Electronic Notes in Theoretical Computer Science 89 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 22 pages

Transactions for Software Model Checking

Cormac Flanagan

Hewlett-Packard Labs
1501 Page Mill Road, Palo Alto, CA 94304

Shaz Qadeer

Microsoft Research
One Microsoft Way, Redmond, WA 98052

Abstract

This paper presents a software model checking algorithm that combats state explosion by decomposing each thread's execution into a sequence of transactions that execute atomically. Our algorithm infers transactions using the theory of reduction, and supports both left and right movers, thus yielding larger transactions and fewer context switches than previous methods. Our approach uses *access predicates* to support a wide variety of synchronization mechanisms. In addition, we automatically infer these predicates for programs that use lock-based synchronization.

Key words: Model checking, multithreaded software, reduction, transactions

1 Introduction

The theory of reduction [Lip75] was introduced by Lipton to reduce the intellectual complexity of proving deadlock-freedom for parallel programs. Since then, many researchers [Doe77,LS89,CL98,Mis01] have developed the theory to incorporate general safety and liveness properties. Recently, a few practical applications of this theory have emerged in type and effect systems [FQ03c,FQ03b], verification-condition based static checking [FQ02], and model checking [SC03]. This paper presents a powerful and fully automatic algorithm that uses the theory of reduction to combat the state-explosion problem, the fundamental source of complexity in model checking multithreaded software.

Reduction takes a transactional view of computations occurring in individual threads of a multithreaded program. If all threads follow a synchronization discipline for accessing shared variables, the execution of a thread can be decomposed into transactions that can be considered to execute atomically from the point of view of other threads even though the scheduler interleaves actions of other threads during the transaction. Each transaction is a sequence of actions $a_1, \ldots, a_m, x, b_1, \ldots, b_n$ such that each a_i is a right mover and each b_i is a left mover. A right mover is an action, such as a lock acquire, that commutes to the right of every action by another thread; a left mover is an action, such as a lock release, that commutes to the left of every action by another thread. Accessing (reading or writing) a shared variable while holding its protecting lock is a both-mover, that is, it is both a right-mover and a left-mover. This view of concurrent computations has been used in database theory to prove the serializability of database transactions [Pap86].

The notion of transactions immediately suggests a method to reduce the number of explored states during model checking. The recipe is simply stated: deduce transactions in threads and avoid scheduling a thread in the middle of another thread's transaction. The idea of controlling the process scheduler to combat state-space explosion is not new. For example, the class of techniques known as partial-order reduction [Val90,Pel94,God96] use process scheduling algorithms to effect state reduction. However, these algorithms do not use the notion of transactions. Recently, Stoller and Cohen [SC03] have proposed a model checking algorithm that leverages Lipton's theory of reduction. A careful examination of all these approaches reveals that they use only the notion of left movers. In this paper, we present a model checking algorithm that uses both left and right movers to automatically deduce transactions in threads and reduce the state space explored during model checking. Since our algorithm uses both right and left movers, it can infer larger transactions than the algorithm of Stoller and Cohen.

The key to deducing transactions automatically is the notion of *exclusive access predicates* for shared variables. Exclusive access predicates were first introduced in the Calvin checker [FQ02] to semantically capture a multitude of synchronization disciplines by a single specification mechanism. Each shared variable is associated with an exclusive access predicate which states the condition under which a thread has exclusive access to that variable. For example, the exclusive predicate for a variable `x` protected by a mutex `mx` says that a thread has exclusive access to `x` if it holds the lock `mx`. A second thread must respect this exclusive predicate, and not access `x` whenever the first thread holds the lock.

The presentation of our approach proceeds as follows. Section 2 presents an example that illustrates the benefit our approach. Section 3 formalizes our notion of multithreaded programs, and Section 4 presents our transaction-based model checking algorithm for such programs. Section 5 specializes this approach to lock-based programs, and Section 6 describes how to infer exclusive predicates during transaction-based model checking of lock-based programs. Section 7 discusses related work, and we conclude with Section 8.

2 Overview

We illustrate our approach by a simple example. Below, we show an implementation of a multithreaded counter with three methods —`incr`, `decr`, and `read`. The shared variable `x` keeps track of the number of increment operations performed on the counter and is protected by the mutex `mx`. Similarly, the shared variable `y` keeps track of the number of decrement operations performed on the counter and is protected by the mutex `my`. A mutex is modeled an an integer variable. If the mutex is held by a thread, its value is the positive integer identifying that thread; otherwise, its value is 0. The variable `count` is an *abstract* variable introduced solely for the purpose of specifying the correctness of the counter implementation. The correctness property is stated as an assertion in the method `read`. Since `count` is a specification variable, we assume that the increment and decrement operations on `count` in `incr` and `decr` respectively are atomic. Hence, there is no need to protect `count` with a mutex. For the purpose of model checking, we restrict all integer variables to the first k natural numbers and interpret addition and subtraction modulo k.

```
int x = 0, y = 0, mx = 0, my = 0, count = 0;

void incr() {         void decr() {         int read() {
  acquire(mx);          acquire(my);          acquire(mx);
  int tx = x;           int ty = y;           int tx = x;
  count++;              count--;              acquire(my);
  x = tx+1;             y = ty+1;             int ty = y;
  release(mx);          release(mx);          assert count == tx-ty;
}                     }                       release(my);
                                              release(mx);
                                              return tx-ty;
                                            }
```

Consider a multithreaded program consisting of two threads, each of which repeatedly calls a nondeterministically chosen method—`incr`, `decr`, or `read`. Each thread follows the synchronization discipline of accessing `x` and `y` only when the correct mutexes are held. An action that acquires a lock, such as `acquire(mx)`, is a right mover. An action that releases a lock, such as `release(mx)`, is a left mover. An access to a thread-local variable, such as `tx`, is both a right and left mover. An access to a shared variable `x` performed while holding its lock `mx` is also both a right and left mover. An access to an unprotected shared variable, such as `count`, is neither a right nor a left mover. Thus, the body of each method is a transaction since all the actions preceding the unique access to `count` are right movers and all succeeding actions are left movers. We say that the transaction is in the *pre-commit* stage before the access to `count` happens and in the *post-commit* stage afterwards. Our model checking algorithm is able to automatically deduce these transactions, one per procedure. On the other hand, the algorithm of Stoller and Cohen will deduce two transactions in `incr` and `decr` and three transactions in `read`.

Our algorithm depends on the notion of exclusive predicates for shared variables to infer these transactions automatically. The exclusive predicates for the shared variables in this example are:

$$
\begin{aligned}
E(t, \mathtt{x}) &\stackrel{\text{def}}{=} \mathtt{mx} == t \\
E(t, \mathtt{y}) &\stackrel{\text{def}}{=} \mathtt{my} == t \\
E(t, \mathtt{mx}) &\stackrel{\text{def}}{=} \mathtt{mx} == t \\
E(t, \mathtt{my}) &\stackrel{\text{def}}{=} \mathtt{my} == t \\
E(t, \mathtt{count}) &\stackrel{\text{def}}{=} \mathit{false}
\end{aligned}
$$

For example, the exclusive predicate for `x` states that the thread t has exclusive access to `x` whenever the value of `mx` is t, that is, the thread t holds lock `mx`. Note that a mutex is just another shared variable for our algorithm; consequently, the mutexes have exclusive access predicates associated with them as well. No thread ever has exclusive access to `count` since its predicate is *false*.

Our algorithm uses these exclusive predicates to infer whether an action performed by a thread is a right or a left mover. An action is a right mover if that thread has exclusive access to all shared variables accessed by the action in the post-state. Similarly, an action is a left mover if the thread has exclusive access to all accessed shared variables in the pre-state.

3 Multithreaded programs

This section formalizes a semantics of multithreaded programs that we use to describe our transaction-based model checking algorithm.

Domains

$x, y \in \mathit{Var}$		(variables)
$v \in \mathit{Val}$		(values)
$t, u \in \mathit{Tid}$		(thread identifiers)
$l \in \mathit{Loc}$	$= \{\mathit{Init}, \mathit{Wrong}, \ldots\}$	(program counter locations)
$s \in \mathit{Store}$	$= \mathit{Var} \rightarrow \mathit{Val}$	(global stores)
$ls \in \mathit{Locs}$	$= \mathit{Tid} \rightarrow \mathit{Loc}$	(local stores)
$(s, ls) \in \mathit{State}$	$= \mathit{Store} \times \mathit{Locs}$	(states)

A state of the multithreaded program P consists of a global store mapping variables to values and a local store mapping thread identifiers to program counters. The set *Loc* of program counters contains the program counter *Init*, which is the initial program counter of each thread. Thus, the initial state of P is (s_0, ls_0), where s_0 is the initial value of the store and $ls_0(t) = \mathit{Init}$ for all threads t.

Each thread has a set of actions, one for each location in *Loc*. The action in thread t at location l is $Act(t,l) \subseteq Store \times (Store \times Loc)$. The transition relation $\rightarrow^t_P$ of thread t and the transition relation $\rightarrow_P$ of the multithreaded program P are defined as follows:

$$\begin{array}{rcl} \rightarrow^t_P & \subseteq & State \times State \\ (s, ls) \rightarrow^t_P (s', ls') & \stackrel{\text{def}}{=} & \exists l, l' \in Loc. \wedge\, l = ls(t) \\ & & \qquad\qquad\quad \wedge\, ls' = ls[t := l'] \\ & & \qquad\qquad\quad \wedge\, (s, s', l') \in Act(t, l) \\ \rightarrow_P & \subseteq & State \times State \\ \rightarrow_P & \stackrel{\text{def}}{=} & \exists t.\ \rightarrow^t_P \end{array}$$

The set Loc also contains a special location $Wrong$. When a thread fails an assertion, it moves its program counter to $Wrong$, and does not perform any subsequent transitions. Thus, for all threads t, we have $Act(t, Wrong) = \emptyset$. The program P *goes wrong* if it reaches a state (s, ls) where $ls(t) = Wrong$ for some thread t.

For clarity, we use bullet-style notation for $\wedge$ and $\vee$ in large formulas, following Lamport [Lam94]. In addition, we sometimes interpret sets as predicates, and vice-versa.

4 Inferring transactions

In this section, we show how to infer transactions in the threads of a multithreaded program.

4.1 Exclusive access predicates

We assume that for each variable $x \in Var$ and thread $t \in Tid$, there is an *exclusive predicate* $E(t, x) \subseteq Store$. The predicate $E(t, x)$ gives the condition under which thread t is guaranteed to have exclusive access to x. We assume that the exclusive predicates for any variable x satisfies the following two properties:

(i) The predicates $E(t, x)$ and $E(u, x)$ cannot be true simultaneously for different threads t and u.

$$\forall t, u \in Tid.\ t = u \vee E(t, x) \cap E(u, x) = \emptyset$$

(ii) The exclusive access to x can neither be given to thread t nor taken away from thread t by a different thread u.

$$\forall t, u \in Tid.\ (t \neq u \wedge (s, ls) \rightarrow^u_P (s', ls')) \Rightarrow (s \in E(t, x) \Leftrightarrow s' \in E(t, x))$$

The action $Act(t, l)$ *does not access* the variable x if it neither writes nor reads x. Formally, the the action $Act(t, l)$ does not access x if for all $(s, s', l') \in Act(t, l)$ and values v, we have $s(x) = s'(x)$ and $(s[x := v], s'[x := v], l') \in Act(t, l)$. Let $\alpha(t, l)$ denote the set of all variables that are accessed by the action $Act(t, l)$.

The predicate $E(t, l)$ is the conjunction of the exclusive predicates of all accessed variables of $Act(t, l)$:

$$E(t, l) \stackrel{\text{def}}{=} \forall x \in \alpha(t, l).\ E(t, x)$$

Our definition of exclusive access predicates does not require that $E(t, x)$ hold when x is accessed by thread t. This property, while desirable, is not enforced statically by the definition. It is checked dynamically through model checking by instrumenting the program with assertions. The scheme for program instrumentation is described in detail below.

4.2 Program instrumentation

We now present a method for automatically instrumenting a multithreaded program P to get another multithreaded program $P^{\#}$, for which it is easier to infer transaction boundaries during model checking.

Let $\mathit{enabled}(t, l) \subseteq \mathit{Store}$ denote the set of those stores from which the action $Act(t, l)$ is enabled, that is, $\mathit{enabled}(t, l) = \{s \mid \exists s', l'.\ (s, s', l') \in Act(t, l)\}$. The action $Act(t, l)$ is *safe* for a variable x if it blocks until no other thread has exclusive access to x. Formally, the action $Act(t, l)$ is safe *w.r.t.* x iff $\mathit{enabled}(t, l) \cap E(u, x) = \emptyset$ for all $u \neq t$. If $Act(t, l)$ is not safe for a variable $x \in \alpha(t, l)$, that action could violate the exclusive predicate for x if it is performed in an incorrect state, where another thread has exclusive access to x. To ensure this violation does not happen, we require that if $Act(t, l)$ is not safe for $x \in \alpha(t, l)$, then whenever this action is performed, the thread must have exclusive access to x. The set $\delta(t, l) \subseteq \mathit{Store}$ for $Act(t, l)$ describes states that satisfy this restriction.

$$\delta(t, l) \stackrel{\text{def}}{=} \bigwedge_{x \in \alpha(t,l)} \delta(t, l, x)$$

$$\delta(t, l, x) \stackrel{\text{def}}{=} \begin{cases} \mathit{true}, & \text{if } Act(t, l) \text{ is safe } w.r.t.\ x, \\ E(t, x), & \text{otherwise.} \end{cases}$$

To assist in identifying transactions, the instrumentation process extends the local store of each thread with a boolean *phase* variable which indicates whether that thread is currently executing in the pre- or post-commit phase of a transaction.

$$\begin{array}{lll} p \in \mathit{boolean} & & \text{(phase variables)} \\ \ell \in \mathit{Loc}^{\#} & = \mathit{Loc} \times \mathit{boolean} & \text{(program counter and phase variable)} \\ \mathit{ls} \in \mathit{Locs}^{\#} & = \mathit{Tid} \rightarrow \mathit{Loc}^{\#} & \text{(new local stores)} \end{array}$$

Thus, a state of $P^{\#}$ is a tuple (s, ls). If $\mathit{ls}(t) = \langle l, \mathit{true} \rangle$, then thread t is in the pre-phase of some transaction. If $\mathit{ls}(t) = \langle l, \mathit{false} \rangle$, then thread t is either in the post-phase of some transaction or at the beginning or end of a transaction. The initial store of $P^{\#}$ is (s_0, ls_0), where $\mathit{ls}_0(t) = \langle \mathit{Init}, \mathit{false} \rangle$ for all $t \in \mathit{Tid}$. A consequence of our instrumentation code is that in any reachable state, only one of the phase variables will be *true*, and hence these phase variables only extend the reachable state space by a factor of $|Tid|$, rather than $2^{|Tid|}$ as might be expected.

We instrument the action $Act(t, l)$ to transform it into $Act^{\#}(t, \langle l, p\rangle) \subseteq Store \times (Store \times Loc^{\#})$ as follows. The instrumentation adds an assertion that the restriction $\delta(t, l)$ described above holds. If the assertion $\delta(t, l)$ fails, then thread t goes wrong.

$$
\begin{array}{lcl}
(s, s', \langle l', p'\rangle) \in Act^{\#}(t, \langle l, p\rangle) & \stackrel{\text{def}}{=} & \vee \; \wedge \; s \notin \delta(t, l) \\
& & \quad\; \wedge \; s' = s \\
& & \quad\; \wedge \; l' = Wrong \\
& & \quad\; \wedge \; p' = p \\
& & \vee \; \wedge \; s \in \delta(t, l) \\
& & \quad\; \wedge \; (s, s', l') \in Act(t, l) \\
& & \quad\; \wedge \; p' = \begin{pmatrix} \wedge \; s' \in E(t, l) \\ \wedge \; (p \vee l = Init \vee s \notin E(t, l)) \end{pmatrix}
\end{array}
$$

In addition, the instrumentation code updates the phase variable of thread t as thread t moves from one phase of a transaction to another. The intuition behind the phase variable update is as follows. Suppose thread t is in the pre-commit phase before $Act(t, \ell)$ happens. If thread t has exclusive access to all accessed variables in the post-state of the action, then this action is a right mover and the transaction remains in the pre-commit phase. Otherwise, the transaction moves into the post-commit phase.

Conversely, suppose thread t is in the post-commit phase before $Act(t, \ell)$ happens. If thread t has exclusive access to all accessed variables in the pre-state of the action, then this action is a left mover and therefore the transaction continues to remain in the post-commit phase. Otherwise, the current transaction has ended and this action begins a new transaction. If thread t has exclusive access to all accessed variables in the post-state of the action, then this action is a right mover and the new transaction is in the pre-commit phase. Otherwise, the new transaction moves into the post-commit phase.

Finally, we define the transition relation $\rightarrow^{t}_{P\#}$ as follows.

$$
\begin{array}{lcl}
(s, \mathit{ls}) \rightarrow^{t}_{P\#} (s', \mathit{ls}') & \stackrel{\text{def}}{=} & \exists \ell \in Loc^{\#}. \wedge \; \ell = \mathit{ls}(t) \\
& & \qquad\qquad\;\; \wedge \; \mathit{ls}' = \mathit{ls}[t := \ell'] \\
& & \qquad\qquad\;\; \wedge \; (s, s', \ell') \in Act^{\#}(t, \ell)
\end{array}
$$

The program $P^{\#}$ goes wrong if there is a state (s, ls) and a thread t such that $(s_0, \mathit{ls}_0) \rightarrow^{*}_{P\#} (s, \mathit{ls})$ and $\mathit{ls}(t) = \langle Wrong, p\rangle$ for some p.

The program $P^{\#}$ has two properties. First, it goes wrong at least as often as the program P. Second, it has been designed so a transaction-based model checking algorithm can be easily formulated.

Our algorithm is parameterized by a set of partitions $(R(t), L(t), N(t))$ of $State$, one for each thread $t \in Tid$. Informally, $N(t)$ means that thread t is not in a transaction, and $R(t)$ and $L(t)$ mean that thread t is in the right-mover (or pre-commit) and left-mover (or post-commit) parts of a transaction, respectively. Each partition must satisfy the following three properties:

A. $R(t) = \{(s, \mathit{ls}) \mid \exists l.\ \mathit{ls}(t) = \langle l, \mathit{true}\rangle \wedge l \notin \{\mathit{Init}, \mathit{Wrong}\}\}$.

B. $L(t) \subseteq \{(s, \mathit{ls}) \mid \exists l.\ \mathit{ls}(t) = \langle l, \mathit{false}\rangle \wedge l \notin \{\mathit{Init}, \mathit{Wrong}\} \wedge s \in E(t, l)\}$.

C. For all $(s, \mathit{ls}) \in L(t)$, there is a state $(s', \mathit{ls}') \in N(t)$ such that $(s, \mathit{ls}) \rightarrow^t_{P\#} \cdots \rightarrow^t_{P\#} (s', \mathit{ls}')$.

Property A says that a thread is in the right-mover part of a transaction if its phase variable is *true* and the program counter is not *Init* or *Wrong*. Property B says that a thread is in the left-mover part of a transaction if its phase variable is *false*, the program counter is not *Init* or *Wrong*, and the thread has exclusive access to all variables accessed by the next action, that is, the next action of the thread is a left-mover. Property C ensures that the left-mover part of each transaction terminates. Given these partitions, the model checking algorithm simply computes the least fixpoint of the following set of rules.

Model checking reduced multithreaded programs

(INIT)
$$\frac{}{\mathcal{S}(s_0, \mathit{ls}_0)}$$

(STEP)
$$\frac{\mathcal{S}(s, \mathit{ls}) \quad \forall u \neq t.\ (s, \mathit{ls}) \in N(u) \quad (s, \mathit{ls}) \rightarrow^t_{P\#} (s', \mathit{ls}')}{\mathcal{S}(s', \mathit{ls}')}$$

The following theorem states the soundness of our approach.

Theorem 4.1 *Let P be a multithreaded program and let $P^{\#}$ be the instrumented version of program. For each $t \in \mathit{Tid}$, let $(R(t), L(t), N(t))$ be a partition of State satisfying conditions A, B and C. Then, the following statements hold.*

(i) *If P goes wrong, then $P^{\#}$ goes wrong.*

(ii) *If $P^{\#}$ goes wrong, then there is a state (s, ls) and a thread t such that $\mathcal{S}(s, \mathit{ls})$ and $\mathit{ls}(t) = \langle \mathit{Wrong}, p\rangle$ for some p.*

Proof Follows from Theorem A.1 and Lemmas B.1 and B.5 in the appendix.

Our transaction-based model checking algorithm uses the set $N(t)$ for each thread $t \in \mathit{Tid}$. The larger the set $N(t)$ the smaller the transactions inferred by our algorithm. In the limit, when the set $N(t) = \mathit{State}$, our algorithm reduces to the standard model checking algorithm. Therefore, subject to the conditions A, B, and C, we would like to pick the set $N(t)$ as small as possible, and consequently the sets $R(t)$ and $L(t)$ as large as possible. The set $R(t)$ is completely defined by condition A. The sets $L(t)$ and $N(t)$ must be chosen subject to conditions B and C. We can select these sets in an optimal manner by modifying Tarjan's depth-first search based algorithm for computing strongly-connected components of a graph [Tar72]. Of course, there are cheaper heuristics for performing this selection, e.g., ensuring that every cycle of states in $L(t)$ is broken by at least one state in $N(t)$.

4.3 Example

We now illustrate the program instrumentation defined in the last subsection by instrumenting the method `read` from Section 2. We show the code for `read` below, where we have made the program counter explicit. The local boolean variable `p` keeps track of the phase.

```
   int read() {
1:   acquire(mx); p := true;
2:   assert mx == tid; int tx = x; p := (p && mx == tid);
3:   acquire(my); p := true;
4:   assert my == tid; int ty = y; p := (p && my == tid);
5:   assert count == tx-ty; p := false;
6:   release(my); p := false;
7:   release(mx); p := false;
8:   return tx-ty; p := p;
9: }
```

We can make several simplifications in the instrumentation because the program uses only mutexes for synchronization. First, we need to add assertions only for the data access operations; the acquire and release operations on mutexes are safe by definition. Second, the phase update can be simplified. For lock acquire operations, the phase must be updated to true. For lock release operations, the phase must be updated to false. For data access operations, the locks held in the pre and post state are the same. The new phase is the conjunction of the old phase and the boolean value determining whether the mutex on the data accessed is held. The instrumentation for the action on line 5 is particularly interesting. Among the variables accessed on line 5, only `count` is global and its exclusive access predicate is *false*. Therefore, any action is safe *w.r.t.* `count` and consequently we do not introduce any assertion before the action. However, the phase variable `p` is updated to `false` since the exclusive access predicate *false* does not hold in the post-state.

Thus, the instrumentation in our method is simple and intuitive for mutex-based programs. At the same time, the flexibility afforded by the generality of exclusive access predicates makes our technique applicable to other programs with more complicated synchronization.

5 Application to lock-based programs

The transaction-based model checking algorithm of the previous section can accommodate a wide variety of synchronization mechanisms. To reduce the complexity of this analysis, we next specialize our approach to lock-based synchronization, which is the dominant synchronization mechanism in multi-threaded programs.

We classify variables as either *data variables* $z \in \mathit{DataVar}$ or *lock variables* $m \in \mathit{MutexVar}$. Lock variables are manipulated only by acquire and release operations. If the lock m is not held by any thread, then $m = 0$; if the lock is held by thread t, then $m = t$. The acquire operation blocks until m is 0 (unheld), and then updates m with the identifier of the current thread. Conversely, the release operation blocks until m is equal to the identifier of the current thread, and then sets m to 0. The acquire and release operations do not access other variables. A *data operation* only accesses data variables; it does not access any lock variables. We assume that each program operation $\mathit{Act}(t, l)$ is either a data operation, an acquire operation, or a release operation.

A thread has exclusive access to a lock variable only when it holds that lock:

$$E(t, m) \stackrel{\text{def}}{=} (m = t)$$

Clearly, the properties (1) and (2) on exclusive predicates hold for lock variables. In addition, all actions are safe for lock variables. Therefore, whenever $\mathit{Act}(t, l)$ is an acquire or a release operation on the lock m, we have:

$$\delta(t, l, m) = \mathit{true}$$

We assume that each data variable z has an associated set of protecting locks $M_z \subseteq \mathit{MutexVar}$. We say z is *unprotected* if $M_z = \emptyset$. A thread has exclusive access to z only when z is protected and the thread holds all the protecting locks:

$$E(t, z) \stackrel{\text{def}}{=} (M_z \neq \emptyset \wedge \forall m \in M_z.\ m = t)$$

The properties (1) and (2) on exclusive predicates also hold for data variables. Suppose $\mathit{Act}(t, l)$ is a data operation and z is a variable accessed by $\mathit{Act}(t, l)$. If z is an unprotected variable, then

$$\delta(t, l, z) = \mathit{true}$$

since all actions are safe for unprotected data variables. Otherwise, we have

$$\delta(t, l, z) = (\forall m \in M_z.\ m = t)$$

Based on these exclusive access predicates, we make the following observations about the various operations. We later define the action relation for the instrumented program based on these observations.

(i) If $\mathit{Act}(t, l)$ is a data operation and $(s, s', l') \in \mathit{Act}(t, l)$, then $s \in E(t, l)$ iff $s' \in E(t, l)$ iff $\forall z \in \alpha(t, l).\ (M_z \neq \emptyset \wedge \forall m \in M_z.\ s(m) = t)$.

(ii) If $\mathit{Act}(t, l)$ is an acquire operation and $(s, s', l') \in \mathit{Act}(t, l)$, then $s \notin E(t, l)$ and $s' \in E(t, l)$.

(iii) If $\mathit{Act}(t, l)$ is a release operation and $(s, s', l') \in \mathit{Act}(t, l)$, then $s' \notin E(t, l)$.

The action relation for the instrumented program is as follows.

$$
(s, s', \langle l', p' \rangle) \in Act^{\#}(t, \langle l, p \rangle) \stackrel{\text{def}}{=}
\begin{array}{l}
\vee \ \wedge \ Act(t,l) \text{ is a data operation} \\
\quad \wedge \ \forall z \in \alpha(t,l).\ \forall m \in M_z.\ s(m) = t \\
\quad \wedge \ (s, s', l') \in Act(t,l) \\
\quad \wedge \ p' = \begin{pmatrix} \wedge \ (p \vee l = Init) \\ \wedge \ \forall z \in \alpha(t,l).\ M_z \neq \emptyset \end{pmatrix} \\
\vee \ \wedge \ Act(t,l) \text{ is a data operation} \\
\quad \wedge \ \exists z \in \alpha(t,l).\ \exists m \in M_z.\ s(m) \neq t \\
\quad \wedge \ s' = s \\
\quad \wedge \ l' = Wrong \\
\quad \wedge \ p' = p \\
\vee \ \wedge \ Act(t,l) \text{ is an acquire operation} \\
\quad \wedge \ (s, s', l') \in Act(t,l) \\
\quad \wedge \ p' = true \\
\vee \ \wedge \ Act(t,l) \text{ is a release operation} \\
\quad \wedge \ (s, s', l') \in Act(t,l) \\
\quad \wedge \ p' = false
\end{array}
$$

Using these definitions, we can leverage the transaction-based algorithm of Section 4.2 to model check programs with lock-based synchronization.

5.1 Thread-local variables

We can extend this approach to handle thread-local variables as well as lock-protected variables in a straightforward manner. For example, we could introduce, for each thread t, a corresponding lock variable $m_t \in MutexVar$, and have thread t initially acquire the lock m_t. Each data variable z that is only used by thread t can then be given the protecting lock set $M_z = \{m_t\}$, yielding the exclusive predicate $E(u, z) \stackrel{\text{def}}{=} (m_t = u)$, which only holds if $u = t$. Thus, in this approach, thread t always has exclusive access to z.

To avoid the overhead of introducing the additional lock variables m_t, we could alternatively define, for each data variable z, a synchronization discipline

$$M_z \quad \in \quad 2^{MutexVar} \cup Tid$$

If $M_z \in Tid$, then z is local to thread with identifier M_z; otherwise M_z denotes a set of protecting locks for z as before. The corresponding exclusive predicate for z is:

$$
E(t, z) \stackrel{\text{def}}{=} \begin{cases} t = M_z, & \text{if } M_z \in Tid, \\ M_z \neq \emptyset \wedge \forall m \in M_z.\ m = t, & \text{otherwise.} \end{cases}
$$

5.2 Re-entrant locks

To handle re-entrant locks, for each lock variable m, we introduce a corresponding variable c_m that records the number of outstanding acquires on the lock m. The exclusive access predicate for c_m is:

$$E(t, c_m) \stackrel{\text{def}}{=} m = t$$

The variable c_m is initialized to 0. The operation $acquire(m)$ blocks until $m \in \{0, tid\}$ (where tid denotes the identifier of the current thread), and then atomically sets m to tid and increments c_m. The acquire operation is a right mover. The operation $release(m)$ blocks until $m = tid$. When $m = tid$, this operation decrements c_m, and if c_m becomes 0, it also sets m to 0. The release operation is a left mover.

5.3 Wait and notify

Our approach can also handle *wait* and *notify* operations. For each lock variable m, we introduce a corresponding variable w_m that records the *wait set* for m. This wait set is initialized to the empty set. The exclusive predicate for w_m is:

$$E(t, w_m) \stackrel{\text{def}}{=} m = t$$

The operation $wait(m)$ consists of a sequence of two atomic operations: the first blocks until $m = tid$, and then atomically sets m to 0 and adds tid to the wait set w_m; the second atomic operation blocks until $m = 0$ and $tid \notin w_m$, and atomically sets m to tid. Thus, $wait(m)$ consists of a left mover followed by a right mover.

The operation $notify(m)$ blocks until $m = tid$. Then, if w_m is nonempty, it removes some thread identifier from w_m; if w_m is empty, it does nothing. The operation $notifyAll(m)$ blocks until $m = tid$, and then removes all thread identifiers from w_m. Each of these notify operations is both a right mover and a left mover.

6 Inferring protecting locks

The previous section relies on the programmer to specify an appropriate set of protecting locks M_z for each data variable $z \in DataVar$. In this section, we extend our algorithm to infer these sets of protecting locks automatically.

Our algorithm maintains the set of reachable states in the variable $\mathcal{Q}$. It starts with the assumption that $M_z = MutexVar$ for each data variable $z \in DataVar$, and reduces M_z to contain only locks that are consistently held on every access to z. During transaction-based model checking, when an access to a data variable z is encountered, any lock not held by the current thread is removed from M_z. If the resulting set M_z is non-empty, then that access is considered protected and hence both a right and a left mover, and the current transaction continues in the same phase. If the set M_z is empty, then the access is an unprotected action that is neither a right nor a left mover, and the transaction either transitions to its post-commit phase, or terminates, if it was already in its post-commit phase.

Model checking with lock inference

Initially $\mathcal{Q} = \emptyset$ and $M_z = \mathit{MutexVar}$ for all data variables z

(INIT)
$$\frac{}{\mathcal{Q}(s_0, \mathit{ls}_0)}$$

(STEP)
$$\frac{\begin{array}{c}\mathcal{Q}(s, \mathit{ls}) \\ \forall u \neq t.\ (s, \mathit{ls}) \in N(u) \\ \mathit{ls}(t) = \langle l, p \rangle \\ \mathit{Act}(t, l) \text{ is a data operation} \\ (s, s', l') \in \mathit{Act}(t, l) \\ \forall z \in \alpha(t, l).\ \forall m \in M_z.\ s(m) = t \\ p' = ((p \vee l = \mathit{Init}) \wedge \forall z \in \alpha(t, l).\ M_z \neq \emptyset)\end{array}}{\mathcal{Q}(s', \mathit{ls}[t := \langle l', p' \rangle])}$$

(ACCESS)
$$\frac{\begin{array}{c}\mathcal{Q}(s, \mathit{ls}) \\ \forall u \neq t.\ (s, \mathit{ls}) \in N(u) \\ \mathit{ls}(t) = \langle l, p \rangle \\ \mathit{Act}(t, l) \text{ is a data operation} \\ (s, s', l') \in \mathit{Act}(t, l) \\ z \in \alpha(t, l) \\ m \in M_z \\ s(m) \neq t\end{array}}{\text{remove } m \text{ from } M_z}$$

(ACQUIRE)
$$\frac{\begin{array}{c}\mathcal{Q}(s, \mathit{ls}) \\ \forall u \neq t.\ (s, \mathit{ls}) \in N(u) \\ \mathit{ls}(t) = \langle l, p \rangle \\ \mathit{Act}(t, l) \text{ is an acquire operation} \\ (s, s', l') \in \mathit{Act}(t, l)\end{array}}{\mathcal{Q}(s', \mathit{ls}[t := \langle l', \mathit{true} \rangle])}$$

(RELEASE)
$$\frac{\begin{array}{c}\mathcal{Q}(s, \mathit{ls}) \\ \forall u \neq t.\ (s, \mathit{ls}) \in N(u) \\ \mathit{ls}(t) = \langle l, p \rangle \\ \mathit{Act}(t, l) \text{ is a release operation} \\ (s, s', l') \in \mathit{Act}(t, l)\end{array}}{\mathcal{Q}(s', \mathit{ls}[t := \langle l', \mathit{false} \rangle])}$$

Since all variables are initially protected, the model checking algorithm initially explores program executions using overly large transactions. As the protecting sets M_z are reduced, more data variables become unprotected and the derived predicate $N(t)$ becomes larger, with the result that the algorithm eventually considers transactions of suitably small granularity and explores enough context switches to ensure soundness.

Since $\mathcal{Q}$ is increasing and the the protecting sets M_z are decreasing, this algorithm always terminates. In addition, the algorithm for simultaneously inferring protecting locks and performing reduction-based model-checking is sound.

Theorem 6.1 *If P goes wrong, then* $\exists s, \mathit{ls}, t, p.\ \mathcal{Q}(s, \mathit{ls}) \wedge \mathit{ls}(t) = \langle \mathit{Wrong}, p \rangle$.

Proof sketch: Suppose

(i) the application of this algorithm to P yields reachable (reduced) state space $\mathcal{Q}$ and computed protecting sets M_z;

(ii) these protecting sets yield corresponding exclusive access predicates;

(iii) given P and these exclusive access predicates, the instrumentation algorithm of Section 4.2 yields the derived program $P^\#$; and

(iv) the transaction-based model checking algorithm of Section 4.2 yields reachable (reduced) state space $\mathcal{S}$ for $P^\#$.

Then $\mathcal{S} \subseteq \mathcal{Q}$. Thus, if P goes wrong, then by Theorem 4.1, $P^\#$ goes wrong, and hence $\exists s, \mathit{ls}, t, p.\ \mathcal{Q}(s, \mathit{ls}) \wedge \mathit{ls}(t) = \langle \mathit{Wrong}, p \rangle$.

7 Related work

The transaction-based model checking algorithm presented in this paper is based on the theory of reduction [Lip75], which was introduced by Lipton to reduce the intellectual complexity of reasoning about parallel programs. Reduction is based on commutativity between actions of different processes and distinguishes between right-commuting and left-commuting actions. Many researchers [Doe77,LS89,CL98,Mis01] have developed the theory of reduction to incorporate general safety and liveness properties.

Recently, Stoller and Cohen [SC03] have developed a model checking algorithm for multithreaded programs based on the theory of reduction. Their algorithm uses only left movers whereas our algorithm uses both right and left movers. At the same time, some of the hypotheses of our reduction theorem are stronger than the hypotheses of their reduction theorem. Thus, although our reduction theorem is less general, it applies to most common multithreaded programs to achieve larger transactions. One cost of using both right and left movers is that our algorithm does not directly catch deadlocks. However, typical multithreaded programs follow a locking discipline where locks are acquired in a partial order. A violation of such a partial order can be encoded as an assertion and our system can be used to check these assertions.

Stoller and Cohen [SC03] also run the lockset algorithm in parallel with the model checker to infer lock sets for shared variables. However, they restart the model checker every time the lockset is refined. In contrast, our algorithm is fully incremental. It starts with large locksets and transactions and both are gradually refined until a fixpoint is reached.

The class of techniques called partial-order reduction [Val90,Pel94,God96] also leverages commutativity between actions of different processes to reduce the explored state space during enumerative model checking. These techniques typically do not distinguish between right-commuting and left-commuting actions. Moreover, the commuting actions are inferred by a static analysis performed prior to model checking. These methods have mostly been applied to message-passing systems where static analysis is able to yield a large number of commuting action pairs. The algorithm in this paper distinguishes between right and left movers, infers them dynamically during model checking, and works on multithreaded programs, a domain in which static commutativity analysis is typically not precise enough. Recently, Dwyer et al. [DHRR03] have used static and dynamic object escape analysis and information about locks to perform a more precise commutativity analysis that improves the performance of partial-order techniques on shared-memory programs.

The notion of transactions is related to the correctness conditions of serializability [Pap86] for databases and linearizability [HW90] for concurrent software objects. The novelty of our work is in the automatic inference of transactions and their use to combat state-space explosion in model checking.

Static race-detection tools [FF00,BR01,BLR02,Gro03] check that accesses to data are protected by appropriate locks typically specified as program annotations. Using the scheme in Section 5, these annotations can be checked by our algorithm as well. In fact, our scheme is more general because it can check exclusive access predicates which are a generalization of locks.

Our scheme for inferring the locks protecting shared variables during model checking is inspired by the lockset algorithm implemented in the dynamic race-detection tool Eraser [SBN+97]. Eraser requires the programmer to provide test sequences, whereas in our scheme the model checker provides all possible test sequences.

8 Conclusions and future work

This paper presents a model checking algorithm for multithreaded software systems. This algorithm incorporates *reduction*, a promising technique that combats the state explosion problem of such systems by identifying transactions and avoiding needlessly scheduling one thread during a transaction of another thread. Our notion of reduction incorporates both left and right movers, and thus yields larger transactions and fewer contexts switches than previous techniques that only support left movers [SC03]. Our basic approach relies on the programmer to specify access restrictions or access predicates for program variables. This paper also presents a technique to avoid this programmer overhead by automatically inferring these access predicates for lock-based programs. This inference is an integral part of our transaction-based model checking algorithm.

Since this paper models each thread as a finite transition relation, several additional issues remain to be considered when model checking software. For example, program variables often range over essentially infinite domains such as integers, and threads normally rely on the presence of an unbounded stack. An important area for future work is to extend the model checking algorithm of this paper to tackle these issues, for example, via techniques such as predicate abstraction, counterexample-based predicate inference, and exploiting the regularity of stack frames. Another possible extension is to combine the techniques of this paper with thread-modular model checking [FQ03a].

Acknowledgments

We thank Scott Stoller and Yichen Xie for their careful reading of this paper. Their suggestions were very helpful in improving the presentation of our work.

References

[BLR02] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA 02: Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230. ACM Press, 2002.

[BR01] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA 01: Object-Oriented Programming, Systems, Languages, and Applications*, pages 56–69. ACM Press, 2001.

[CL98] E. Cohen and L. Lamport. Reduction in TLA. In *CONCUR 98: Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1998.

[DHRR03] M. B. Dwyer, J. Hatcliff, V. P. Ranganath, and Robby. Exploiting object escape and locking information in partial order reductions for concurrent object-oriented programs. Technical Report SAnToS-TR2003-1, Department of Computing and Information Sciences, Kansas State University, 2003.

[Doe77] T. W. Doeppner, Jr. Parallel program correctness through refinement. In *POPL 77: Principles of Programming Languages*, pages 155–169. ACM Press, 1977.

[FF00] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI 00: Programming Language Design and Implementation*, pages 219–232. ACM Press, 2000.

[FQ02] S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. Technical Note 01-2002, Williams College, December 2002.

[FQ03a] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN 03: Workshop on Model Checking Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 213–225. Springer-Verlag, 2003.

[FQ03b] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI 03: Programming Language Design and Implementation*, 2003. to appear.

[FQ03c] C. Flanagan and S. Qadeer. Types for atomicity. In *TLDI 03: Types in Language Design and Implementation*, pages 1–12. ACM Press, 2003.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Lecture Notes in Computer Science 1032. Springer-Verlag, 1996.

[Gro03] D. Grossman. Type-safe multithreading in Cyclone. In *TLDI 03: Types in Language Design and Implementation*, pages 13–25. ACM Press, 2003.

[HW90] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[Lam94] L. Lamport. How to write a long formula. Technical Report 119, DEC Systems Research Center, 1994.

[Lip75] R. J. Lipton. Reduction: A method of proving properties of parallel programs. In *Communications of the ACM*, volume 18:12, pages 717–721, 1975.

[LS89] L. Lamport and F. B. Schneider. Pretending atomicity. Research Report 44, DEC Systems Research Center, May 1989.

[Mis01] J. Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Springer-Verlag, 2001.

[Pap86] C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, 1986.

[Pel94] 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.

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

[SC03] S. D. Stoller and E. Cohen. Optimistic synchronization-based state-space reduction. In *TACAS 03: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 489–504. Springer-Verlag, 2003.

[Tar72] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[Val90] A. Valmari. A stubborn attack on state explosion. In R.P. Kurshan and E.M. Clarke, editors, *CAV 90: Computer Aided Verification*, Lecture Notes in Computer Science 531, pages 25–42. Springer-Verlag, 1990.

A Reduction theorem

In this section of the appendix, we present our reduction theorem. In the next section, we use the reduction theorem to prove that our transaction-based model checking algorithm is correct. We state the reduction theorem in terms of the following abstract domains.

Domains

$$\begin{array}{rll}
\sigma \in \mathit{State} & = \mathit{Store} \times \mathit{Locs} \\
\rho \in \mathit{Predicate} & \subseteq \mathit{State} \\
a, b \in \mathit{Action} & \subseteq \mathit{State} \times \mathit{State}
\end{array}$$

The action a *right-commutes* with the action b if for all $\sigma_1, \sigma_2, \sigma_3$, whenever $(\sigma_1, \sigma_2) \in a$ and $(\sigma_2, \sigma_3) \in b$, then there exists σ_2' such that $(\sigma_1, \sigma_2') \in b$ and $(\sigma_2', \sigma_3) \in a$. The action a *left-commutes* with the action b if b right-commutes with a. We define the *left restriction* $\rho \cdot a$ and the *right restriction* $a \cdot \rho$ of an action a with respect to a set of states ρ.

$$\rho \cdot a \stackrel{\text{def}}{=} \{(\sigma, \sigma') \in a \mid \sigma \in \rho\}$$
$$a \cdot \rho \stackrel{\text{def}}{=} \{(\sigma, \sigma') \in a \mid \sigma' \in \rho\}$$

Theorem A.1 *Let σ_0 be a state. For all $t \in \mathit{Tid}$, let $T(t)$ be an action, let $\mathcal{W}(t)$ be a set of states, and let $(\mathcal{R}(t), \mathcal{L}(t), \mathcal{N}(t))$ be a partition of the set of states. Suppose the following conditions hold.*

A1. $\sigma_0 \in \mathcal{N}(t)$ *and* $\mathcal{W}(t) \subseteq \mathcal{N}(t)$.

A2. $(\mathcal{L}(t) \cdot T(t) \cdot \mathcal{R}(t))$ *is empty.*

A3. For all $u \neq t$, $(T(t) \cdot \mathcal{R}(t))$ *right-commutes with* $T(u)$.

A4. For all $u \neq t$, $(\mathcal{L}(t) \cdot T(t))$ *left-commutes with* $T(u)$.

A5. For all $u \neq t$, *if* $(\sigma, \sigma') \in T(t)$, *then* $\sigma \in \mathcal{R}(u) \Leftrightarrow \sigma' \in \mathcal{R}(u)$, $\sigma \in \mathcal{L}(u) \Leftrightarrow \sigma' \in \mathcal{L}(u)$, *and* $\sigma \in \mathcal{W}(u) \Leftrightarrow \sigma' \in \mathcal{W}(u)$.

A6. For all $\sigma \in \mathcal{L}(t)$, *we have* $(\sigma, \sigma') \in T(t)^*$ *for some* $\sigma' \in \mathcal{N}(t)$.

Let $\hookrightarrow = \exists t.\ T(t)$ *and* $\hookrightarrow_c = \exists t.\ (\forall u \neq t.\ \mathcal{N}(u)) \cdot T(t)$. *If* $\sigma_0 \hookrightarrow^* \sigma$ *and* $\sigma \in \mathcal{W}(t)$ *for some* $t \in \mathit{Tid}$, *then there is* σ' *such that* $\sigma_0 \hookrightarrow_c^* \sigma'$ *and* $\sigma' \in \mathcal{W}(t)$.

Proof The proof is similar to the proof of the reduction theorem in our earlier paper [FQ03c].

B Transactions

This section refers to definitions and concepts introduced in Sections 3 and 4. We prove two main results. First, we show that if the program P goes wrong, then so does the instrumented program $P^{\#}$. Second, we define the set of erroneous stores $W(t)$ of thread $t \in \mathit{Tid}$ to be the set $\{(s, \ell s) \mid \exists p.\ \ell s(t) = \langle \mathit{Wrong}, p\rangle\}$. We show that the conditions A1-A6 of Theorem A.1 are satisfied if we substitute the state $(s_0, \ell s_0)$ for σ_0, the relation $\rightarrow^t_{P\#}$ for $T(t)$, the partition $(R(t), L(t), N(t))$ for $(\mathcal{R}(t), \mathcal{L}(t), \mathcal{N}(t))$, and the set $W(t)$ for $\mathcal{W}(t)$. The statement of Theorem A.1 immediately allows us to conclude the soundness of the transaction-based model checking algorithm applied to the instrumented program $P^{\#}$. Thus, combining the first and the second result, we get a sound model checking algorithm for P. We use the following definitions in the remainder of this section.

$$
\begin{array}{rcl}
((s, ls), (s', ls')) \in \Gamma(t, l) & \stackrel{\text{def}}{=} & \wedge\ ls(t) = l \\
 & & \wedge\ ls' = ls[t := ls'(t)] \\
 & & \wedge\ (s, s', ls'(t)) \in \mathit{Act}(t, l) \\
((s, \ell s), (s', \ell s')) \in \Gamma^{\#}(t, \ell) & \stackrel{\text{def}}{=} & \wedge\ \ell s(t) = \ell \\
 & & \wedge\ \ell s' = ls[t := \ell s'(t)] \\
 & & \wedge\ (s, s', \ell s'(t)) \in \mathit{Act}^{\#}(t, l)
\end{array}
$$

Lemma B.1 *If P goes wrong, then $P^{\#}$ goes wrong.*

Proof We do the proof by induction on the following inductive hypothesis.

Inductive hypothesis:
Let $n \geq 0$ and $(s_0, ls_0) \rightarrow_P (s_1, ls_1) \rightarrow_P \ldots \rightarrow_P (s_n, ls_n)$ be an execution of P. Then, there is $m \geq 0$ and an execution $(s_0, \ell s_0) \rightarrow_{P\#} (s_1, \ell s_1) \rightarrow_{P\#} \ldots \rightarrow_{P\#} (s_m, \ell s_m)$ of $P^{\#}$ such that either (1) $n = m$ and for each $i \in 0..n$, there is p_i such that $\ell s_i(t) = \langle ls_i(t), p_i\rangle$, or (2) $\exists t, p.\ \ell s_m(t) = \langle \mathit{Wrong}, p\rangle$.

Proof of inductive hypothesis:
Suppose $(s_0, ls_0) \rightarrow^*_P (s_n, ls_n) \rightarrow_P (s_{n+1}, ls_{n+1})$. Then, there is $m \geq 0$ and an execution $(s_0, \ell s_0) \rightarrow_{P\#} (s_1, \ell s_1) \rightarrow_{P\#} \ldots \rightarrow_{P\#} (s_m, \ell s_m)$ of $P^{\#}$ such that either (1) $n = m$ and for each $i \in 0..n$ and $t \in \mathit{Tid}$, there is p such that $\ell s_i(t) = \langle ls_i(t), p\rangle$, or (2) $\exists t, p.\ \ell s_m(t) = \langle \mathit{Wrong}, p\rangle$. In the second case, we are done immediately. In the first case, suppose $(s_n, ls_n), (s_{n+1}, ls_{n+1})) \in \Gamma(t, l)$ for some thread t and location $l \in \mathit{Loc}$. There are two cases:
Suppose $(s_n \notin \delta(t, l))$. There is p such that $((s_n, \ell s_n), (s_n, \ell s_n[t := \langle \mathit{Wrong}, p\rangle])) \in \Gamma^{\#}(t, l)$.
Suppose $(s_n \in \delta(t, l))$. There is ℓs_{n+1} such that for each $t \in \mathit{Tid}$, there is p_t such that $\ell s_{n+1}(t) = \langle ls_{n+1}(t), p_t\rangle$ and $((s_n, \ell s_n), (s_{n+1}, \ell s_{n+1})) \in \Gamma^{\#}(t, l)$.

Lemma B.2 *For all $t \in Tid$, $l \in Loc$, and $x \in \alpha(t,l)$, if $(s, s', l') \in Act(t,l)$ and $s \in \delta(t,l)$, then $s \notin E(u,x)$ for all $u \neq t$.*

Proof Pick $u \neq t$. We show that if $(s, s', l') \in Act(t,l)$ and $s \in \delta(t,l)$, then $s \notin E(u,x)$. If $s \in \delta(t,l)$, then $s \in \delta(t,l,x)$ for all $x \in \alpha(t,l)$. Therefore, for each $x \in \alpha(t,l)$, either $enabled(t,l) \cap E(u,x) = \emptyset$ or $s \in E(t,x)$. In the first case, since $s \in enabled(t,l)$ we get $s \notin E(u,x)$. In the second case, since $E(t,x) \cap E(u,x) = \emptyset$ we get $s \notin E(u,x)$.

Lemma B.3 *Let $t, u \in Tid$ be two different thread identifiers and $l, m \in Loc$ be two locations. If $((s_1, \ell s_1), (s_2, \ell s_2)) \in (\Gamma^\#(t,l) \cdot R(t))$ and $((s_2, \ell s_2), (s_3, \ell s_3)) \in \Gamma^\#(u,m)$, then there is $(s_4, \ell s_4)$ such that $((s_1, \ell s_1), (s_4, \ell s_4)) \in \Gamma^\#(u,m)$ and $((s_4, \ell s_4), (s_3, \ell s_3)) \in (\Gamma^\#(t,l) \cdot R(t))$.*

Proof Let $\ell s_1(t) = \langle l, p \rangle$ and $\ell s_2(t) = \ell s_3(t) = \langle l', p' \rangle$. Let $\ell s_1(u) = \ell s_2(u) = \langle m, q \rangle$ and $\ell s_3(u) = \langle m', q' \rangle$. Then, we have $(s_1, s_2, \langle l', p' \rangle) \in Act^\#(t, \langle l, p \rangle)$ and $(s_2, s_3, \langle m', q' \rangle) \in Act^\#(u, \langle m, q \rangle)$. From the definition of $R(t)$ and $Act^\#(t, \langle l, p \rangle)$ and the fact that $(s_2, \ell s_2) \in R(t)$, we conclude that $s_1 \in \delta(t,l)$ and $s_2 \in E(t,l)$. We do a case analysis on the condition $s_2 \in \delta(u,m)$.

$[s_2 \notin \delta(u,m)]$. In this case, we have $s_3 = s_2$, $\ell s_3 = \ell s_2[u := \langle Wrong, p \rangle]$. Since $s_2 \notin \delta(u,m)$, we get that $s_2 \notin E(u,x)$ for some $x \in \alpha(u,m)$. Since $s_1 \in E(u,x)$ iff $s_2 \in E(u,x)$, we get $s_1 \notin E(u,x)$. Thus, we get that $s_1 \notin \delta(u,m)$. Therefore, we have $((s_1, \ell s_1), (s_1, \ell s_1[u := \langle Wrong, p \rangle])) \in \Gamma^\#(u,m)$ and $((s_1, \ell s_1[u := \langle Wrong, p \rangle]), (s_2, \ell s_2[u := \langle Wrong, p \rangle])) \in \Gamma^\#(t,l) \cdot R(t)$. Let $s_4 = s_1$ and $\ell s_4 = \ell s_1[u := \langle Wrong, p \rangle]$. Since $(s_2, \ell s_2[u := \langle Wrong, p \rangle]) = (s_3, \ell s_3)$, we are done.

$[s_2 \in \delta(u,m)]$. From Lemma B.2, we have $s_2 \notin E(t,x)$ for all $x \in \alpha(u,m)$. But we know that $s_2 \in E(t,y)$ for all $y \in \alpha(t,l)$. Therefore $\alpha(t,l) \cap \alpha(u,m) = \emptyset$. Let $s_4 = s_1[\alpha(u,m) := s_3(\alpha(u,m))]$ and $\ell s_4 = \ell s_1[u := \ell s_3(u)]$. Then s_4 can also be written as $s_3[\alpha(t,l) := s_1(\alpha(t,l))]$. We show that the actions by thread t and u commute in several steps. We first show that $s_1 \in \delta(u,m)$. Since $s_2 \in \delta(u,m)$, we know that for all $x \in \alpha(u,m)$, either $Act(u,m)$ is safe w.r.t. x or $s_2 \in E(u,x)$. We know that $s_1 \in E(u,x)$ iff $s_2 \in E(u,x)$ for all $x \in Var$. Therefore $s_1 \in \delta(u,m)$. Therefore, we get that $(s_2, s_3, m') \in Act(u,m)$. We next show that $((s_1, \ell s_1), (s_4, \ell s_4)) \in \Gamma^\#(u,m)$. Since no variable in $\alpha(t,l)$ is accessed by $Act(u,m)$, we have

$$
\begin{aligned}
&(s_2, s_3, m') \in Act(u,m) \;\Leftrightarrow \\
&(s_2[\alpha(t,l) := s_1(\alpha(t,l))], s_3[\alpha(t,l) := s_1(\alpha(t,l))], m') \in Act(u,m) \;\Leftrightarrow \\
&(s_1, s_4, m') \in Act(u,m)
\end{aligned}
$$

Thus, we get that $s_1 \in E(t,x)$ iff $s_4 \in E(t,x)$ for all $x \in Var$. Since no variable in $\alpha(u,m)$ is accessed by $Act(t,l)$, we have

$$
\begin{aligned}
&(s_1, s_2, l') \in Act(t,l) \;\Leftrightarrow \\
&(s_1[\alpha(u,m) := s_3(\alpha(u,m))], s_2[\alpha(u,m) := s_3(\alpha(u,m))], l') \in Act(t,l) \;\Leftrightarrow \\
&(s_4, s_3, l') \in Act(t,l)
\end{aligned}
$$

Thus, we get that $s_4 \in E(u,x)$ iff $s_3 \in E(u,x)$ for all $x \in \mathit{Var}$. Since $s_1, s_2 \in \delta(u,m)$, $s_1 \in E(u,x)$ iff $s_2 \in E(u,x)$ for all $x \in \mathit{Var}$, $s_4 \in E(u,x)$ iff $s_3 \in E(u,x)$ for all $x \in \mathit{Var}$, and $(s_2, s_3, m') \in \mathit{Act}(u,m)$ iff $(s_1, s_4, m') \in \mathit{Act}(u,m)$, we get that $(s_2, s_3, \langle m', q'\rangle) \in \mathit{Act}(u, \langle m, q\rangle)$ iff $(s_1, s_4, \langle m', q'\rangle) \in \mathit{Act}(u, \langle m, q\rangle)$. Thus, we get $((s_1, \mathit{ls}_1), (s_4, \mathit{ls}_4)) \in \Gamma^{\#}(u,m)$. We now show that $s_4 \in \delta(t,l)$. Since $s_1 \in \delta(t,l)$, we know that for all $x \in \alpha(t,l)$, either $\mathit{Act}(t,l)$ is safe w.r.t. x or $s_1 \in E(t,x)$. We know that $s_1 \in E(t,x)$ iff $s_4 \in E(t,x)$ for all $x \in \mathit{Var}$. Therefore $s_4 \in \delta(t,l)$. Since $s_1, s_4 \in \delta(t,l)$, $s_1 \in E(t,x)$ iff $s_4 \in E(t,x)$ for all $x \in \mathit{Var}$, $s_2 \in E(t,x)$ iff $s_3 \in E(t,x)$ for all $x \in \mathit{Var}$, and $(s_1, s_2, l') \in \mathit{Act}(t,l)$ iff $(s_4, s_3, l') \in \mathit{Act}(t,l)$, we get that $(s_1, s_2, \langle l', p'\rangle) \in \mathit{Act}(t, \langle l, p\rangle)$ iff $(s_4, s_3, \langle l', p'\rangle) \in \mathit{Act}(t, \langle l, p\rangle)$. Thus, we get $((s_4, \mathit{ls}_4), (s_3, \mathit{ls}_3)) \in \Gamma^{\#}(t,l)$. Also, since $\mathit{ls}_3(t) = \mathit{ls}_2(t)$, we get $(s_3, \mathit{ls}_3) \in R(t)$.

Lemma B.4 *Let $t, u \in \mathit{Tid}$ be two different thread identifiers and $l, m \in \mathit{Loc}$ be two locations. If $((s_1, \mathit{ls}_1), (s_2, \mathit{ls}_2)) \in \Gamma^{\#}(u,m)$ and $((s_2, \mathit{ls}_2), (s_3, \mathit{ls}_3)) \in (L(t) \cdot \Gamma^{\#}(t,l))$, then there is (s_4, ls_4) such that $((s_1, \mathit{ls}_1), (s_4, \mathit{ls}_4)) \in (L(t) \cdot \Gamma^{\#}(t,a))$ and $((s_4, \mathit{ls}_4), (s_3, \mathit{ls}_3)) \in \Gamma^{\#}(u,b)$.*

Proof Similar to the proof of Lemma B.3.

Lemma B.5 *For each $t \in \mathit{Tid}$, let $(R(t), L(t), N(t))$ be a partition of the set of states satisfying conditions A, B, and C, and let $W(t) = \{(s, \mathit{ls}) \mid \exists p.\ \mathit{ls}(t) = \langle \mathit{Wrong}, p\rangle\}$. Then the following conditions hold.*

(i) *$(s_0, \mathit{ls}_0) \in N(t)$ and $W(t) \subseteq N(t)$.*

(ii) *$(L(t) \cdot \rightarrow^t_{P\#} \cdot R(t))$ is empty.*

(iii) *For all $u \neq t$, $(\rightarrow^t_{P\#} \cdot R(t))$ right-commutes with $\rightarrow^u_{P\#}$.*

(iv) *For all $u \neq t$, $(L(t) \cdot \rightarrow^t_{P\#})$ left-commutes with $\rightarrow^u_{P\#}$.*

(v) *For all $u \neq t$, if $((s, \mathit{ls}), (s', \mathit{ls}')) \in \rightarrow^t_{P\#}$, then $(s, \mathit{ls}) \in R(u) \Leftrightarrow (s', \mathit{ls}') \in R(u)$, $(s, \mathit{ls}) \in L(u) \Leftrightarrow (s', \mathit{ls}') \in L(u)$, and $(s, \mathit{ls}) \in W(u) \Leftrightarrow (s', \mathit{ls}') \in W(u)$.*

(vi) *For all $(s, \mathit{ls}) \in L(t)$, we have $((s, \mathit{ls}), (s', \mathit{ls}')) \in (\rightarrow^t_{P\#})^*$ for some $(s', \mathit{ls}') \in N(t)$.*

Proof We prove each condition separately.

(i) Obvious from the definition of $R(t)$, $L(t)$, and $N(t)$.

(ii) Suppose $((s, \mathit{ls}), (s', \mathit{ls}')) \in T^{\#}(t)$, $(s, \mathit{ls}) \in L(t)$, and $(s', \mathit{ls}') \in R(t)$. Therefore there are $l, l' \in \mathit{Loc}$ such that $((s, \mathit{ls}), (s', \mathit{ls}')) \in \Gamma^{\#}(t,l)$, $\mathit{ls}(t) = \langle l, \mathit{false}\rangle$, $\mathit{ls}'(t) = \langle l', \mathit{true}\rangle$, and $l' \neq \mathit{Wrong}$. From the definition of $\mathit{Act}^{\#}(t,l)$, we get that $s \notin E(t,l)$. At the same time, from the definition of $L(t)$ we get that $s \in E(t,l)$. Thus, we arrive at a contradiction.

(iii) Suppose $((s_1, \mathit{ls}_1), (s_2, \mathit{ls}_2)) \in T^{\#}(t) \cdot R(t)$ and $((s_2, \mathit{ls}_2), (s_3, \mathit{ls}_3)) \in T^{\#}(u)$. Then we have $((s_1, \mathit{ls}_1), (s_2, \mathit{ls}_2)) \in \Gamma^{\#}(t,l) \cdot R(t)$ for some $l \in \mathit{Loc}$ and $((s_2, \mathit{ls}_2), (s_3, \mathit{ls}_3)) \in \Gamma^{\#}(u,m)$ for some $m \in \mathit{Loc}$. From Lemma B.3, we get (s_4, ls_4) such that $((s_1, \mathit{ls}_1), (s_4, \mathit{ls}_4)) \in \Gamma^{\#}(u,m)$ and $((s_4, \mathit{ls}_4), (s_3, \mathit{ls}_3)) \in$

$\Gamma^{\#}(t, l) \cdot R(t)$.

(iv) Suppose $((s_1, \mathit{ls}_1), (s_2, \mathit{ls}_2)) \in T^{\#}(u)$ and $((s_2, \mathit{ls}_2), (s_3, \mathit{ls}_3)) \in L(t) \cdot T^{\#}(t)$. Then we have $((s_2, \mathit{ls}_2), (s_3, \mathit{ls}_3)) \in L(t) \cdot \Gamma^{\#}(t, l)$ for some $l \in \mathit{Loc}$ and $((s_1, \mathit{ls}_1), (s_2, \mathit{ls}_2)) \in \Gamma^{\#}(u, m)$ for some $m \in \mathit{Loc}$. From Lemma B.4, we get a state (s_4, ls_4) such that $((s_1, \mathit{ls}_1), (s_4, \mathit{ls}_4)) \in L(t) \cdot \Gamma^{\#}(t, l)$ and $((s_4, \mathit{ls}_4), (s_3, \mathit{ls}_3)) \in \Gamma^{\#}(u, m)$.

(v) Suppose $((s, \mathit{ls}), (s', \mathit{ls}')) \in T^{\#}(u)$ and $t \neq u$. Then $((s, \mathit{ls}), (s', \mathit{ls}')) \in \Gamma^{\#}(u, m)$ for some $m \in \mathit{Loc}$. From the definition of $\Gamma^{\#}(u, m)$, we get $\mathit{ls}(t) = \mathit{ls}'(t)$ and either $s = s'$ or $((s, \mathit{ls}(u)), (s', \mathit{ls}'(u))) \in \mathit{Act}(u, m)$. Since $u \neq t$, we have $s \in E(t, x)$ iff $s' \in E(t, x)$ for all $x \in \mathit{Var}$. Thus, we get that $(s, \mathit{ls}) \in R(t)$ iff $(s', \mathit{ls}') \in R(t)$, $(s, \mathit{ls}) \in W(t)$ iff $(s', \mathit{ls}') \in W(t)$, and $(s, \mathit{ls}) \in L(t)$ iff $(s', \mathit{ls}') \in L(t)$.

(vi) Follows from condition C.