constructing a sequential program and verifying that it does not go wrong. Thus, our analysis can leverage all existing techniques for verifying sequential programs. In particular, our tool Calvin constructs a verification condition [9, 19] for each check and then checks the verification condition with the automatic theorem prover Simplify [28].

Since $\mathit{InlineSpec}(\mathcal{B}(m))$ and $\mathcal{S}(m)$ do not contain procedure calls, we need sequential checkers to verify that: (1) a statement S without procedure calls is nice, and (2) $S \preceq T$ for statements S and T without procedure calls. In the next two subsections, we describe these two checkers.

5.1 Checker for nice threads

In this section, we construct a statement $\mathit{CN}(S)$ (Check Nice) with the property that if $|t, \mathit{CN}(S)|$ does not go wrong then $|t,S|$ is nice. To show that $|t,S|$ is nice, we must show that it makes no non-nice transitions when executed in parallel with other nice threads. We wish to check this property modularly, with no knowledge of other threads except that they are nice. Thus, we construct a sequential program based on S in which actions modeling all possible nice transitions of other threads are interleaved between the steps of $|t,S|$.

If a thread performs a nice transition, it must satisfy the following predicate:

$$Z \stackrel{\text{def}}{=} \bigwedge_{\mathtt{x} \in \mathit{Var}} (A_{\mathtt{x}} \vee \mathtt{x} = \mathtt{x}'),$$

The action EA, defined below, when executed by thread t, can perform all possible nice transitions by any thread other than t.

$$\mathrm{EA}(t,\sigma,\sigma') \stackrel{\text{def}}{=} \exists j \in \mathit{Tid}.\ j \neq t \wedge Z(j,\sigma,\sigma')$$

For any atomic operation a, we use EA to construct the statement $\mathit{CN}(a)$ with the property that the thread $|t,\mathit{CN}(a)|$ goes wrong only if $|t,a|$ is not nice.

Figure 9: Is $a = p?X$ nice?

$$\begin{array}{lll} \mathit{CN}(a) & \stackrel{\text{def}}{=} & \mathrm{EA}^*; \\ & & \mathtt{assert}\ \mathit{enabled}(a) \Rightarrow A_a; \\ & & \mathtt{assume}\ p; \\ & & X; \end{array}$$

This statement models an arbitrary number of environment steps with the statement EA^* and then asserts that every transition allowed by the operation a is nice. The operation a will only execute when it does not block in the current state. It will not block in state σ when the predicate $enabled(a)(t,\sigma)$ is true:

$$enabled(p?X)(t,\sigma) \quad \stackrel{\text{def}}{=} \quad p(t,\sigma) \Rightarrow \exists\sigma'.\, X(t,\sigma,\sigma').$$

Finally, the statement `assume` p followed by X allows execution to proceed only if a does not go wrong.

To check that $|t,S|$ is nice, we construct the sequential program $CN(S)$ by replacing every atomic operation a in S with $CN(a)$.

THEOREM 5.1. *Let S be a statement without procedure calls. For all $t \in Tid$, if $|t, CN(S)|$ does not go wrong, then $|t,S|$ is nice.*

5.2 Checker for abstraction

We now construct a statement $CA(S,T,W)$ (Check Abstraction) with the property that for all $t \in Tid$, if $|t, CA(S,T,W)|$ does not go wrong then $|t,S| \preceq |t,T|$. W provides the witness for the simulation checker. The witness is defined below and is similar to the witness used in [18].

The atomic operations in the statement T are restricted to be actions, *i.e.*, atomic operations of the form `true`$?X$. We have found such specifications to be sufficient for checking Daisy and a variety of other programs [18]. We could handle more general T at the cost of extra complexity in the construction of $CA(S,T,W)$.

For ease of implementation, we perform two simplifications in our checking algorithm. First, while abstracting any path of $|t,S|$, we only apply reduce transformations followed by t-simulate transformations. Although our checker does not support arbitrary combinations of these transformations, it has been able to check all programs we have encountered. Second, when performing a t-simulate transformation to simulate a path $\overline{a}$ with a path $\overline{a}'$, we require that $\overline{a}'$ be at most one step long, i.e. $\overline{a}'$ is empty or a single action.

The checking algorithm embedded in $CA(S,T,W)$ partitions any path $\overline{a}$ of S into disjoint paths $\overline{a}_1,\ldots,\overline{a}_n$, each consisting of a sequence of right movers followed by a single atomic operation followed by a sequence of left movers. Using a sequence of reduce transformations, the atomic operations in each path $\overline{a}_i$ can be brought adjacent to each other. Our method then uses simulation to check that there is a path $b_1,\ldots,b_n$ such that every execution of $\overline{a}_i$ is allowed by b_i, where $b_1,\ldots,b_n$ is obtained from a path of T by inserting zero or more `skip` actions between consecutive actions in the path. The `skip` actions handle t-simulate transformations that require zero steps to be taken by the specification. Thus, a sequence of reduce transformations followed by t-simulate transformations yields the abstraction $b_1,\ldots,b_n$ for $\overline{a}_1,\ldots,\overline{a}_n$.

In general, it is a difficult problem to discover the path $b_1,\ldots,b_n$ given only the path $\overline{a}_1,\ldots,\overline{a}_n$ in S. Therefore, the programmer indicates the correct path by providing annotations in S and T. First, the programmer annotates each action in T by a unique label. For example, the specification $(l_1 : c_1\ ;\ l_2 : c_2)^*\ ;\ l_3 : c_3$ labels each c_i with l_i. The action $Act_T(l)$ is the action in T labeled l, and $Lab(T)$ is the set of all labels in T. Calvin-R also attaches a unique label to each atomic operation in S. Our checker uses the edge relation $lr(T)$ of the control flow graph of T:

$$lr(T) \subseteq (Lab(T) \cup \{\texttt{begin}\}) \times (Lab(T) \cup \{\texttt{end}\})$$

The definition of $lr(T)$ uses the distinct labels `begin` and `end` to represent the entry and exit labels for T. The relation $lr(T)$ can be easily computed based on the structure of T.

The witness W provides the mapping for the abstraction checker between steps taken by S in an execution and steps taken by T. In our framework, the witness W is a map from the labels in S to expressions over the variables in Var and a special variable `pc`. For each label $l \in Lab(S)$, the expression $W(l)$ evaluates (given a valuation for $Var \cup \{\texttt{pc}\}$), to a label in $Lab(T) \cup \{\texttt{begin}, \texttt{end}\}$. The variable `pc` is a fresh variable introduced in $CA(S,T,W)$ to track the label of the the witnessing action in T. The variable `pc` is initialized to `begin`, and updated before every action in S labeled l to the label obtained by evaluating the expression $W(l)$.

In practice, the programmer specifies W by inserting annotations into S, as illustrated in Figure 3. Calvin-R uses these witness annotations in S to construct the witness expressions as follows. If a witness annotation

`//@ witness` e

is present just before a statement labeled l, then $W(l) = e$. If no witness annotation is present, then $W(l) = \texttt{pc}$. For example, in the program above, if l is the label for `return i`, then $W(l) =$ `"Allocated"`. If l were instead the label for `bits[i] = true`, then $W(l) = \texttt{pc}$.

After partitioning $\overline{a}$ into $\overline{a}_1,\ldots,\overline{a}_n$, the checker examines each $\overline{a}_i$ in turn. Let $l_0 = \texttt{begin}$ be the initial value of `pc` and l_i the value of `pc` at the end of the execution of $\overline{a}_i$. If $l_i \neq l_{i-1}$, then the simulating action b_i is deemed to be the action $Act_T(l_i)$ of T. If $l_i = l_{i-1}$, then the simulating action b_i can be either `skip` or $Act_T(l_i)$. $CA(S,T,W)$ then verifies that $b_1,\ldots,b_n$ simulates $\overline{a}_1,\ldots,\overline{a}_n$ by performing the following checks:

(C1) For each i, either the composition of atomic operations in $\overline{a}_i$ is simulated by the action b_i and $(l_{i-1}, l_i) \in lr(T)$ or the composition is simulated by `skip` and $l_i = l_{i-1}$.

(C2) The statement T can terminate after b_n, *i.e.*, $(l_n, \texttt{end}) \in lr(T)$.

The conditions (C1) and (C2) ensure that the sequence obtained by removing the `skip` actions from $b_1,\ldots,b_n$ is a path of T. To check (C1), the statement $CA(S,T,W)$ performs the following steps:

1. Record the current store of all variables and `pc`. The assignment $\overrightarrow{\texttt{x}_{\text{orig}} \texttt{ = x}}$ stores each variable `x` in $\texttt{x}_{\text{orig}}$.
2. Execute steps from S as long as the sequence executed is reducible, i.e., it is a sequence of right movers, followed by a single atomic operation, followed by left movers. At each step, `pc` is updated using the witness expression.
3. Check (C1) by ensuring that the action $\alpha(T)$, defined below, allows the transition from the store recorded in step 1 to the current store.

$$\alpha(T) \stackrel{\text{def}}{=} \left(\begin{array}{l} \texttt{skip} \\ \wedge\ \texttt{pc} = \texttt{pc}' \end{array}\right) \vee \left(\begin{array}{l} Act_T(\texttt{pc}') \\ \wedge\ (\texttt{pc}, \texttt{pc}') \in lr(T) \end{array}\right)$$

   The action $\alpha(T)$ requires that the reduced sequence of steps in S be simulated either by the stutter step `skip` or by the action in T labeled with the final value of `pc`. This check can be easily performed by asserting the predicate $\rho(T)$ obtained by substituting $\texttt{x}_{\text{orig}}$ for `x` and `x` for `x'` in the action $\alpha(T)$.

$$\rho(T) = \alpha(T)[\overrightarrow{\texttt{x} := \texttt{x}_{\text{orig}}}, \overrightarrow{\texttt{x}' := \texttt{x}}, \texttt{pc} := \texttt{pc}_{\text{orig}}, \texttt{pc}' := \texttt{pc}]$$

4. Model steps taken by other threads by executing EA^*.

The program repeats these steps until all of S has been executed.

At that point, condition (C2) is verified for the final state of the execution.

The most subtle part of the algorithm described informally in the steps 1-4 above is the deduction of a reducible sequence of actions in S by inferring right and left movers. We now give the details of how $CA(S, T, W)$ infers right and left movers. We start by constructing the *exclusive access predicate* $E_{\mathtt{x}}$ from the access predicate $A_{\mathtt{x}}$ as

$$E_{\mathtt{x}}(t,\sigma) \stackrel{\text{def}}{=} \forall j \in \mathit{Tid}.\, j \neq t \Rightarrow \neg A_{\mathtt{x}}(j,\sigma).$$

During the execution of a nice program, if $E_{\mathtt{x}}(t,\sigma)$ holds, then any action from σ by a thread other than t will not access the variable `x`. As with access predicates, the exclusive access predicate for an atomic operation a, written E_a, is the conjunction of exclusive access predicates for all variables accessed by a. If $E_a(t,\sigma)$ holds immediately after a step $|t,a|$, then a following operation b by a different thread e cannot access any variables accessed by a. Therefore, the steps $|t,a|$ and $|e,b|$ can be commuted, and the step $|t,a|$ is a right mover. Similarly, if $E_a(t,\sigma)$ holds immediately before a step $|t,a|$, then that step is a left mover.

To construct $CA(S, T, W)$, we first transform the statement S into the statement $CODE(S, T, W)$ by replacing every atomic operation $l : a$ in S by the statement $CODE(l : a, T, W)$ (defined below). We then put extra checking code both before and after the statement $CODE(S, T, W)$ to obtain $CA(S, T, W)$.

Figure 10: Checker for abstraction

$$
\begin{array}{lll}
CODE(l:a,T,W) & \stackrel{\text{def}}{=} & \texttt{if}\ (\texttt{phase} == \mathrm{RM})\ \{ \\
 & & \quad a; \\
 & & \quad \texttt{if}\ (!E_a)\ \texttt{phase} = \mathrm{LM}; \\
 & & \}\ \texttt{else}\ \texttt{if}\ (E_a \wedge \mathit{enabled}(a))\ \{ \\
 & & \quad a; \\
 & & \}\ \texttt{else}\ \{ \\
 & & \quad \texttt{assert}\ \rho(T)\ ; \\
 & & \quad \mathrm{EA}^*\ ; \\
 & & \quad \overrightarrow{\texttt{x}_{\text{orig}}\ \texttt{=}\ \texttt{x}}\ ; \\
 & & \quad \texttt{pc}_{\text{orig}}\ \texttt{=}\ \texttt{pc} \\
 & & \quad a; \\
 & & \quad \texttt{if}\ (E_a)\ \texttt{phase} = \mathrm{RM}; \\
 & & \} \\
 & & \texttt{pc = }\ W(l)\ ; \\
 & & \\
CA(S,T,W) & \stackrel{\text{def}}{=} & \texttt{pc = begin}\ ; \\
 & & \overrightarrow{\texttt{x}_{\text{orig}}\ \texttt{=}\ \texttt{x}}\ ; \\
 & & \texttt{pc}_{\text{orig}}\ \texttt{= pc}\ ; \\
 & & CODE(S,T,W)\ ; \\
 & & \texttt{assert}\ \rho(T)\ ; \\
 & & \texttt{assert}\ (\texttt{pc}, \texttt{end}) \in \mathit{lr}(T)
\end{array}
$$

Our construction of $CODE(l : a, T, W)$ uses the variable `phase` to record which type of steps —right movers or left movers— the checker program is matching. The possible phases are RM (matching right movers) and LM (matching left movers).

An action is not a left mover if it may block, because commuting a blocking operation to a position earlier in the trace may prevent the environment from executing actions (and going wrong) from a store where it previously could have.

The statement $CA(S, T, W)$ initializes `pc`, $\overrightarrow{\texttt{x}_{\text{orig}}}$, and $\texttt{pc}_{\text{orig}}$, executes $CODE(S, T, W)$, and checks condition (C1) for the last time by asserting $\rho(T)$. It checks condition (C2) by asserting that the final value of `pc` allows T to terminate.

THEOREM 5.2. *Let S be a statement without procedure calls. Let T be a statement without procedure calls whose atomic operations are actions. Let W be a witness expression. For all $t \in \mathit{Tid}$, if $|t, CA(S, T, W)|$ does not go wrong, then $|t, S| \preceq |t, T|$.*

5.3 Implementation

We have implemented the algorithms from Section 5.1 and 5.2 in Calvin-R, an automatic checking tool for multithreaded Java programs. The Calvin-R implementation leverages ESC/Java [15]. In previous papers, we have described Calvin-R's thread-modular checking algorithm based on assume-guarantee decomposition [14] and extensions to check procedure specifications based on simulation [18].

Calvin-R takes as input a Java program annotated with procedure specifications, access predicates, and assertions. Figures 1 and 3 show the `accessible_if` and `performs` annotations used in Java programs. A `performs` statement is built by combining actions with the operators "`[]`" for choice, "`;`" for sequential composition, and "`*`" for iteration.

To check a procedure specification, Calvin-R inlines the specifications of all calls in the procedure. If a specification for a procedure is not provided, its implementation is inlined instead. Calvin-R then constructs the sequential programs described earlier. Each sequential program is translated into a verification condition [19] and checked with the Simplify automatic theorem prover [28]. If the theorem prover finds the verification condition to be valid, the procedure conforms to its specification, satisfies the access predicates, and satisfies all the assertions. Otherwise, the theorem prover returns a counter example demonstrating why the verification condition is invalid. Calvin-R processes the counter example to construct a meaningful error message for the user.

6 Daisy file system

To validate our methodology and algorithms, we have applied our prototype to Daisy, a file system we designed for our first case study. Since the focus of our work is concurrency, we designed Daisy to use synchronization mechanisms similar in complexity to those found in other file systems. However, the data structures and algorithms in Daisy are relatively simple, allowing us to implement it in approximately 1200 lines of Java code. In Daisy, each inode, block, and directory is guarded by a separate mutual exclusion lock, and the file system metadata is read and written from disk on demand.

To minimize the impact of our language choice on the evaluation of the Calvin-R checker, we avoided the use of most Java-specific features, such as inheritance, exceptions, extraneous library classes, and object constructors. Daisy supports a subset of the POSIX file system interface and can be mounted as an NFS server.

We have checked that all procedures in Daisy respect access predicates describing the program's synchronization discipline. This analysis required 30 lines of annotations in the source code. In addition, we added roughly 150 lines of procedure specifications in order to check the functional correctness of all low-level procedures, such as `alloc`, and some of the most complex top-level procedures, such as `create`. Calvin-R performs all of these checks in only a few minutes of CPU time on a desktop workstation.

The `create` procedure is the most complex procedure verified for correctness. It reads a directory structure from disk, allocates a new inode and block, and updates the directory to contain the new file, performing all of the appropriate error checking and recovery in the event of a failure. We added 15 lines of annotations to specify this functionality for `create`.

Two unknown bugs in Daisy found during this exercise exhibit the potential of our approach. The first error occurred in the routine to write directory structures to disk. This procedure would not properly release a lock if a disk write were to fail in one specific situation. Testing did not uncover the problem because we never caused the write error that triggered this behavior.

Checking Daisy with Calvin-R also identified a bug in `create` that was introduced while changing the the way Daisy creates new files. Directory structures contain a fixed-size array of inode numbers and a length field indicating how much of the array is in use. We introduced an error that caused `create` to incorrectly increment the length field without changing the array if it was unable to allocate a free inode for the new file.

The errors we found in Daisy are similar to those found in real file systems only after extensive and costly testing [37].

7 Related work

Lipton [26] first proposed reduction as a way to reason about concurrent programs without considering all possible interleavings. He focused primarily on checking deadlock freedom. Doeppner [10], Back [2], and Lamport and Schneider [25] extended this work to allow proofs of general safety properties. Misra [27] has proposed a reduction theorem for programs built with monitors [23] communicating via procedure calls. Cohen and Lamport [7] have extended reduction to allow proofs of liveness properties. All of these papers have focused on the theory of reduction. However, they do not describe a methodology for verifying programs. We go beyond their work by developing a specification and verification methodology for a widely used programming language.

Partial-order methods [21, 29] have been used to limit state-space explosion while model checking concurrent programs. These methods identify sequences of interleaved steps for which the property being checked is insensitive to the exact ordering. A single representative interleaving of the operations is then explored. These methods have mostly been applied to systems of processes communicating through message passing. Verisoft [20] is one such tool. While these methods are typically unable to reorder accesses to shared variables, Calvin-R uses access predicates to determine when it is safe to reorder accesses to shared variables as well. Dwyer et al. extended partial order reduction to take advantage of object escape and locking information [11].

Using ideas from reduction and partial-order methods, Bruening [4] has built an assertion checker based on state-space exploration for multithreaded Java programs. His tool requires another checker to ensure the absence of races. This assumption allows `synchronized` code blocks to be treated as atomic. Stoller [35] provides a generalization of Verisoft and Bruening's method to allow model checking of programs with either message-passing or shared-memory communication. Both of these approaches are restricted to mutex-based synchronization and operate on the concrete program without any abstraction. In our work, access predicates provide a more general mechanism for specifying synchronization. More recently, Stoller and Cohen have adopted access predicates in order to capture a richer set of synchronization idioms and to perform reduction during model checking [36].

Insights gained while developing Calvin-R have recently led us to develop several light-weight verification tools that focus on identifying procedures in multithreaded programs that may be considered to execute atomically. Flanagan and Qadeer use reduction in a syntactic type-based analysis to identify atomic procedures [17, 16]. The type system scales more easily because it requires fewer and less complex program annotations, but it is limited to checking this single atomicity property. Flanagan and Freund have implemented the Atomizer tool [13], which uses reduction to identify atomic blocks of code dynamically. Although the Atomizer is unsound because it does not check all execution paths, it requires little or no programmer-supplied information and has identified atomicity errors in a number of large systems. In other work, the notions of reduction and atomicity are used by Qadeer et al. [30] to infer concise procedure summaries in an analysis for multithreaded programs.

Hatcliff et al. [22] explore an alternative approach for verifying atomicity using model-checking. In addition to using Lipton's theory of reduction, they also investigate an approach based on partial order reductions. Their experimental results suggest that the model-checking approach for verifying atomicity is feasible for unit-testing, where the reachable state space is smaller than in integration-testing. In related work, Robby et al. [32] demonstrate how to check some properties of multithreaded code with standard pre- and post-conditions by refactoring code into methods that do not update global state from critical sections nested inside methods.

A number of static tools have been designed to detect data races. These include several type systems [12, 3], Warlock [34], and a race detector for SPMD programs [1]. Dynamic race detection tools [33, 5] require very few annotations, if any, but may fail to detect some errors due to insufficient coverage. Several tools combining dynamic and static analyses have recently been proposed [38, 6]. Access predicates provide a simple, general method for specifying and verifying a wider variety of synchronization mechanisms than allowed by these tools.

Several tools [8, 39] verify safety properties using a combination of data abstraction and model checking. These tools consider all possible thread interleavings while performing state exploration. The approach in this paper can be used to abstract a program, thereby reducing the possible interleavings. Invariant checking can then be performed on the abstract program thus improving the efficiency of these techniques.

8 Conclusions

Calvin-R provides a way for programmers to specify and check complex properties of multithreaded programs in a simple, intuitive way. The access predicates used by Calvin-R enable synchronization mechanisms to be expressed in a simple and uniform way. In addition, the abstraction relation based on simulation and reduction permits both data and control abstraction in method specifications. Many properties of multithreaded code can be expressed concisely and checked with this technique.

The next step is to validate this methodology further by showing that it can scale to larger programs. Some issues to address are how to best map errors in the generated sequential program back to errors in the original, multithreaded program; how to reduce annotation overhead by automatically inferring some annotations, such as simple access predicates for variables guarded by mutual exclusion locks; and how to ensure that our translation does not produce sequential programs too complex for the underlying theorem prover to handle.

Acknowledgments: We thank Cormac Flanagan for the initial design of Daisy and for useful discussions and feedback on this paper. We also thank Sanjit Seshia for useful discussions, and Mark Lillibridge for comments on a draft of this paper. Stephen Freund's work was supported in part by faculty research funds granted by Williams College and by the National Science Foundation under Grant CCR-0341387.

9 References

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

[2] R.-J. Back. A method for refining atomicity in parallel algorithms. In *PARLE 89: Parallel Architectures and Languages Europe*, volume 366 of *Lecture Notes in Computer Science*, pages 199–216. Springer-Verlag, 1989.

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

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

[5] G. Cheng, M. Feng, C. Leiserson, K. Randall, and A. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 298–309, 1998.

[6] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridhara. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 258–269, 2002.

[7] E. Cohen and L. Lamport. Reduction in TLA. In *International Conference on Concurrency Theory*, pages 317–331, 1998.

[8] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[9] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[10] T. W. Doeppner, Jr. Parallel program correctness through refinement. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 155–169, 1977.

[11] M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Exploiting object escape and locking information in partial order reduction for concurrent object-oriented programs. *Formal Methods in System Design Journal*, 2004. (to appear).

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

[13] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, 2004.

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

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

[16] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[17] C. Flanagan and S. Qadeer. Types for atomicity. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*, pages 1–12, 2003.

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

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

[20] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[21] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, pages 406–415. IEEE Computer Society Press, 1991.

[22] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*, 2004.

[23] C. Hoare. Monitors: an operating systems structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

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

[25] L. Lamport and F. B. Schneider. Pretending atomicity. Research Report 44, DEC Systems Research Center, 1989.

[26] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[27] J. Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Springer-Verlag, 2001.

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

[29] D. Peled. Combining partial order reductions with on-the-fly model checking. In D. Dill, editor, *CAV 94: Computer Aided Verification*, Lecture Notes in Computer Science 818, pages 377–390. Springer-Verlag, 1994.

[30] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, 2004.

[31] M. C. Rinard. Analysis of multithreaded programs. In *SAS 01: Static Analysis Symposium*, Lecture Notes in Computer Science 2126, pages 1–19. Springer-Verlag, 2001.

[32] Robby, E. Rodriguez, M. B. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model checking framework. Technical Report SAnToS-TR-2003-10, Kansas State University, 2004.

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

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

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

[36] S. D. Stoller and E. Cohen. Optimistic synchronization-based state-space reduction. In H. Garavel and J. Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 489–504. Springer-Verlag, 2003.

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

[38] C. von Praun and T. Gross. Object race detection. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 70–82, 2001.

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

A Proofs for Section 4

In this section, we present proofs for Theorems 1 and 2. We begin with a formal definition of nice threads. An execution τ is t-nice if for every prefix $\omega_1 \xrightarrow{\overline{u}} \omega_2 \xrightarrow{|t,a|} \omega_3$ of τ, the transition $\omega_2 \xrightarrow{|t,a|} \omega_3$ is nice whenever $\omega_1 \xrightarrow{\overline{u}} \omega_2$ is nice. A thread $|t,S|$ is nice if every $|t,S|$-complete execution is t-nice. The following fact can be proved easily:

FACT 1. *Suppose $\omega_1 \xrightarrow{\overline{u}} \omega_2 \xrightarrow{\overline{v}} \omega_3$ is an execution and the following hold:*

1. *$\omega_1 \xrightarrow{\overline{u}} \omega_2$ is t-nice.*
2. *If $\omega_1 \xrightarrow{\overline{u}} \omega_2$ is nice, then $\omega_1 \xrightarrow{\overline{u}} \omega_2 \xrightarrow{\overline{v}} \omega_3$ is t-nice.*

Then $\omega_1 \xrightarrow{\overline{u}} \omega_2 \xrightarrow{\overline{v}} \omega_3$ is a nice execution.

We now turn our attention to Theorem 4.1 and prove that if a program does not go wrong, then any program constructed by abstracting each thread of the original program does not go wrong.

RESTATEMENT OF THEOREM 4.1. Suppose $|i,S_i|$ and $|i,T_i|$ are nice threads and $|i,S_i| \preceq |i,T_i|$ for all $1 \le i \le n$. If $S_1 \parallel \ldots \parallel S_n$ goes wrong, then $T_1 \parallel \ldots \parallel T_n$ goes wrong.

PROOF. Let $P = S_1 \parallel \ldots \parallel S_n$ and $Q = T_1 \parallel \ldots \parallel T_n$. For any execution of P, we show that there is an execution of Q with the same final state, thus proving that the program Q goes wrong if P goes wrong. We proceed by proving the following statement by induction on j:

> For any $0 \le j \le n$ and for any execution τ of the program $Q_j = S_1 \parallel \ldots \parallel S_j \parallel T_{j+1} \parallel \ldots \parallel T_n$, there is an execution of Q with the same final state.

Base Case: If $j = 0$, the inductive hypothesis is trivially satisfied.

Inductive Step: We show that if τ is an execution of Q_j, there is an execution of Q_{j-1} with the same final state as τ. The inductive hypothesis then allows to conclude that there is an execution of Q with the same final state.

Since each thread in Q_j is nice, the execution τ is nice. In addition, we know that $|j,S_j| \preceq |j,T_j|$. Therefore, there is a series of nice executions $\tau_0, \ldots, \tau_k$ such that $\tau_0 = \tau$, τ_k is an execution of Q_{j-1} and each τ_{i+1} is obtained from τ_i by either a reduce or j-simulate transformation. Since both transformations preserve the final state of the execution, the executions $\tau_0, \ldots, \tau_k$ all end in the same final state. □

Before proving Theorem 4.2, we state and prove the following lemma relating procedure implementations to their specifications.

LEMMA 1. *Suppose the statement $\mathit{InlineSpec}(\mathcal{B}(m))$ is nice and $\mathit{InlineSpec}(\mathcal{B}(m)) \preceq \mathcal{S}(m)$ for all procedures $m \in \mathit{Proc}$. Then $m()$ is nice and $m() \preceq \mathcal{S}(m)$ for all procedures $m \in \mathit{Proc}$.*

PROOF. We prove that $[\![m()]\!]^d$ is nice and $[\![m()]\!]^d \preceq \mathcal{S}(m)$ by induction on d:

Base case: If $d = 0$, then $[\![m()]\!]^d = \texttt{assume false}$. Thus, $[\![m()]\!]^d$ is nice because it has no executions, and $[\![m()]\!]^d \preceq \mathcal{S}(m)$.

Inductive step: If $d > 0$, then $[\![m()]\!]^d = [\![\mathcal{B}(m)]\!]^{d-1}$. We show that $|t,[\![m()]\!]^d|$ is nice and $|t,[\![m()]\!]^d| \preceq |t,\mathcal{S}(m)|$ for all $t \in \mathit{Tid}$. Suppose τ is a $|t,[\![m()]\!]^d|$-complete execution corresponding to the path $\overline{u}$. The path $\overline{u}$ must have the form $\overline{u}_1;\overline{v}_1 \ldots \overline{u}_{k-1};\overline{v}_{k-1};\overline{u}_k$, where each $\overline{v}_i$ represents execution inside a procedure call $m_i()$ inside the body of m. Let $|t,\overline{a}_i|$ be the projection of the path $\overline{u}_i$ onto the steps by thread t. The execution τ has the form $\omega_1 \xrightarrow{\overline{u}_1} \omega_1' \xrightarrow{\overline{v}_1} \omega_2 \ldots \omega_{k-1}' \xrightarrow{\overline{v}_{k-1}} \omega_k \xrightarrow{\overline{u}_k} \omega_k'$, where $\omega_i' \xrightarrow{\overline{v}_i} \omega_{i+1}$ is a $|t,[\![m_i()]\!]^{d-1}|$-complete execution for all $1 \le i < k$. By the inductive hypothesis, we know that the following are true for all $1 \le i < k$:

- the execution $\omega_i' \xrightarrow{\overline{v}_i} \omega_{i+1}$ is t-nice, and
- if the execution $\omega_i' \xrightarrow{\overline{v}_i} \omega_{i+1}$ is nice, then there is a $|t,\mathcal{S}(m_i)|$-complete execution $\omega_i' \xrightarrow{\overline{w}_i} \omega_{i+1}$ obtained from $\omega_i' \xrightarrow{\overline{v}_i} \omega_{i+1}$ by a sequence of reduce and t-simulate transformations.

In addition, since $|t,\mathit{InlineSpec}(\mathcal{B}(m))|$ is nice, any $|t,\overline{a}_1;\mathcal{S}(m_1);\ldots;\overline{a}_{k-1};\mathcal{S}(m_{k-1});\overline{a}_k|$-complete execution (as well as any prefix of such an execution) is t-nice.

Let the path $\overline{x}_i = \overline{u}_1;\overline{v}_1;\ldots\overline{u}_{i-1};\overline{v}_{i-1}$. We now prove the following property by induction on i:

> For all $1 \le i \le k$,
>
> 1. if the execution $\omega_1 \xrightarrow{\overline{x}_i} \omega_i$ is nice, there is a nice and $|t,\overline{a}_1;\mathcal{S}(m_1);\ldots;a_{i-1};\mathcal{S}(m_{i-1})|$-complete execution $\omega_1 \xrightarrow{\overline{y}_i} \omega_i$ obtained from $\omega_1 \xrightarrow{\overline{x}_i} \omega_i$ by a sequence of reduce and t-simulate transformations, and
> 2. the execution $\omega_1 \xrightarrow{\overline{x}_i;\overline{u}_i} \omega_i'$ is t-nice.

In the base case, $i = 1$ and we need only show that $\omega_1 \xrightarrow{\overline{u}_1} \omega_1'$ is t-nice. Since $\omega_1 \xrightarrow{\overline{u}_1} \omega_1'$ is a prefix of a $|t,\overline{a}_1;\mathcal{S}(m_1);\ldots;\overline{a}_{k-1};\mathcal{S}(m_{k-1});\overline{a}_k|$-complete execution, it is t-nice.

We now show that the two conditions hold in the inductive step:

1. Suppose $\omega_1 \xrightarrow{\overline{x}_i} \omega_i \xrightarrow{\overline{u}_i} \omega'_i \xrightarrow{\overline{v}_i} \omega_{i+1}$ is nice. Then $\omega_1 \xrightarrow{\overline{x}_i} \omega_i$, $\omega_i \xrightarrow{\overline{u}_i} \omega'_i$, and $\omega'_i \xrightarrow{\overline{v}_i} \omega_{i+1}$ are all nice. By the inductive hypothesis, there is a nice and $|t,\overline{a}_1;\mathcal{S}(m_1);\ldots;a_{i-1};\mathcal{S}(m_{i-1})|$-complete execution $\omega_1 \xrightarrow{\overline{y}_i} \omega_i$ obtained from $\omega_1 \xrightarrow{\overline{x}_i} \omega_i$ by a sequence of reduce and t-simulate transformations. We also know that there is a nice and $|t,\mathcal{S}(m_i)|$-complete execution $\omega'_i \xrightarrow{\overline{w}_i} \omega_{i+1}$ obtained from $\omega'_i \xrightarrow{\overline{v}_i} \omega_{i+1}$ by a sequence of reduce and t-simulate transformations. We combine the two sequences of transformations to get the nice and $|t,\overline{a}_1;\mathcal{S}(m_1);\ldots;a_i;\mathcal{S}(m_i)|$-complete execution $\omega_1 \xrightarrow{\overline{x}_{i+1}} \omega_{i+1}$.

2. Suppose $\omega_1 \xrightarrow{\overline{x}_i;\overline{u}_i} \omega'_i$ is t-nice. We know that $\omega'_i \xrightarrow{\overline{v}_i} \omega_{i+1}$ is t-nice, which implies that $\omega_1 \xrightarrow{\overline{x}_{i+1}} \omega_{i+1}$ is t-nice. We now prove that if $\omega_1 \xrightarrow{\overline{x}_{i+1}} \omega_{i+1}$ is nice, then $\omega_1 \xrightarrow{\overline{x}_{i+1};\overline{u}_{i+1}} \omega'_{i+1}$ is t-nice. Once we have shown this, Fact 1 allows us to conclude that $\omega_1 \xrightarrow{\overline{x}_{i+1};\overline{u}_{i+1}} \omega'_{i+1}$ is t-nice. If $\omega_1 \xrightarrow{\overline{x}_{i+1}} \omega_{i+1}$ is nice, then there is a path $\overline{y}_{i+1}$ such that $\omega_1 \xrightarrow{\overline{y}_{i+1}} \omega_{i+1}$ is a nice and $|t,\overline{a}_1;\mathcal{S}(m_1);\ldots;a_i;\mathcal{S}(m_i)|$-complete execution. Therefore, $\omega_1 \xrightarrow{\overline{x}_{i+1};\overline{u}_{i+1}} \omega'_{i+1}$ is a $|t,\overline{a}_1;\mathcal{S}(m_1);\ldots;a_i;\mathcal{S}(m_i);a_{i+1}|$-complete execution. Since this execution is the prefix of another execution that is $|t,\overline{a}_1;\mathcal{S}(m_1);\ldots;\overline{a}_{k-1};\mathcal{S}(m_{k-1});\overline{a}_k|$-complete, it is t-nice.

We now return to $\overline{u}$. Since $\overline{u} = \overline{x}_k;\overline{u}_k$, the property we just proved allows us to conclude that $\omega_1 \xrightarrow{\overline{u}} \omega'_k$ is t-nice. Also, if the execution $\omega_1 \xrightarrow{\overline{x}_k} \omega_k$ is nice, there is a nice and $|t,\overline{a}_1;\mathcal{S}(m_1);\ldots;a_{k-1};\mathcal{S}(m_{k-1})|$-complete execution $\omega_1 \xrightarrow{\overline{y}_k} \omega_k$ obtained from $\omega_1 \xrightarrow{\overline{x}_k} \omega_k$ by a sequence of reduce and t-simulate transformations. Therefore, we can obtain a nice and $|t,\overline{a}_1;\mathcal{S}(m_1);\ldots;a_{k-1};\mathcal{S}(m_{k-1});a_k|$-complete execution $\omega_1 \xrightarrow{\overline{y}_k;u_k} \omega_k$ by a sequence of reduce and t-simulate transformations from the execution $\omega_1 \xrightarrow{\overline{u}} \omega_k$. Since $|t, \mathit{InlineSpec}(\mathcal{B}(m))| \preceq |t,\mathcal{S}(m)|$, we can perform further reduce and t-simulate transformations on $\omega_1 \xrightarrow{\overline{y}_k;u_k} \omega_k$ to obtain a nice and $|t,\mathcal{S}(m)|$-complete execution. □

We use the previous lemma prove that procedure specifications will go wrong if procedure implementations with inlined specifications go wrong, as stated in Theorem 4.2.

RESTATEMENT OF THEOREM 4.2. Let P be the program $l_1() \parallel\ldots\parallel l_n()$ and Q be the program $\mathcal{S}(l_1) \parallel\ldots\parallel \mathcal{S}(l_n)$. Suppose $\mathit{InlineSpec}(\mathcal{B}(m))$ and $\mathcal{S}(m)$ are nice statements and $\mathit{InlineSpec}(\mathcal{B}(m)) \preceq \mathcal{S}(m)$ for every procedure $m \in \mathit{Proc}$. If P goes wrong, then Q goes wrong.

PROOF. From Lemma 1, we get that $m()$ is nice and $m() \preceq \mathcal{S}(m)$ for all $m \in \mathit{Proc}$. Thus we have $|i,l_i()| \preceq |i,\mathcal{S}(l_i)|$ and $|i,l_i()|$ and $|i,\mathcal{S}(l_i)|$ are nice for all $1 \le i \le n$. From Theorem 4.1, we get that Q will go wrong if P goes wrong. □

B Proofs for CN

The following lemma states that the program $\mathit{CN}(a)$ can be used to test whether a transition made by a is nice.

LEMMA 2. *Suppose $\sigma \xrightarrow{\overline{u}} \sigma'$ is a nice execution such that no step in $\overline{u}$ is a t-step and a is an atomic operation. Then, there is $\overline{v} \in [\![\mathit{CN}(a)]\!]$ such that:*

1. *If $\sigma' \xrightarrow{|t,a|} \omega$ is nice and $\omega \neq \mathbf{wrong}$, then $\sigma \xrightarrow{|t,\overline{v}|} \omega$.*

2. *If $\sigma' \xrightarrow{|t,a|} \omega$ is not nice, then $\sigma \xrightarrow{|t,\overline{v}|} \mathbf{wrong}$.*

PROOF. Let

$$\begin{array}{rcl} a & = & p?X \\ u & = & |e_1,b_1|;\ldots;|e_n,b_n| \\ \mathrm{EA}^n & = & \underbrace{\mathrm{EA};\ldots;\mathrm{EA}}_{n \text{ times}} \\ \overline{v} & = & \mathrm{EA}^n;\texttt{assert}\ \mathit{enabled}(a) \Rightarrow A_a;\texttt{assume}\ p;X \end{array}$$

Clearly $\overline{v} \in [\![\mathit{CN}(a)]\!]$. Suppose $\sigma = \sigma_1 \xrightarrow{|e_1,b_1|} \sigma_2 \cdots \sigma_n \xrightarrow{|e_n,b_n|} \sigma_{n+1} = \sigma'$. For all $1 \le i \le n$, since $e_i \neq t$ and $\sigma_i \xrightarrow{|e_i,b_i|} \sigma_{i+1}$ is a nice transition, we have that $\mathrm{EA}(t,\sigma_i,\sigma_{i+1})$ holds. This fact can be inferred from the construction of EA. Therefore, $\sigma = \sigma_1 \xrightarrow{|t,\mathrm{EA}^n|} \sigma_n = \sigma'$.

1. If $\sigma' \xrightarrow{|t,a|} \omega$ is nice, then $A_a(t,\sigma')$ holds. If $\omega \neq \mathbf{wrong}$, the condition $p(t,\sigma')$ must also hold. Therefore, the following transitions may occur:
$$\sigma' \xrightarrow{|t,\texttt{assert}\ \mathit{enabled}(a)\Rightarrow A_a|} \sigma' \xrightarrow{|t,\texttt{assume}\ p|} \sigma' \xrightarrow{|t,X|} \omega$$
Thus $\sigma \xrightarrow{|t,\overline{v}|} \omega$.

2. If $\sigma' \xrightarrow{|t,a|} \omega$ is not nice, then $A_a(t,\sigma)$ does not hold. The atomic operation $|t,a|$ does not block in state σ', meaning that $\mathit{enabled}(a)(t,\sigma')$ holds. Therefore, the following transition may occur:
$$\sigma' \xrightarrow{|t,\texttt{assert}\ \mathit{enabled}(a)\Rightarrow A_a|} \mathbf{wrong}.$$
Thus $\sigma \xrightarrow{|t,\overline{v}|} \mathbf{wrong}$.

□

We extend the previous lemma to show that the algorithm to check nice transitions works properly on sequences of transitions.

LEMMA 3. *Suppose $\overline{a} = a_1;\ldots;a_n$ is a sequence of atomic operations and $\sigma_1 \xrightarrow{\overline{u}_1} \sigma'_1 \xrightarrow{|t,a_1|} \cdots \xrightarrow{\overline{u}_{n-1}} \sigma'_{n-1} \xrightarrow{|t,a_{n-1}|} \sigma_n \xrightarrow{\overline{u_n}} \sigma'_n$*

is a nice execution such that no step in $\overline{u}_1, \ldots, \overline{u}_n$ is a t-step. Then there is a $\overline{v} \in [\![CN(\overline{a})]\!]$ such that the following statements are true:

1. *If $\sigma'_n \xrightarrow{|t,a_n|} \omega$ is nice and $\omega \neq \mathbf{wrong}$, then $\sigma_1 \xrightarrow{|t,\overline{v}|} \omega$.*

2. *If $\sigma'_n \xrightarrow{|t,a_n|} \omega$ is not nice, then $\sigma_1 \xrightarrow{|t,\overline{v}|} \mathbf{wrong}$.*

PROOF. The proof is by induction on the length of $\overline{a} = n$.

Base case: If $n = 1$, Lemma 2 shows that both parts of the lemma are true.

Inductive Step: Consider the sequence $a_1; \ldots; a_n; a_{n+1}$. Suppose $\sigma_1 \xrightarrow{\overline{u}_1} \sigma'_1 \xrightarrow{|t,a_1|} \cdots \sigma_n \xrightarrow{\overline{u}_n} \sigma'_n \xrightarrow{|t,a_n|} \sigma_{n+1} \xrightarrow{\overline{u}_{n+1}} \sigma'_{n+1}$ is a nice execution such that no step in $\overline{u}_1, \ldots, \overline{u}_{n+1}$ is a t-step. By the induction hypothesis, there is $\overline{v} \in [\![CN(a_1; \ldots; a_n)]\!]$ such that $\sigma_1 \xrightarrow{|t,\overline{v}|} \sigma_{n+1}$. From Lemma 2, there is $\overline{v}' \in [\![CN(a_{n+1})]\!]$ such that

1. If $\sigma'_{n+1} \xrightarrow{|t,a_{n+1}|} \omega$ is nice and $\omega \neq \mathbf{wrong}$, then $\sigma_{n+1} \xrightarrow{|t,\overline{v}|} \omega$.

2. If $\sigma'_{n+1} \xrightarrow{|t,a_{n+1}|} \omega$ is not nice, then $\sigma_{n+1} \xrightarrow{|t,\overline{v}|} \mathbf{wrong}$.

We then get $\overline{v}; \overline{v}' \in [\![CN(a_1; \ldots; a_n; a_{n+1})]\!]$ with the desired property. □

RESTATEMENT OF THEOREM 5.1. Let S be a statement without procedure calls. For all $t \in \mathit{Tid}$, if $|t, CN(S)|$ does not go wrong, then $|t, S|$ is nice.

PROOF. Suppose that $|t, S|$ were not nice. Then there would exist a path $\overline{a} = a_1; \ldots; a_n; a_{n+1}; \ldots; a_{n+k} \in [\![S]\!]$ and a nice execution

$$\tau = \sigma_1 \xrightarrow{\overline{u_1}} \sigma'_1 \xrightarrow{|t,a_1|} \sigma_2 \xrightarrow{\overline{u_2}} \sigma'_2 \cdots \sigma'_{n-1} \xrightarrow{|t,a_{n-1}|} \sigma_n \xrightarrow{\overline{u_n}} \sigma'_n,$$

where each $\overline{u}_1, \ldots, \overline{u}_n$ contains no t-step, such that $\sigma'_n \xrightarrow{|t,a_n|} \omega$ is not a nice transition. From Lemma 3, there is a path $\overline{v} \in [\![CN(a_1; \ldots; a_n)]\!]$ such that $\sigma_1 \xrightarrow{|t,\overline{v}|} \mathbf{wrong}$. Therefore, there is a path $\overline{v}' \in CN(\overline{a})$ such that $\sigma_1 \xrightarrow{|t,\overline{v}'|} \mathbf{wrong}$. Since $\overline{a} \in [\![S]\!]$, we it must be the case that $\overline{v'} \in CN(S)$. However, this implies that $|t, CN(S)|$ goes wrong, and this contradicts our assumptions. □

C Proofs for CA

To prove Theorem 5.2, we must show that if $|t, CA(S, T, W)|$ does not go wrong, then $|t, S| \preceq |t, T|$. We proceed as follows. Consider any nice and $|t, S|$-complete execution τ. Lemmas 5–9 show that τ may be transformed via reduction into another nice execution τ' having the same initial and ending state such that every right-moving t-transition immediately precedes another t-transition and every left-moving t-transitions immediately follows another t-transition. Assuming that $\overline{a}$ is the projection of τ' onto the actions of thread t, we then show in Lemmas 10–11 that the abstraction checking algorithm considers the same steps in $\overline{a}$ right movers and left movers as those considered movers in the construction of τ'. In addition, those lemmas show that $|t, CA(\overline{a}, T, W)|$ goes wrong if $\overline{a}$ is not simulated by T, according to witness W.

We begin the proof by developing the notion of independent variables for an action and stating a few properties of left and right movers.

The set $\mathit{Access}(a)$ is the set of variables on which action a depends, and the set $\mathit{Indep}(a) = \mathit{Var} \setminus \mathit{Access}(a)$ is the set of variables of which a is independent. The following conditions are satisfied for all threads t and values $\overline{V}$:

1. if $\sigma \xrightarrow{|t,a|} \sigma'$, then $\sigma[\overline{\mathit{Indep}(a)} \mapsto \overline{V}] \xrightarrow{|t,a|} \sigma'[\overline{\mathit{Indep}(a)} \mapsto \overline{V}]$ and $\forall \mathtt{x} \in \mathit{Indep}(a).\ \sigma(\mathtt{x}) = \sigma'(\mathtt{x})$.

2. if $\sigma \xrightarrow{|t,a|} \mathbf{wrong}$, then $\sigma[\overline{\mathit{Indep}(a)} \mapsto \overline{V}] \xrightarrow{|t,a|} \mathbf{wrong}$.

The following lemma states that a nice transition is insensitive to the values of its independent variables.

LEMMA 4. *If $\sigma \xrightarrow{|t,a|} \omega$ is nice, then If $\sigma[\overline{\mathit{Indep}(a)} \mapsto \overline{V}] \xrightarrow{|t,a|} \omega'$ is also nice for all $\overline{V}$.*

PROOF. Consider any $\mathtt{x} \in \mathit{Access}(a)$. If $\sigma \xrightarrow{|t,a|} \omega$ is nice, then $A_{\mathtt{x}}(t, \sigma)$ holds. We must show that $A_{\mathtt{x}}(t, \sigma[\overline{\mathit{Indep}(a)} \mapsto \overline{V}])$ holds. By the definition of Access, all $\mathtt{y} \in \mathit{Access}(A_{\mathtt{x}})$ are also in $\mathit{Access}(a)$. Therefore, $\mathit{Access}(A_{\mathtt{x}}) \cap \mathit{Indep}(a) = \emptyset$, and thus $A_{\mathtt{x}}(t, \sigma[\overline{\mathit{Indep}(a)} \mapsto \overline{V}]) = A_{\mathtt{x}}(t, \sigma)$, which means that it holds. The same argument applies for all variables in $\mathit{Access}(a)$. □

We now present two lemmas that capture when an atomic operation may be commuted to the left or the right in a nice execution.

LEMMA 5. *Let t and e be different thread identifiers. Suppose $\sigma_1 \xrightarrow{|t,a|} \sigma_2 \xrightarrow{|e,b|} \omega$ is a nice execution and $E_a(t, \sigma')$ holds. Then there exists ω' such that $\sigma_1 \xrightarrow{|e,b|} \omega' \xrightarrow{|t,a|} \omega$ is a nice execution.*

PROOF. From the definitions of E_a and nice transitions, $\mathit{Access}(a) \cap \mathit{Access}(b) = \emptyset$, which implies that $\mathit{Access}(a) \subseteq \mathit{Indep}(b)$. Consider each case separately:

1. $\omega = \mathbf{wrong}$: $\sigma_1 = \sigma_2[\overline{\mathit{Access}(a)} \mapsto \overline{V}]$, for some $\overline{V}$. From the definition of access, we may show that if $\sigma_2 \xrightarrow{|e,b|} \mathbf{wrong}$, then

$$\sigma_2[\overline{\mathit{Indep}(b)} \mapsto \overline{V'}] \xrightarrow{|e,b|} \mathbf{wrong}$$

   Therefore, $\omega' = \mathbf{wrong}$, and we have $\sigma_2[\overline{\mathit{Access}(a)} \mapsto \overline{V}] \xrightarrow{|e,b|} \omega' \xrightarrow{|t,a|} \omega$. Lemma 4 indicates that this is a nice execution.

2. $\omega \neq \mathbf{wrong}$: Note that:

$$\begin{aligned} \sigma_1 &= \sigma_2[\overline{\mathit{Access}(a)} \mapsto \overline{V}] \\ \omega &= \sigma_2[\overline{\mathit{Access}(b)} \mapsto \overline{W}] \end{aligned}$$

Similar to the previous case, if $\sigma_2 \xrightarrow{|e,b|} \omega$, then

$$\sigma_2[\overline{Access(a)} \mapsto \overline{V}] \xrightarrow{|e,b|} \sigma_2'$$

where $\sigma_2' = \omega[\overline{Access(a)} \mapsto \overline{V}]$. Therefore, $\sigma_1 \xrightarrow{|e,b|} \sigma_2'$. Lemma 4 indicates that this is a nice execution. In addition, if $\sigma_1 \xrightarrow{|t,a|} \sigma_2$, then

$$\sigma_1[\overline{Indep(a)} \mapsto \overline{W'}] \xrightarrow{|t,a|} \sigma_2[\overline{Indep(a)} \mapsto \overline{W'}]$$

which indicates that

$$\sigma_1[\overline{Access(b)} \mapsto \overline{W}] \xrightarrow{|t,a|} \omega$$

Moreover, this is a nice execution, according to Lemma 4. Finally,

$$\begin{aligned} \sigma_2' &= \omega[\overline{Access(a)} \mapsto \overline{V}] \\ &= \omega[\overline{Access(a)} \mapsto \overline{V}][\overline{Access(b)} \mapsto \overline{W}] \\ &= \sigma_1[\overline{Access(b)} \mapsto \overline{W}] \end{aligned}$$

Therefore, $\sigma_1 \xrightarrow{|e,b|} \sigma_2' \xrightarrow{|t,a|} \omega$.

□

LEMMA 6. *Let t and e be different thread identifiers. Suppose $\sigma \xrightarrow{|e,b|} \omega' \xrightarrow{|t,a|} \omega$ is a nice execution and $E_a(t,\sigma)$ and $enabled(a)(t,\sigma)$ hold. Then there exists ω'' such that $\sigma \xrightarrow{|t,a|} \omega'' \xrightarrow{|e,b|} \omega$ is a nice execution.*

PROOF. Similar to the previous lemma. □

We would like to be able to identify which transitions in an execution will be treated as left movers and right movers by the abstraction checking algorithm. The variable `phase` records this information in the algorithm, and we now present a function $phase_\tau$ that computes the same information for any nice execution τ. An integral part of the following proofs is that the function $phase_\tau$ both accurately characterizes transitions that may be left movers or right movers and matches the value recorded in the `phase` variable during execution of the check abstraction algorithm.

In order to make certain technical aspects of the proof more straight-forward, we shall actually show the correctness of a variation of the algorithm presented in Section 5. The algorithm only differs in how it tracks left-movers, right-movers, and the `phase` variable. In particular, we introduce the new variable `rm` to record whether the action just executed is a right mover. In addition, `phase` may now be one of three values: RM (matching right movers), LM (matching left movers), and EOR (at end of reduction). Actions performed by the environment of the current thread occur when `phase` is EOR. This translation also does not update the `phase` variable after executing the operation `a`.

Figure 11: Checker for abstraction

```
CODE(l : a, T, W)  =def  if (rm) {
                           phase = RM
                         } else if (E_a ∧ enabled(a)) {
                           phase = LM
                         } else {
                           phase = EOR
                           assert ρ(T) ;
                           EA* ;
                           x_orig = x ;
                           pc_orig = pc
                         } ;
                         pc = W(l) ;
                         a ;
                         rm = (phase ≠ LM ∧ E_a)

CA(S, T, W)  =def  pc = begin ;
                   x_orig = x ;
                   pc_orig = pc ;
                   rm = false ;
                   CODE(S, T, W) ;
                   assert ρ(T) ;
                   assert (pc, end) ∈ lr(T)
```

Consider the following nice execution for some $k \geq 0$:

$$\tau = \omega_1 \xrightarrow{u_1} \cdots \xrightarrow{u_k} \omega_{k+1}$$

where $u_i = |t_i, a_i|$ for all $1 \leq i \leq k$. We define a function $phase_\tau$ that takes an integer (a thread identifier) and an index in the set $\{0, 1, \ldots, k+1\}$ and returns a member of the set { RM, LM, EOR }. To define $phase_\tau(t,i)$, we need the following definitions:

$$\begin{aligned} after_\tau(t,i) &= \{j \mid i \leq j \wedge t_j = t\} \\ before_\tau(t,i) &= \{j \mid i > j \wedge t_j = t\} \end{aligned}$$

We define $phase_\tau(t,i)$ as follows:

- $phase_\tau(t,0) = \mathrm{EOR}$.
- for $1 \leq i \leq k$, $phase_\tau(t,i+1)$ is defined by the following rules:
  1. If $\omega_{i+1} = \mathbf{wrong}$, then $phase_\tau(t,i+1) = \mathrm{EOR}$.
  2. Otherwise, if $phase_\tau(t,i) \neq \mathrm{LM}$, and $before_\tau(t,i+1) \neq \emptyset$, and n is the greatest element of $before_\tau(t,i+1)$, and the condition $E_{a_n}(t,\omega_{i+1})$ holds, then $phase_\tau(t,i+1) = \mathrm{RM}$.
  3. Otherwise, if $after_\tau(t,i+1) \neq \emptyset$, and n is the least element of $after_\tau(t,i+1)$, and the conditions $E_{a_n}(t,\omega_{i+1})$ and $enabled(a_n)(t,\omega_{i+1})$ hold, then $phase_\tau(t,i+1) = \mathrm{LM}$.
  4. Otherwise, $phase_\tau(t,i+1) = \mathrm{EOR}$.

We now state several properties of the $phase_\tau$ that will be useful for later proofs.

LEMMA 7. *If $\tau = \omega_1 \xrightarrow{|t_1,a_1|} \cdots \xrightarrow{|t_k,a_k|} \omega_{k+1}$ is a nice execution, then:*

1. *$phase_\tau(t,1) \neq \mathrm{RM}$ and $phase_\tau(t,k+1) \neq \mathrm{LM}$.*
2. *For all $1 \leq i \leq k$:*
   (a) *if $phase_\tau(t,i) = \mathrm{LM}$, then $phase_\tau(t,i+1) \neq \mathrm{RM}$.*
   (b) *if $t_i \neq t$, then $phase_\tau(t,i+1) = phase_\tau(t,i)$.*

(c) *if* $\omega_i \xrightarrow{|t_i,a_i|} \omega_{i+1} \xrightarrow{|t_{i+1},a_{i+1}|} \omega_{i+2}$, $t_i = t \neq t_{i+1}$, *and* $phase_\tau(t,i+1) = \mathrm{RM}$, *then there is an execution* $\tau' = \omega_1 \xrightarrow{|t_1,a_i|} \cdots \xrightarrow{|t_{i-1},a_{i-1}|} \omega_i \xrightarrow{|t_{i+1},a_{i+1}|} \omega'_{i+1} \xrightarrow{|t_i,a_i|} \omega_{i+2} \xrightarrow{|t_{i+2},a_{i+2}|} \cdots \xrightarrow{|t_k,a_k|} \omega_k$ *such that* $phase_{\tau'}(t,j) = phase_\tau(t,j)$ *for all* $j \neq i+1$.

(d) *if* $\omega_i \xrightarrow{|t_i,a_i|} \omega_{i+1} \xrightarrow{|t_{i+1},a_{i+1}|} \omega_{i+2}$, $t_i \neq t = t_{i+1}$, *and* $phase_\tau(t,i) = \mathrm{LM}$, *then there is an execution* $\tau' = \omega_1 \xrightarrow{|t_1,a_i|} \cdots \xrightarrow{|t_{i-1},a_{i-1}|} \omega_i \xrightarrow{|t_{i+1},a_{i+1}|} \omega'_{i+1} \xrightarrow{|t_i,a_i|} \omega_{i+2} \xrightarrow{|t_{i+2},a_{i+2}|} \cdots \xrightarrow{|t_k,a_k|} \omega_k$ *such that* $phase_{\tau'}(t,j) = phase_\tau(t,j)$ *for all* $j \neq i+1$.

PROOF. Parts 1, 2(a), and 2(b) follow from the definition of $phase_\tau$. Parts 3 and 4 use Lemmas 5 and 6 to identify the new executions having the transitions commuted, and the preservation of $phase_\tau$ for thread t at all states other than the one between the commuted actions follows from the definition of $phase_\tau$. □

In the next stage of the proof, we show that any nice execution τ of a program may be transformed via reduction into another nice execution τ' such that every right-moving t-transition immediately precedes another t-transition and every left-moving t-transitions immediately follows another t-transition.

We define a distance function $\delta_\tau(t)$ that measures how far an execution is from having this form. If a trace τ has the form described, $\delta_\tau(t)$ will be zero.

Given $\tau = \omega_1 \xrightarrow{u_1} \cdots \xrightarrow{u_k} \omega_{k+1}$ for some $k \geq 0$, we define $\delta_\tau(t)$ as follows:

$$\delta_\tau(t) = \sum_{i=1}^{k} \delta_\tau(t,i)$$

$$\delta_\tau(t,i) = \begin{cases} C_\tau(t,i) & \text{if } t_i \neq t \text{ and } phase_\tau(t,i) = \mathrm{RM} \\ D_\tau(t,i) & \text{if } t_i \neq t \text{ and } phase_\tau(t,i) = \mathrm{LM} \\ 0 & \text{otherwise} \end{cases}$$

$$C_\tau(t,i) = \begin{cases} 0 & \text{if } phase_\tau(t,i) = \mathrm{EOR} \\ 1 + C_\tau(t,i-1) & \text{otherwise} \end{cases}$$

$$D_\tau(t,i) = \begin{cases} 0 & \text{if } phase_\tau(t,i) = \mathrm{EOR} \\ 1 + D_\tau(t,i+1) & \text{otherwise} \end{cases}$$

Lemma 7 ensures that the quantities $\delta_\tau(t,i)$, $C_\tau(t,i)$, and $D_\tau(t,i)$ are well-defined. To see that $C_\tau(t,i)$ is well-defined, note that if $phase_\tau(t,i) = \mathrm{RM}$, then there is $1 \leq j < i$ such that $phase_\tau(t,j) = \mathrm{EOR}$. This means that any call to $C_\tau(t,i)$ from $\delta_\tau(t,i)$ terminates. If not then for all $1 \leq j < i$, $phase_\tau(t,i) \in \{\mathrm{RM}, \mathrm{LM}\}$. Since $phase_\tau(t,j+1) = \mathrm{RM} \Rightarrow phase_\tau(t,j) \neq \mathrm{LM}$, we get that for all $1 \leq j < i$, $phase_\tau(t,i) = \mathrm{RM}$, which contradicts the fact that $phase_\tau(t,1) \neq \mathrm{RM}$. Similarly, if $phase_\tau(t,i) = \mathrm{LM}$, we can argue that there is $i < j \leq k+1$ such that $phase_\tau(t,j) = \mathrm{EOR}$, which means that any call to $D_\tau(t,i)$ from $\delta_\tau(t,i)$ terminates.

LEMMA 8. *For any nice execution τ, there is another nice execution τ' such that τ' is obtained from τ by a sequence of reduce transformations and $\delta_{\tau'}(t) = 0$.*

PROOF. We prove the lemma by induction on $\delta_\tau(t)$.

Base Case: If $\delta_\tau(t) = 0$, then $\tau' = \tau$.

Inductive Case: We show that if $\delta_\tau(t) > 0$ then there is a nice sequence τ' such that τ' is obtained from τ by a single reduce transformation and $\delta_{\tau'}(t) < \delta_\tau(t)$. Suppose $\tau = \omega_1 \xrightarrow{u_1} \cdots \xrightarrow{u_k} \omega_{k+1}$ for some $k \geq 0$. If $\delta_\tau(t) > 0$, then there is some i such that $\delta_\tau(t,i) > 0$. There are two cases:

1. $t_i = t$, $phase_\tau(t,i+1) = \mathrm{RM}$ and $C_\tau(t,i) > 0$. Since $C_\tau(t,i) > 0$, there is a j such that $i \leq j < k$, $\omega_j \xrightarrow{|t_j,a_j|} \omega_{j+1} \xrightarrow{|t_{j+1},a_{j+1}|} \omega_{j+2}$, $t_j = t \neq t_{j+1}$, and $phase_\tau(t,j+1) = \mathrm{RM}$. By Lemma 7, there exists a nice execution $\tau' = \omega_1 \xrightarrow{\overline{u}} \omega_i \xrightarrow{|e,b|} \omega'_{i+1} \xrightarrow{|t,a|} \omega_{i+2} \xrightarrow{\overline{w}} \omega_k$, such that for all $j \neq i+1$, $phase_{\tau'}(t,j) = phase_\tau(t,j)$. According to Lemma 7, $phase_\tau(t,i) \neq \mathrm{LM}$. Therefore, $phase_{\tau'}(t,i) = phase_\tau(t,i) \neq \mathrm{LM}$. Lemma 7 also indicates that $phase_{\tau'}(t,i+1) = phase_{\tau'}(t,i) \neq \mathrm{LM}$. Therefore, $phase_{\tau'}(t,i) = phase_{\tau'}(t,i+1) \in \{\mathrm{RM}, \mathrm{EOR}\}$. Clearly, $\delta_{\tau'}(t,i) < \delta_\tau(t,i)$. Moreover, for any $j \neq i$ such that $phase_{\tau'}(t,j) = \mathrm{RM}$ and $t_j \neq t$, we have $\delta_{\tau'}(t,i) \leq \delta_\tau(t,i)$. Thus, $\delta_{\tau'}(t) < \delta_\tau(t)$.

2. $t_i = t$, $phase_\tau(t,i) = \mathrm{LM}$ and $D_\tau(t,i) > 0$. Similar to the first case.

□

An important property of any execution of the form described above is that the phase for transitions taken by threads other than t will be always be EOR.

LEMMA 9. *Suppose $\tau = \sigma_1 \xrightarrow{|t_1,a_1|} \cdots \xrightarrow{|t_k,a_k|} \sigma_{k+1}$ is a nice execution such that $\delta_\tau(t) = 0$. For all $1 \leq i \leq k$, if $t \neq t_i$ then $phase_\tau(t,i) = \mathrm{EOR}$.*

PROOF. Suppose that there exists i such that $t \neq t_i$ and $phase_\tau(t,i) = \mathrm{RM}$. According to Lemma 7, $i > 1$. Thus, $\delta_\tau(t) \geq \delta_\tau(t,i) = C_\tau(t,i) = 1 + C_\tau(t,i-1) > 0$. Therefore, $\delta_\tau(t) \neq 0$. The case when $phase_\tau(t,i) = \mathrm{LM}$ is similar. □

In the remainder of this section, we fix two statements S and T without procedure calls, such that each atomic operation in T is an action. We also fix a witness expression W and assume that all atomic operations have been assigned a unique label. An *augmented store* is a map giving values to the variables $Var \cup \{\mathtt{x}_{\mathrm{orig}} \mid \mathtt{x} \in Var\} \cup \{\mathtt{pc}, \mathtt{pc}_{\mathrm{orig}}\} \cup \{\mathtt{phase}, \mathtt{rm}\}$. We use the symbols $\pi, \pi_0, \ldots$ to denote an augmented store. Given a store σ and an augmented store π, we say that $\sigma \doteq \pi$ if $\sigma(\mathtt{x}) = \pi(\mathtt{x})$ for all $\mathtt{x} \in Var$.

LEMMA 10. *Suppose*

- *$\overline{u}$ is a path of length $p \geq 0$ without any t-steps,*
- *$|t, \overline{a}|$ is a path of length $q > 0$ with only t-steps,*

- $\tau = \sigma_0 \xrightarrow{\overline{u}} \sigma_p \xrightarrow{|t,a_1|} \sigma_{p+1} \cdots \sigma_{p+q-1} \xrightarrow{|t,a_q|} \omega$ *is a nice execution such that* $\delta_\tau(t) = 0$ *and* $phase_\tau(t, p+i) \neq \mathrm{EOR}$ *for all* $1 \leq i < q$*, and*

- π_0 *is an augmented store such that* $\pi_0 \doteq \sigma_0$, $\pi_0(\mathtt{rm}) = \mathtt{false}$*, and* $|t, CODE(\overline{a}, T, W)|$ *does not go wrong from* π_0.

Then $\omega \neq \mathbf{wrong}$ *and there is a* $\overline{v} \in [\![CODE(\overline{a}, T, W)]\!]$ *and an augmented store* π *such that*

1. $\pi_0 \xrightarrow{|t,\overline{v}|} \pi$, $\pi \doteq \omega$*, and* $\pi(\mathtt{rm})$ *iff* $phase_\tau(t, p+q) = \mathrm{RM}$.

2. *if* $\rho(T)$ *holds in* π*, then either* $\pi_0(\mathtt{pc}) = \pi(\mathtt{pc})$ *and* $\sigma_p = \omega$ *or* $(\pi_0(\mathtt{pc}), \pi(\mathtt{pc})) \in lr(T)$ *and* $\sigma_p \xrightarrow{|t,Act_T(\pi(\mathtt{pc}))|} \omega$.

PROOF. By induction over the length p of $|t, \overline{a}|$. □

LEMMA 11. *Suppose*

- $\overline{u}_1, \ldots, \overline{u}_n$ *are paths of lengths* $p(1), \ldots, p(n) \geq 0$ *respectively each without a* t*-step,*

- $|t, \overline{a}_1|, \ldots, |t, \overline{a}_n|$ *are paths of lengths* $q(1), \ldots, q(n) > 0$ *respectively each with only* t*-steps,*

- $\tau = \sigma_{r(0)} \xrightarrow{\overline{u}_1} \sigma_{r(1)} \xrightarrow{|t,\overline{a}_1|} \sigma_{r(2)} \cdots \sigma_{r(2n-2)} \xrightarrow{\overline{u}_n} \sigma_{r(2n-1)} \xrightarrow{|t,\overline{a}_n|} \omega$ *is a nice execution such that* $\delta_\tau(t) = 0$ *and* $phase_\tau(t, j) \neq \mathrm{EOR}$ *for all* $0 \leq i < n$ *and* $r(2i+1) < j < r(2i+2)$*, where* $r(0), r(1), r(2), \ldots, r(2n-1), r(2n)$ *are defined such that* $r(0) = 0$ *and for all* $0 \leq i < n$,

$$
\begin{aligned}
r(2i+1) &= r(2i) + p(i+1) \\
r(2i+2) &= r(2i+1) + q(i+1),
\end{aligned}
$$

and

- π_0 *is an augmented store such that* $\pi_0 \doteq \sigma_{r(0)}$, $\pi_0(\mathtt{rm}) = \mathtt{false}$*, and* $|t, CODE(\overline{a}_1; \ldots; \overline{a}_n, T, W)|$ *does not go wrong from* π_0.

Then $\omega \neq \mathbf{wrong}$ *and there is a* $\overline{v} \in [\![CODE(\overline{a}_1; \ldots; \overline{a}_n, T, W)]\!]$ *and an augmented store* π *such that*

1. $\pi_0 \xrightarrow{|t,\overline{v}|} \pi$, $\pi \doteq \omega$, $\pi(\mathtt{x}_{\mathrm{orig}}) = \sigma_{r(2n-1)}(\mathtt{x})$ *for all* $\mathtt{x} \in Var$*, and* $\pi(\mathtt{rm})$ *iff* $phase_\tau(t, r(2n)) = \mathrm{RM}$*, and*

2. *if* $\rho(T)$ *holds in* π*, then there are labels* $l_0, l_1, \ldots, l_n$*, actions* $b_1, \ldots, b_n$*, and an execution* $\sigma_{r(0)} \xrightarrow{\overline{u}_1} \sigma_{r(1)} \xrightarrow{|t,b_1|} \sigma_{r(2)} \cdots \sigma_{r(2n-1)} \xrightarrow{|t,b_n|} \sigma_{r(2n)}$ *such that* $l_0 = \pi_0(\mathtt{pc})$, $l_n = \pi(\mathtt{pc})$*, and for all* $1 \leq i \leq n$*, either* $b_i = \mathtt{skip}$ *and* $l_{i-1} = l_i$ *or* $b_i = Act_T(l_i)$ *and* $(l_{i-1}, l_i) \in lr(T)$.

PROOF. We do the proof by induction over n.

Base case: $n = 1$. By direct application of Lemma 10.

Inductive step: Suppose $\tau = \sigma_{r(0)} \xrightarrow{\overline{u}_1} \sigma_{r(1)} \xrightarrow{|t,\overline{a}_1|} \sigma_{r(2)} \cdots \sigma_{r(2n-2)} \xrightarrow{\overline{u}_n} \sigma_{r(2n-1)} \xrightarrow{|t,\overline{a}_n|} \sigma_{r(2n)} \xrightarrow{\overline{u}_{n+1}} \sigma_{r(2n+1)} \xrightarrow{|t,\overline{a}_{n+1}|} \omega$ is a nice execution such that $\delta_\tau(t) = 0$ and $phase_\tau(t, j) \neq \mathrm{EOR}$ for all $0 \leq i < n+1$ and $r(2i+1) < j < r(2i+2)$. Suppose π_0 is an augmented store such that $\pi_0 \doteq \sigma_0$, $\pi_0(\mathtt{rm}) = \mathtt{false}$, and $|t, CODE(\overline{a}_1; \ldots; \overline{a}_{n+1}, T, W)|$ does not go wrong from π_0.

Let $\overline{a} = \overline{a}_1; \ldots; \overline{a}_n$. Let τ_1 be the prefix $\sigma_{r(0)} \xrightarrow{\overline{u}_1} \sigma_{r(1)} \xrightarrow{|t,\overline{a}_1|} \sigma_{r(2)} \cdots \sigma_{r(2n-2)} \xrightarrow{\overline{u}_n} \sigma_{r(2n-1)} \xrightarrow{|t,\overline{a}_n|} \sigma_{r(2n)}$ of τ and τ_2 be the suffix $\sigma_{r(2n)} \xrightarrow{\overline{u}_{n+1}} \sigma_{r(2n+1)} \xrightarrow{|t,\overline{a}_{n+1}|} \omega$ of τ.

We apply the inductive hypothesis to the augmented store π_0 and the execution τ_1. There is $\overline{v}_1 \in [\![CODE(\overline{a}, T, W)]\!]$ and an augmented store π_1 such that $\pi_0 \xrightarrow{|t,\overline{v}_1|} \pi_1$, $\pi_1 \doteq \sigma_{r(2n)}$, $\pi_1(\mathtt{x}_{\mathrm{orig}}) = \sigma_{r(2n-1)}(\mathtt{x})$ for all $\mathtt{x} \in Var$, and $\pi_1(\mathtt{rm})$ iff $phase_{\tau_1}(t, r(2n)) = \mathrm{RM}$. Since $|t, CODE(\overline{a}_1; \ldots; \overline{a}_{n+1}, T, W)|$ does not go wrong from π_0, we have that $\rho(T)$ holds in π_1. Therefore, we also get labels $l_0, l_1, \ldots, l_n$, actions $b_1, \ldots, b_n$, and an execution $\sigma_{r(0)} \xrightarrow{\overline{u}_1} \sigma_{r(1)} \xrightarrow{|t,b_1|} \sigma_{r(2)} \cdots \sigma_{r(2n-1)} \xrightarrow{|t,b_n|} \sigma_{r(2n)}$ such that $l_0 = \pi_0(\mathtt{pc})$, $l_n = \pi_1(\mathtt{pc})$, and for all $1 \leq i \leq n$, either $b_i = \mathtt{skip}$ and $l_{i-1} = l_i$ or $b_i = Act_T(l_i)$ and $(l_{i-1}, l_i) \in lr(T)$.

Since $\delta_\tau(t) = 0$, we have $phase_\tau(t, r(2n)) = \mathrm{EOR}$. Therefore $phase_{\tau_1}(t, r(2n)) \neq \mathrm{RM}$ and $\pi_1(\mathtt{rm})$ is false. We now apply Lemma 10 to the execution τ_2 and the augmented store π_1. We get that $\omega \neq \mathbf{wrong}$ and there is a $\overline{v}_2 \in [\![CODE(\overline{a}_{n+1}, T, W)]\!]$ and an augmented store π_2 such that

1. $\pi_1 \xrightarrow{|t,\overline{v}_2|} \pi_2$, $\pi_2 \doteq \omega$, and $\pi_2(\mathtt{rm})$ iff $phase_{\tau_2}(t, p(n+1) + q(n+1)) = \mathrm{RM}$.

2. if $\rho(T)$ holds in π_2, then either $\pi_1(\mathtt{pc}) = \pi_2(\mathtt{pc})$ and $\sigma_{r(2n+1)} = \omega$ or $(\pi_1(\mathtt{pc}), \pi_2(\mathtt{pc})) \in lr(T)$ and $\sigma_{r(2n+1)} \xrightarrow{|t,Act_T(\pi_2(\mathtt{pc}))|} \omega$.

Let $l_{n+1} = \pi_2(\mathtt{pc})$. If $\rho(T)$ holds in π_2, then either $l_n = l_{n+1}$ and $\sigma_{r(2n+1)} = \omega$, in which case let $b_n = \mathtt{skip}$, or $(l_n, l_{n+1}) \in lr(T)$ and $\sigma_{r(2n+1)} \xrightarrow{|t,Act_T(l_{n+1})|} \omega$, in which case let $b_n = Act_T(l_{n+1})$. We get $\overline{v} = \overline{v}_1; \overline{v}_2 \in [\![CODE(\overline{a}; \overline{a}_{n+1}, T, W)]\!]$ and π_2 with the appropriate properties. □

RESTATEMENT OF THEOREM 5.2. Let S be a statement without procedure calls. Let T be a statement without procedure calls whose atomic operations are actions. Let W be a witness expression. For all $t \in Tid$, if $|t, CA(S, T, W)|$ does not go wrong, then $|t, S| \preceq |t, T|$.

PROOF. Suppose τ is a $|t, S|$-complete and nice execution. From Lemma 8, there is a $|t, S|$-complete and nice execution τ_1 obtained from τ by a sequence of reduce transformations such that $\delta_{\tau_1}(t) = 0$. Let τ_2 be the longest prefix of τ_1 whose last transition is a t-transition starting from a store. Let τ_2' be the remainder of the execution in τ_1 after τ_2. Then $\tau_2 = \sigma_{r(0)} \xrightarrow{\overline{u}_1} \sigma_{r(1)} \xrightarrow{|t,\overline{a}_1|} \sigma_{r(2)} \cdots \sigma_{r(2n-2)} \xrightarrow{\overline{u}_n} \sigma_{r(2n-1)} \xrightarrow{|t,\overline{a}_n|} \omega$ such that

$\overline{u}_1,\ldots,\overline{u}_n$ are paths each without a t-step, $|t,\overline{a}_1|,\ldots,|t,\overline{a}_n|$ are paths each with only t-steps, $\delta_\tau(t)=0$ and $phase_\tau(t,j)\neq \mathrm{EOR}$ for all $0\leq i<n$ and $r(2i+1)<j<r(2i+2)$. Let $\overline{a}=\overline{a}_1;\ldots;\overline{a}_n$. Let π_0 be an augmented store such that $\pi_0\doteq\sigma_{r(0)}$, $\pi_0(\mathtt{rm})=\mathtt{false}$, $\pi_0(\mathtt{pc})=\mathtt{begin}$, and $\pi_0(\mathtt{x}_{\mathrm{orig}})=\sigma(\mathtt{x})$ for all $\mathtt{x}\in Var\cup\{\mathtt{pc}\}$. Since $|t,CA(S,T,W)|$ does not go wrong, then $|t,CODE(\overline{a},T,W)|$ does not go wrong from π_0. From the Lemma 11, we get that $\omega\neq\mathbf{wrong}$. Let $\omega=\sigma_{r(2n)}$ in the remainder of this proof. We also get a sequence $\overline{v}\in[\![CODE(\overline{a}_1;\ldots;\overline{a}_n,T,W)]\!]$ and an augmented store π such that

1. $\pi_0\xrightarrow{|t,\overline{v}|}\pi$, $\pi\doteq\omega$, $\pi(\mathtt{x}_{\mathrm{orig}})=\sigma_{r(2n-1)}(\mathtt{x})$ for all $\mathtt{x}\in Var$, and $\pi(\mathtt{rm})$ iff $phase_\tau(t,r(2n))=\mathrm{RM}$, and

2. if $\rho(T)$ holds in π, then there are labels $l_0,l_1,\ldots,l_n$, actions $b_1,\ldots,b_n$, and an execution $\sigma_{r(0)}\xrightarrow{\overline{u}_1}\sigma_{r(1)}\xrightarrow{|t,b_1|}\sigma_{r(2)}\cdots\sigma_{r(2n-1)}\xrightarrow{|t,b_n|}\sigma_{r(2n)}$ such that $l_0=\pi_0(\mathtt{pc})$, $l_n=\pi(\mathtt{pc})$, and for all $1\leq i\leq n$, either $b_i=\mathtt{skip}$ and $l_{i-1}=l_i$ or $b_i=Act_T(l_i)$ and $(l_{i-1},l_i)\in lr(T)$.

We do a case analysis on whether τ_2 is $|t,S|$-complete.

1. Suppose τ_2 is $|t,S|$-complete. From the definition of $CA(S,T,W)$, we get that $\rho(T)$ holds in π and that $(\pi(\mathtt{pc}),\mathtt{end})\in lr(T)$. We get labels $l_0,l_1,\ldots,l_{n-1},l_n$, actions $b_1,\ldots,b_{n-1},b_n$, and an execution $\tau_3=\sigma_{r(0)}\xrightarrow{\overline{u}_1}\sigma_{r(1)}\xrightarrow{|t,b_1|}\sigma_{r(2)}\cdots\sigma_{r(2n-1)}\xrightarrow{|t,b_n|}\sigma_{r(2n)}$ such that $l_0=\mathtt{begin}$, $(l_n,\mathtt{end})\in lr(T)$, and for all $1\leq i\leq n$, either $b_i=\mathtt{skip}$ and $l_{i-1}=l_i$ or $b_i=Act_T(l_i)$ and $(l_{i-1},l_i)\in lr(T)$. Thus τ_3 is $|t,T|$-complete and is obtainable from τ_2 by a sequence of t-simulate transformations. The execution τ_2' does not have any t-transitions. We obtain the desired execution by concatenating τ_3 with τ_2'.

2. Suppose τ_2 is not $|t,S|$-complete. Since τ_1 is $|t,S|$-complete, there is a prefix of τ_1 given by $\sigma_{r(0)}\xrightarrow{\overline{u}_1}\sigma_{r(1)}\xrightarrow{|t,\overline{a}_1|}\sigma_{r(2)}\cdots\sigma_{r(2n-1)}\xrightarrow{|t,\overline{a}_n|}\sigma_{r(2n)}\xrightarrow{\overline{u}_{n+1}}\mathbf{wrong}\xrightarrow{|t,a|}\mathbf{wrong}$, such that there are no t-steps in $\overline{u}_{n+1}$. Since $\delta_{\tau_1}(t)=0$, we have $phase_{\tau_1}(t,r(2n))=\mathrm{EOR}$. Therefore $\pi(\mathtt{rm})=\mathtt{false}$ and $(E_a\wedge enabled(a))(t,\sigma_{r(2n)})=\mathtt{false}$. Let $\overline{a}=\overline{a}_1;\ldots;\overline{a}_n$. We have that

   $$CODE(\overline{a};a,T,W)=CODE(\overline{a},T,W);CODE(a,T,W).$$

   Since $|t,CODE(\overline{a};a,T,W)|$ does not go wrong from π_0, we must have that $|t,CODE(a,T,W)|$ does not go wrong from π. Therefore $\rho(T)$ holds in π. Therefore we get labels $l_0,l_1,\ldots,l_n$, actions $b_1,\ldots,b_n$, and an execution $\tau_3=\sigma_{r(0)}\xrightarrow{\overline{u}_1}\sigma_{r(1)}\xrightarrow{|t,b_1|}\sigma_{r(2)}\cdots\sigma_{r(2n-1)}\xrightarrow{|t,b_n|}\sigma_{r(2n)}$ such that $l_0=\pi_0(\mathtt{pc})=\mathtt{begin}$, $l_n=\pi(\mathtt{pc})$, and for all $1\leq i\leq n$, either $b_i=\mathtt{skip}$ and $l_{i-1}=l_i$ or $b_i=Act_T(l_i)$ and $(l_{i-1},l_i)\in lr(T)$. Then τ_3 is obtainable from τ_2 by a sequence of t-simulate transformations.

   We perform a sequence of reduce transformations on τ_2' and move all transitions that are not t-transitions before all t-transitions. By performing these transformations on τ_2', we get the execution $\sigma_{r(2n)}\xrightarrow{\overline{u}}\mathbf{wrong}\xrightarrow{|t,\overline{a}'|}\mathbf{wrong}$. There must be a sequence $\overline{b}'$ such that $b_1;\ldots;b_n;\overline{b}'\in[\![T]\!]$. If $\overline{b}'$ is empty, we perform a t-simulate transformation to get $\tau_3'=\sigma_{r(2n)}\xrightarrow{\overline{u}}\mathbf{wrong}$. If $\overline{b}'$ is not empty, we perform a t-simulate transformation to get the execution $\tau_3'=\sigma_{r(2n)}\xrightarrow{\overline{u}}\mathbf{wrong}\xrightarrow{|t,\overline{b}'|}\mathbf{wrong}$. We finally get a $|t,T|$-complete execution by concatenating τ_3 and τ_3'.

□