A Calculus of Atomic Actions

Tayfun Elmas
Koç University, İstanbul, Turkey
telmas@ku.edu.tr

Shaz Qadeer
Microsoft Research, Redmond, WA
qadeer@microsoft.com

Serdar Tasiran
Koç University, İstanbul, Turkey
stasiran@ku.edu.tr

Abstract

We present a proof calculus and method for the static verification of assertions and procedure specifications in shared-memory concurrent programs. The key idea in our approach is to use atomicity as a proof tool and to simplify the verification of assertions by rewriting programs to consist of larger atomic actions. We propose a novel, iterative proof style in which alternating use of abstraction and reduction is exploited to compute larger atomic code blocks in a sound manner. This makes possible the verification of assertions in the transformed program by simple sequential reasoning within atomic blocks, or significantly simplified application of existing concurrent program verification techniques such as the Owicki-Gries or rely-guarantee methods. Our method facilitates a clean separation of concerns where at each phase of the proof, the user worries only about only either the sequential properties or the concurrency control mechanisms in the program. We implemented our method in a tool called QED. We demonstrate the simplicity and effectiveness of our approach on a number of benchmarks including ones with intricate concurrency protocols.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification — assertion checkers, correctness proofs, formal methods; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs — assertions, invariants, pre- and post-conditions, mechanical verification; D.1.3 [*Programming Techniques*]: Concurrent Programming — parallel programming

General Terms Languages, Theory, Verification

Keywords Concurrent Programs, Atomicity, Reduction, Abstraction

1. Introduction

This paper is concerned with the problem of statically verifying the (partial) correctness of shared-memory multithreaded programs. This problem is undecidable and, in theory, no harder than the problem of verifying single-threaded programs. In practice, however, it is significantly more difficult to verify multithreaded programs. For single-threaded programs, the undecidability of program verification is circumvented by the use of contracts–pre-conditions, post-conditions, and loop invariants– to decompose the problem into manageable pieces. These contracts need to refer only to the locally visible part of the program state, and can usually be stated with little difficulty. Reasoning about multithreaded programs, on the other hand, requires significantly more intellectual effort. For example, invariant-based reasoning [1, 21] requires for each line[1] of the program an annotation that is guaranteed to be stable under interference from other threads. Writing such a specification is challenging because of the need to consider the effect of all thread interleavings. The resulting annotations in each thread are often non-local and complicated because they refer to the private state of other threads. The rely-guarantee approach [16] provides more flexibility in specifying the interference from the environment but the complexity of the required annotations is still significant.

The fundamental difficulty in reasoning about multithreaded programs is the need to reason about concurrent execution of fine-grained atomic actions. In this paper, we introduce an iterative approach to proving assertions in multithreaded programs aimed at circumventing this difficulty. In our approach, in each step of the proof, we rewrite the program by locally transforming its atomic actions to obtain a simpler program. Each rewrite performs one of two different kinds of transformations—abstraction and reduction. Abstraction replaces an atomic action with a more relaxed atomic action allowing more behaviors. Abstraction transformations include making shared variable reads or writes non-deterministic, and prefacing atomic statements with extra assertions. Reduction [18] replaces a compound statement consisting of several atomic actions with a single atomic action if certain non-interference conditions hold. This transformation has the effect of increasing the granularity of the atomic actions in the program. Abstraction and reduction preserve or expand the set of behaviors of the program, so that assertions proved at the end of a sequence of transformations are valid in the original program.

While reduction and abstraction have been studied in isolation, the iterative and alternating application of abstraction and reduction is a distinguishing and essential aspect of our method. Abstraction and reduction are symbiotic. Reduction creates coarse-grained actions from fine-grained actions and allows a subsequent abstraction step to summarize the entire calculation much as preconditions and post-conditions summarize the behavior of a procedure in a single-threaded program. Conversely, suitably abstracting an atomic action allows us to reason that it does not interfere with other atomic actions, and later application of reduction are able to merge it with other actions. The examples in Sections 2 and 6 show that the combined iterative application of these techniques is a surprisingly powerful proof method able to prove intricate examples correct by a sequence of simple transformations and few annotations. In most cases, we are able to simplify the program enough so that assertions in the final program can be validated by purely sequential reasoning within a single abstract atomic action. In other cases, the atomic actions in the program become large enough for

[1] For brevity, we use the word “line” to refer to the granularity of atomically executed statements.

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM 978-1-60558-379-2/09/01…$5.00

a straightforward application of existing techniques such as rely-guarantee reasoning.

Another distinguishing feature of our approach is the possibility of introducing assertions at any point during the sequence of transformations while deferring their proof until later, when large enough atomic blocks make their proof easy. In our framework, annotating any atomic statement with any assertion is a valid program abstraction. The interpretation is that the action is relaxed so that if the assertion is violated, the execution is made to "go wrong." Annotating an action with an assertion, e.g. one that indicates that it is not simultaneously enabled with another action, may enable further reduction steps. In other proof methods, when an additional assertion is introduced, one is forced to prove that it is valid and preserved under interference by other threads. In our approach, we use the introduced assertion without first proving it and take further proof steps that simplify the program. Yet, the soundness of our method is not compromised as long as all assertions are proved eventually.

We have implemented our verification method in a tool called QED. Our tool accepts as input a multithreaded program written in an extension of the Boogie programming language [2] and a proof script containing a sequence of proof commands. A proof command is used for one of two purposes. First, it may provide a high-level tactic for rewriting the input program using abstraction, reduction or a combination of the two. Second, it may provide a concise specification of the behavior of the current version of the program; common specifications include locking protocols and data invariants. After executing each step in the proof script, QED allows the user to examine the resulting program, intercept the proof, and give new commands. The tool automatically generates the verification conditions justifying each step of the proof and verifies them using Z3 [6], a state-of-the-art solver for satisfiability-modulo-theories.

We have evaluated QED by verifying a number of multithreaded programs with varying degree of synchronization complexity. These examples include programs using fine-grained locking and non-blocking data structures. We have found that the iterative approach embodied in QED provides a simple and convenient way of communicating to the verifier the programmer's understanding of the computation and synchronization in the program. The proofs in our method are invariably simpler and more intuitive than the proofs based on existing approaches.

To summarize, this paper makes the following contributions:

- A novel proof technique for multithreaded programs, based upon rewriting of the input program iteratively using abstraction and reduction, producing in the limit a program that can be verified by sequential reasoning methods.
- A tool QED that implements our proof method using a set of intuitive, concise, and machine-checked proof commands.
- Evaluation of our technique and tool on a variety of small to medium-sized multithreaded programs.

2. Motivating examples

In this section, we provide an overview of our method using several examples. We begin by illustrating reduction and abstraction and, in Section 2.1, present a nontrivial interaction between them. In these examples, each line of code performs at most one access to a global variable. For example, we split the increment of x in Figure 1 and the assignment of newsize to currsize (lines 15–16) in Figure 4 into multiple lines. We use if(*) and while(*) to denote nondeterministic choice. The statements assume e and assert e cause the execution to block or go wrong, respectively, if e evaluates to false; otherwise, they are equivalent to a skip statement.

```
void inc() {                void inc() {
  int t;                      int t;
  acquire(lock);              [havoc t; x := x+1];
    t := x;                 }
    t := t+1;
    x := t;
  release(lock);
}
```

Figure 1. Lock-based atomic increment

The statement havoc x assigns x a nondeterminstic value of proper domain.

Reduction produces a single atomic action from the sequential composition of two atomic actions if either the first action is a right mover or the second action is a left mover. An action R commutes to the right of X if the effect of thread t executing R followed by a different thread u executing action X can be "simulated" by first thread u executing X followed by thread t executing R. If R commutes to the right of all actions in the program, it is said to be right mover. Common cases where R is a right mover are if R and X do not conflict (i.e., read or write any common variables), if R is non-deterministic (i.e. independent of the program store) or if R disables X. A left mover is defined similarly. Lock acquires are right movers and lock releases are left movers. Accesses to local variables and race-free accesses to shared variables are both right and left movers.

In Figure 1, the procedure inc on the left is a lock-based implementation for atomically incrementing a shared variable x. Through an iterative application of reduction, our method can transform the body of this procedure into a single atomic action as shown on the right and indicated by square brackets.

In Figure 2, the procedure inc on the left uses the CAS (Compare-And-Swap) primitive for a lock-free implementation of the same behavior. CAS(x,t,t+1) atomically compares the value of x with t, writing t+1 into x and returning true if the two are identical, and leaving x unchanged and returning false if the two are different. Note that this program has conflicting accesses to x that are simultaneously enabled, and is consequently more difficult to reason about than the previous version. However, through an iterative application of abstraction and reduction, our method can just as easily show that this procedure atomically increments x; the sequence of transformations are shown from left to right in Figure 2.

First, we perform a simple transformation (as in [10]) that peels out the last iteration of the loop, thereby arriving at the version of inc in Figure 2(b). Next, we argue that it is a valid abstraction to replace the read of x in t := x with havoc t since this only increases the set of possible behaviors. We do not expect this abstraction to lead to additional assertion violations since the result of the read action is later verified by CAS(x,t,t+1); reading a value other than the actual value of x would simply lead to a failed CAS operation. We also abstract each unsuccessful execution of CAS(x,t,t+1) inside the loop with the statement skip to arrive at the version of inc in Figure 2(c).

In the version of inc in Figure 2(c), every operation except for the last one accesses only local state and is consequently both a right and a left mover. Therefore, we can use reduction to summarize the entire loop with a single havoc t statement to arrive at the version of inc in Figure 2(d). Finally, we reduce the resulting sequential composition to arrive at a single atomic action in the version of inc in Figure 2(e).

Figure 3(a) shows procedure add, a client of inc. Procedure add has a parameter n required to be at least 0 and calls inc repeatedly in a loop n times. Once the procedure inc has been transformed into an atomic increment of x, we can reason about add by replacing the call to inc with an atomic increment of x to ob-

```
void inc() {
  int t;
  while (true) {
    t := x;
    if (CAS(x,t,t+1))
      break;
  }


}
```
(a)

```
void inc() {
  int t;
  while (*) {
    t := x;
    assume x!=t;
  }
  t := x;
  [assume x==t;
   x := t+1];
}
```
(b)

```
void inc() {
  int t;
  while (*) {
    havoc t;
    skip;
  }
  havoc t;
  [assume x==t;
   x := t+1];
}
```
(c)

```
void inc() {
  int t;
  havoc t;


  havoc t;
  [assume x==t;
   x := t+1];
}
```
(d)

```
void inc() {
  int t;


  [havoc t;
   x := x+1];



}
```
(e)

Figure 2. Lock-free atomic increment

```
void add(int n) {
  while (0<n) {
    inc();
    n := n-1;
  }
}
```
(a)

```
void add(int n) {
  while (0<n) {
    [x := x+1];
    n := n-1;
  }
}
```
(b)

```
void add(int n) {
  [assert 0<=n;
   x := x+n;
   n := 0];
}
```
(c)

Figure 3. Client of inc

tain the version in Figure 3(b). In this version, every action other than the atomic increment accesses only local variables. Moreover, atomic increments commute with each other. Therefore, every action in the second version of inc is both a right and a left mover. Consequently, the entire loop is reducible to a single atomic action. Now, using purely sequential reasoning techniques and by writing an appropriate invariant for the loop in add, we perform abstraction on this atomic action. The resulting atomic action is semantically equivalent to the body of the version of add in Figure 3(c).

2.1 A device cache

We have illustrated abstraction and reduction using a collection of small examples. We now demonstrate the symbiotic nature of these two techniques using a larger and nontrivial example. In the following, in order to make it easier to follow the proof steps, sentences that constitute a proof step are indicated by **(S1)**, **(S2)**, etc.

Device cache operation: Procedure Read in Figure 4 reads a number of bytes from a device's physical storage, modeled by the variable device. Client threads call Read to request to read (up to) size bytes from device into the output parameter buffer, starting from index start in the device. Read returns the number of bytes it was able to read (possibly less than size) from the device in the output parameter bytesread. In order to make subsequent requests to Read faster, the implementation of Read caches the bytes read into an (unbounded) memory buffer cache. In this example, the type int is the set of non-negative integers. The reader can informally assume that all integers are initialized to 0. Initial states will be treated more formally later in the paper. The variables device, cache and buffer are all integer-indexed maps starting from 0.

The variable currsize stores the number of bytes from the device that are already available in the cache. When there are enough bytes in the cache (lines 2-4), Read jumps to COPY_TO_BUFFER to copy the contents of cache to buffer (lines 18-22). In this case, the number of bytes read is the same as the number of bytes requested by the client.

If the cache does not contain all the bytes requested, there are two possibilities:

- If in line 5 a thread t observes newsize > currsize, this means another thread is in the process of copying bytes from the device to the cache as will be explained below. In this case, t can read the portion of cache between 0 and currsize (lines 18-21). If bytesread < size at return, the client thread t may retry later for the rest of the bytes.
- Otherwise, thread t reads the missing bytes from the device into the cache. The driver allows only one thread to read from the device and write to the cache. This is implemented by the following synchronization policy. The first thread that attempts to read from the device sets the variable newsize to start + size (line 9), and then performs the actual read by jumping to READ_DEVICE (line 11). While newsize is greater than currsize, no other thread attempts to access the section of cache and device between currsize and newsize. As explained above, during this period, another thread can read the portion of cache between 0 and currsize. The thread that jumped to READ_DEVICE sets currsize to newsize (lines 14-17), which makes the recently read bytes from the device available for further calls to Read by all threads.

Synchronization mechanisms: This example uses two different mechanisms to synchronize the client threads.

- A lock protects accesses to currsize and newsize, thereby allowing us to reduce the code blocks from lines 1-10 and lines 14-17 into atomic actions **(S1)**. This increased granularity of atomicity also allows us to introduce and prove the invariant currsize <= newsize.
- Only one thread at a time is allowed to update newsize, make the jump to READ_DEVICE from line 10, and update cache. We capture this synchronization by introducing an abstract lock variable al that is associated with the locking predicate currsize < newsize.
  - **Acquiring al:** The lock al is available to a thread in a state where currsize == newsize. A thread increases newsize (line 9), making the locking predicate true, to acquire al.
  - **Releasing al:** A thread sets currsize to newsize (line 15–16), making the locking predicate false to release al.

  Thus, the lock al is acquired just before the jump to READ_DEVICE and held during the execution of lines 11–17.

Specification: We would like to verify that if the input parameters of Read satisfy 0 <= start && 0 <= size at entry, then the output parameters of Read satisfy (forall x:int. start <=

```
procedure Read (start : int, size : int)
 returns (buffer : [int]int, bytesread : int)
{
    var i, j, tmp : int;

 1  acquire();
 2  i := currsize;
 3  if (start + size <= i) {
 4    release(); goto COPY_TO_BUFFER;
 5  } else if (newsize > i) {
 6    size := (start <= i) ? i - start : 0;
 7    release(); goto COPY_TO_BUFFER;
 8  } else {
 9    newsize := start + size;
10    release(); goto READ_DEVICE;
    } // end if

    READ_DEVICE:
11  while (i < start + size) {
12    cache[i] := device[i];
13    i := i + 1;
    }
14  acquire();
15  tmp  := newsize;
16  currsize := tmp;
17  release();

    COPY_TO_BUFFER:
18  j := 0;
19  while(i < size) {
20    buffer[j] := cache[start + j];
21    j := j + 1;
    }
22  bytesread := size; return;
}
```

Figure 4. Original device cache program

```
procedure Read (start : int, size : int)
 returns (buffer : [int]int, bytesread : int)
{
    var i, j, tmp : int;

 A  [if (*) {
       havoc i, size;
       assume (size == 0 || start + size <= currsize);
       goto COPY_TO_BUFFER;
     } else {
       i := currsize;
       assume newsize <= i && i < start + size;
       newsize := start + size;
       goto READ_DEVICE;
     }]


    READ_DEVICE:
 B  while (i < start + size) {
 C    [assert currsize <= i; cache[i] := device[i]];
 D     i := i + 1;
    }
 E  [tmp := newsize;
     currsize := tmp];



    COPY_TO_BUFFER:
 F  j := 0;
 G  while (j < size) {
 H    [assert start + j < currsize; buffer[j] := cache[start + j]];
 I     j := j + 1;
    }
 J  bytesread := size; return;
}
```

Figure 5. Transformed device cache program

x && x < start + bytesread ==> buffer[x] == device[x]) at exit. This task is straightforward if Read executes without any interference. Our goal in the following is to convert the entire body of Read into an abstract atomic action that is strong enough to verify the aforementioned property.

Verification: We will argue that lines and actions that access shared variables are of the desired mover types in order to allow us to convert the body of Read into an atomic action. We will argue that

- the update of cache at line 12 is a right mover **(S2)**, and
- the read of cache at line 20 is a left mover **(S3)**

Note that buffer is a local variable and device is immutable, so the only possible conflict among these lines could be due to the variable cache. The execution of line 12 is not simultaneously enabled in two different threads because of the lock al. The argument that line 12 does not conflict with line 20 requires the introduction of assertions just prior to the execution of these actions. These assertions are shown on lines C and H in Figure 5; they are checked atomically with the execution of the action (shown by surrounding square brackets). These assertions capture why the update and read of cache do not conflict: from a state in which both hold, the update and read access different indices of cache!

The transformation of line 12 and line 20 in Figure 4 into line C and line H in Figure 5 introduces a dependency on the variable currsize because of the assertion annotations. Therefore, we must also argue that lines C and H commute to the right and left respectively of the action updating currsize on line E. Due to the abstract lock al, line C is not enabled simultaneously with line E. Also, line H commutes to the left of line E when executed from a state satisfying the already proved invariant currsize <= newsize.

We now show how to transform the atomic action in lines 1-10 of Figure 4 into a right mover. In its original form, this action does not commute to the right of the atomic action in lines 14-17. To see this, consider the scenario where lines 1-10 are executed by a thread t from a state in which currsize < (start + size) < newsize; consequently size is reduced to either 0 or currsize-start. Suppose that this were followed by another thread u executing the action in lines 14-17 and increasing currsize to newsize. Clearly, this scenario is not equivalent to one in which u executes first followed by t, since in this case t would not modify size.

To transform lines 1-10 into a right mover, we perform two abstractions to get the action shown on line A in Figure 5. We first abstract the check at line 5 of Figure 4 to a non-deterministic decision and enable lines 6-7 to run even when currsize == newsize **(S4)**. This causes a thread to return fewer bytes than size even though it could jump to READ_DEVICE and read all bytes it needs from device. The second abstraction is to update the variable size nondeterministically to a value less than or equal to that assigned by the original program in the case when control jumps to COPY_TO_BUFFER **(S5)**. This allows the code to read fewer than the number of bytes available to it in the cache. This abstraction is justified since the additional behavior in the transformed Read (Figure 5) might have occurred if the call had taken place while an update of cache was being executed by another thread. This abstraction is also safe from the point of view of proving that the buffer contains a valid snapshot of device for the first bytesread bytes. The transformed Read in Figure 5 can now be reduced into an atomic action, because lines A-D are right movers and lines F-J are left movers **(S6).** Finally, we can prove the relevant condition about output parameters at exit using purely sequential reasoning techniques.

We introduce and verify the invariant (forall x:int. 0 <= x < currsize ==> cache[x] == device[x]) once the code in Figure 5 has been converted into a single atomic action **(S7)**. A proof of Read in Owicki-Gries at the line-level granularity would require all introduced invariants to be included in the annotations at several

$$
\begin{array}{rccl}
\mathit{Atomic}: & \alpha & ::= & \varphi \triangleright \tau \\
\mathit{Stmt}: & s & ::= & \alpha \mid \rho() \mid s^{\circlearrowleft} \mid s\,;\,s \mid s \,\Box\, s \mid s \parallel s
\end{array}
$$

Figure 6. The language

lines. In addition, it would require facts about global variables established at assignments and conditional checks to be propagated through the annotations in subsequent lines. Our method can handle this example much more simply by introducing invariants such as those in **(S7)** when atomic blocks are large enough.

3. Preliminaries

A program P is represented by a tuple $P = \langle \mathit{Var}, \mathit{Main}, \mathit{Body}\rangle$. We refer to elements of the tuple using dotted notation (e.g., $P.\mathit{Main}$) and omit reference to P when clear from the context. Var is the set of uniquely-named global variables. The program heap is modeled by using a map similarly to ESC/Java [11] and Boogie [2]. The statement Main is the body of the program. Body is a map from procedure names to statements to be executed when a procedure is called.

Figure 6 shows the language that we use to describe programs formally. In our language, we express each atomic statement (Atomic) with a *gated action* $\varphi \triangleright \tau$. The store predicate φ is the *gate* of the action. If executed at a state that violates φ, the action "goes wrong". From states that satisfy φ, the set of state transitions allowed by this gated action are described by transition predicate τ. Statements (Stmt) are either procedure calls ($\rho()$) or are built from atomic statements by sequential (;) or parallel ($\|$) composition, non-deterministic choice ($\Box$) or looping ($s^{\circlearrowleft}$). We model argument passing using global variables, so a call to ρ is simply denoted by $\rho()$.

We sometimes express transition predicates in the compact notation: $\langle\tau\rangle_M \equiv \tau \wedge \bigwedge_{x\in \mathit{Var}\setminus M}(x' = x)$ where M is list of variables written by α and x' is a variable that refers to the value of x after the transition described by τ is taken. For example, we can express the Boogie statements `assume e`, `assert e` and `x := e` (where `e` is an expression and `x` is a variable) with the gated actions $\mathtt{true} \triangleright \langle \mathtt{e}\rangle_\emptyset$, $\mathtt{e} \triangleright \langle \mathtt{true}\rangle_\emptyset$ and $\mathtt{true} \triangleright \langle \mathtt{x}' = \mathtt{e}\rangle_{\mathtt{x}}$, respectively.

$\mathit{Atoms}(s)$, the set of all gated atomic actions in s, is computed recursively as follows:

$$
\begin{array}{rcl}
\mathit{Atoms}(\varphi \triangleright \tau) & = & \{\varphi \triangleright \tau\} \\
\mathit{Atoms}(s^{\circlearrowleft}) & = & \mathit{Atoms}(s) \\
\mathit{Atoms}(s_1; s_2) & = & \mathit{Atoms}(s_1) \cup \mathit{Atoms}(s_2) \\
\mathit{Atoms}(s_1 \Box s_2) & = & \mathit{Atoms}(s_1) \cup \mathit{Atoms}(s_2) \\
\mathit{Atoms}(s_1 \| s_2) & = & \mathit{Atoms}(s_1) \cup \mathit{Atoms}(s_2) \\
\mathit{Atoms}(\rho()) & = & \emptyset
\end{array}
$$

We extend the definition of Atoms to programs as follows: $\mathit{Atoms}(P) = \mathit{Atoms}(\mathit{Main}) \cup \bigcup_{\rho\in \mathrm{Dom}(\mathit{Body})} \mathit{Atoms}(\mathit{Body}(\rho))$

3.1 Execution semantics

The program state evolves over time as threads execute gated actions. Each thread has a unique identifier from a set Tid. t_{main} is a distinguished thread id that runs the statement $P.\mathit{Main}$, i.e., the program body. The operational semantics of our programming language is given in Figure 7.

To represent the fact that a statement $s \in \mathit{Stmt}$ is executed by a thread $t \in \mathit{Tid}$, we use the *dynamic statement* $t : s$. The statements skip and error are also dynamic statements representing a program that has terminated normally and with an assertion violation, respectively. The semantics only allows dynamic statements ($\mathit{Dynamic}$) to be executed, so, before being executed, each static statement must be labeled by the id of the executing thread. The gate and the transition predicate of a gated action may refer to the current thread id through the special variable $\mathit{tid} \in \mathit{Var}$. When the gated action $\varphi \triangleright \tau$ is being executed by a thread $t \in \mathit{Tid}$, t is substituted for tid in both φ and τ. To represent the creation of new threads by the parallel composition statement, we make use of two functions $\mathit{left}, \mathit{right} : \mathit{Tid} \to \mathit{Tid}$. We assume that left and right together define a tree over Tid with t_{main} being the root of the tree.

A state of P is a tuple (σ, d) where σ is the current store, and d is the dynamic statement representing a partially executed program. We express a feasible state transition derivable from the operational semantics in Figure 7, as $(\sigma_1, d_1) \to (\sigma_2, d_2)$, where d_1 and d_2 are dynamic statements. We denote the transitive closure of $\to$ with $\to^*$. The program starts with an arbitrary store and the dynamic statement $t_{main} : P.\mathit{Main}$.

Let $\varphi[t/\mathit{tid}]$ and $\tau[t/\mathit{tid}]$ be formulas obtained by substituting t for tid in φ and τ, respectively. $\sigma \vDash \varphi[t/\mathit{tid}]$ states that $\varphi[t/\mathit{tid}]$ is satisfied given the valuation of the variables in σ. Similarly, $(\sigma_1, \sigma_2) \vDash \tau[t/\mathit{tid}]$ states that $\tau[t/\mathit{tid}]$ is satisfied given the valuation of unprimed variables in σ_1 and primed variables in σ_2. We will denote with $(\varphi \triangleright \tau)[t]$ the gated action $\varphi[t/\mathit{tid}] \triangleright \tau[t/\mathit{tid}]$.

The semantics of a gated action $t : \varphi \triangleright \tau$ is given in the rules ATOMIC and ERROR. We call transitions obtained from ATOMIC and ERROR *atomic transitions* of the program. ATOMIC states that if the current store satisfies $\varphi[t/\mathit{tid}]$, the store is modified atomically consistent with $\tau[t/\mathit{tid}]$ and the current dynamic statement becomes skip. Otherwise, ERROR states that the execution *goes wrong*, where the store does not change but the statement becomes error.

An *execution* of s is a sequence of feasible transition steps $(\sigma_1, t : s) \to (\sigma_2, d_2) \to \cdots \to (\sigma_n, d_n)$, where t is a thread id from Tid. An execution is *terminating* if it ends in skip or error. An execution is *blocking* if it cannot be extended. Clearly, any terminating execution is also a blocking execution. An execution *succeeds* if it ends in skip and *fails* (or *goes wrong*) if it ends in error. In the following, we define $\mathit{Good}(t, s, \varphi)$ as the set of pre- and post-store pairs associated with succeeding executions of s executed by thread t from stores satisfying φ. $\mathit{Bad}(t, s, \varphi)$ is the set of pre-stores associated with failing executions. Formally,

$$
\begin{array}{rcl}
\mathit{Good}(t, s, \varphi) & = & \{(\sigma_1, \sigma_2) \mid \sigma_1 \vDash \varphi,\ (\sigma_1, t : s) \to^* (\sigma_2, \mathsf{skip})\} \\
\mathit{Bad}(t, s, \varphi) & = & \{\sigma_1 \mid \sigma_1 \vDash \varphi,\ \exists \sigma_2.\ (\sigma_1, t : s) \to^* (\sigma_2, \mathsf{error})\}
\end{array}
$$

A statement s *may go wrong from* φ if there exists a failing run of s from φ, i.e. $\exists t \in \mathit{Tid}.\ \mathit{Bad}(t, s, \varphi) \neq \emptyset$. A program P *may go wrong from* φ if $P.\mathit{Main}$ may go wrong from φ.

$\sigma|_{\mathrm{V}}$ represents the projection of σ to the set of variables $V \subseteq \mathit{Var}$. The projections of Good and Bad to V are defined as as follows:

$$
\begin{array}{rcl}
\mathit{Good}|_{\mathrm{V}}(t, s, \varphi) & = & \{(\sigma_1|_{\mathrm{V}}, \sigma_2|_{\mathrm{V}}) \mid (\sigma_1, \sigma_2) \in \mathit{Good}(t, s, \varphi)\} \\
\mathit{Bad}|_{\mathrm{V}}(t, s, \varphi) & = & \{\sigma_1|_{\mathrm{V}} \mid \sigma_1 \in \mathit{Bad}(t, s, \varphi)\}
\end{array}
$$

The definitions of Good and Bad are also extended to programs: $\mathit{Good}(P, \varphi)$ and $\mathit{Bad}(P, \varphi)$ are shorthands for $\mathit{Good}(t_{main}, P.\mathit{Main}, \varphi)$ and $\mathit{Bad}(t_{main}, P.\mathit{Main}, \varphi)$, respectively.

4. The proof method

In our approach, a proof of a program is performed by a sequence of steps each of which modifies the current proof context. The proof context consists of the current program P and a store predicate $\mathcal{I}$. A proof context $P, \mathcal{I}$ specifies a set of executions of P that start from stores satisfying $\mathcal{I}$. $\mathcal{I}$ not only constrains initial stores from which P is run, but also is an invariant guaranteed to be preserved by every atomic transition of P. We give a formal definition of $\mathcal{I}$ in Section 4.1. A proof step is denoted $P_1, \mathcal{I}_1 \dashrightarrow P_2, \mathcal{I}_2$. The core proof rules are given in Figure 8.

$$\boxed{(\sigma_1, d_1) \rightarrow (\sigma_2, d_2)}$$

$$\begin{array}{rrcl} \mathit{Dynamic}: & d & ::= & \mathsf{skip} \mid \mathsf{error} \mid t : s \mid d \,;\, d \mid d \,\|\, d \\ & E & ::= & [\cdot] \mid E \,;\, d \mid E \,\|\, d \mid d \,\|\, E \end{array}$$

ATOMIC
$$\frac{\sigma_1 \vDash \varphi[t/tid] \qquad (\sigma_1, \sigma_2) \vDash \tau[t/tid]}{(\sigma_1, t : \varphi \rhd \tau) \rightarrow (\sigma_2, \mathsf{skip})}$$

ERROR
$$\frac{\sigma \vDash \neg\varphi[t/tid]}{(\sigma, t : \varphi \rhd \tau) \rightarrow (\sigma, \mathsf{error})}$$

FORK
$$\frac{u = \mathit{left}(t) \qquad w = \mathit{right}(t)}{(\sigma, t : (s_1 \| s_2)) \rightarrow (\sigma, u : s_1 \| w : s_2)}$$

EVALUATE
$$\frac{(\sigma_1, t : s_1) \rightarrow (\sigma_2, t : s_2) \quad s_2 \neq \mathsf{error}}{(\sigma_1, E[t : s_1]) \rightarrow (\sigma_2, E[t : s_2])}$$

EVALUATE-ERROR
$$\frac{(\sigma_1, t : s) \rightarrow (\sigma_2, \mathsf{error})}{(\sigma_1, E[t : s]) \rightarrow (\sigma_2, \mathsf{error})}$$

LABEL
$$\frac{}{(\sigma, t : (s_1 \,;\, s_2)) \rightarrow (\sigma, t : s_1 ; t : s_2)}$$

SEQUENTIAL
$$\frac{}{(\sigma, \mathsf{skip} \,;\, t : s) \rightarrow (\sigma, t : s)}$$

PROC-CALL
$$\frac{\mathit{Body}(\rho) = s}{(\sigma, t : \rho()) \rightarrow (\sigma, t : s)}$$

CHOOSE-FIRST
$$\frac{}{(\sigma, t : (s_1 \,\Box\, s_2)) \rightarrow (\sigma, t : s_1)}$$

CHOOSE-SECOND
$$\frac{}{(\sigma, t : (s_1 \,\Box\, s_2)) \rightarrow (\sigma, t : s_2)}$$

LOOP-SKIP
$$\frac{}{(\sigma, t : s^{\circlearrowleft}) \rightarrow (\sigma, \mathsf{skip})}$$

LOOP-ITER
$$\frac{}{(\sigma, t : s^{\circlearrowleft}) \rightarrow (\sigma, t : s \,;\, t : s^{\circlearrowleft})}$$

JOIN-FIRST
$$\frac{}{(\sigma, \mathsf{skip} \,\|\, t : s) \rightarrow (\sigma, t : s)}$$

JOIN-SECOND
$$\frac{}{(\sigma, t : s \,\|\, \mathsf{skip}) \rightarrow (\sigma, t : s)}$$

Figure 7. The operational semantics

$$\boxed{P_1, \mathcal{I}_1 \dashrightarrow P_2, \mathcal{I}_2}$$

INVARIANT
$$\frac{\vDash \mathcal{I}_2 \Rightarrow \mathcal{I}_1 \qquad P \vdash \mathcal{I}_2}{P, \mathcal{I}_1 \dashrightarrow P, \mathcal{I}_2}$$

AUX-ANNOTATE
$$\frac{\begin{array}{c} \mathit{Atoms}(P) = \{\varphi^1 \rhd \tau_1^1, \ldots, \varphi^n \rhd \tau_1^n\} \qquad \varphi^1 \rhd \tau_2^1 \vdash \mathcal{I} \quad \ldots \quad \varphi^n \rhd \tau_2^n \vdash \mathcal{I} \\ a \notin \mathit{Var} \qquad \vDash (\varphi^1 \wedge \tau_1^1) \Rightarrow \forall a.\exists a'.\, \tau_2^1 \quad \ldots \quad \vDash (\varphi^n \wedge \tau_1^n) \Rightarrow \forall a.\exists a'.\, \tau_2^n \end{array}}{P, \mathcal{I} \dashrightarrow P[\mathit{Var} \mapsto \mathit{Var} \cup \{a\}, \varphi^1 \rhd \tau_1^1 \mapsto \varphi^1 \rhd \tau_2^1, \ldots, \varphi^n \rhd \tau_1^n \mapsto \varphi^n \rhd \tau_2^n], \mathcal{I}}$$

SIMULATE
$$\frac{\mathcal{I} \vdash \alpha_1 \preceq \alpha_2 \qquad \alpha_2 \vdash \mathcal{I}}{P, \mathcal{I} \dashrightarrow P[\alpha_1 \mapsto \alpha_2], \mathcal{I}}$$

REDUCE-SEQUENTIAL
$$\frac{P, \mathcal{I} \vdash \alpha_1 : \mathbb{R} \quad \text{or} \quad P, \mathcal{I} \vdash \alpha_2 : \mathbb{L}}{P, \mathcal{I} \dashrightarrow P[\alpha_1 ; \alpha_2 \mapsto \alpha_1 \circ \alpha_2], \mathcal{I}}$$

REDUCE-CHOICE
$$\frac{}{P, \mathcal{I} \dashrightarrow P[\alpha_1 \Box \alpha_2 \mapsto \alpha_1 \oplus \alpha_2], \mathcal{I}}$$

REDUCE-LOOP
$$\frac{\begin{array}{c} P, \mathcal{I} \vdash \alpha : m \qquad m \in \{\mathbb{R}, \mathbb{L}\} \qquad \varphi \rhd \tau \vdash \mathcal{I} \\ \vDash \varphi \Rightarrow \tau[\mathit{Var}/\mathit{Var}'] \qquad \mathcal{I} \vdash (\varphi \rhd \tau \circ \alpha) \preceq \varphi \rhd \tau \end{array}}{P, \mathcal{I} \dashrightarrow P[\alpha^{\circlearrowleft} \mapsto \varphi \rhd \tau], \mathcal{I}}$$

EXPAND-PARALLEL
$$\frac{\alpha_3 = \alpha_1[\mathit{left}(tid)/tid] \qquad \alpha_4 = \alpha_2[\mathit{right}(tid)/tid]}{P, \mathcal{I} \dashrightarrow P[\alpha_1 \| \alpha_2 \mapsto (\alpha_3 ; \alpha_4) \Box (\alpha_4 ; \alpha_3)], \mathcal{I}}$$

Figure 8. The core proof rules of our method

Each proof step is governed by a proof rule which either rewrites the program or changes the program invariant. A rewrite of P is denoted $P[x \mapsto y]$ where an element x of the program is replaced with another element y of the same type. A rewrite may modify Var by adding new variables to it, or may replace statements in Main or Body by others.

A program is *proved correct* by a sequence of proof steps $P, \mathsf{true} \dashrightarrow P_1, \mathcal{I}_1 \dashrightarrow P_2, \mathcal{I}_2 \cdots \dashrightarrow P_n, \mathcal{I}_n$ if the final program P_n is one in which all the gated actions are *validated*. A gated action $\varphi \rhd \tau$ is *validated* by showing that $\mathcal{I}_n \Rightarrow \varphi$ is a valid formula. Since $\mathcal{I}_n$ is an invariant of P_n, no execution of P_n starting from a state in $\mathcal{I}_n$ causes an assertion violation. The soundness theorem (Theorem 1 below) asserts that in this case the initial program P cannot go wrong starting from a state in $\mathcal{I}_n$.

Soundness. The following lemma states that the application of each core proof rule given in Figure 8 preserves the soundness of assertion checking, i.e., failing runs of the original program are preserved after an application of any rule. In addition, the new program "simulates" succeeding runs of the original program from a store σ as long as the former does not go wrong from σ. We give a proof outline for Lemma 1 below. The complete proofs are given in our technical report [8].

LEMMA 1 (Preservation). *Let $P_1, \mathcal{I}_1 \dashrightarrow P_2, \mathcal{I}_2$ be a proof step. Let $\mathrm{V} = P_1.\mathit{Var}$ and $\mathrm{X} = P_2.\mathit{Var} \backslash P_1.\mathit{Var}$. Then the following hold:*

1. *$\mathit{Bad}(P_1, \exists X.\, \mathcal{I}_2) \subseteq \mathit{Bad}|_{\mathrm{V}}(P_2, \mathcal{I}_2)$*
2. *For each $(\sigma_1, \sigma_2) \in \mathit{Good}(P_1, \exists X.\, \mathcal{I}_2)$:*
   (a) $\sigma_1 \in \mathit{Bad}|_{\mathrm{V}}(P_2, \mathcal{I}_2)$, or
   (b) $(\sigma_1, \sigma_2) \in \mathit{Good}|_{\mathrm{V}}(P_2, \mathcal{I}_2)$)

PROOF SKETCH: We prove that for each execution E_1 of P_1 starting from a store σ_1 in $\mathcal{I}_2$ and ending in state σ_2, there exists a "witness" execution E_2 of P_2 from σ_1 that leads to σ_2 or goes wrong. To obtain E_2, we provide in each case a sequence of local transformations to E_1. Each transformation is either the swap of two adjacent actions justified by their mover types, or the replacement of an action by a more abstract one. The correctness of the execution transformations are justified by the antecedents of the proof rules. □

The following theorem generalizes the lemma to an arbitrary number of proof steps and states the soundness of our proof method.

THEOREM 1 (Soundness). *Let $P_1, \mathsf{true} \dashrightarrow \cdots \dashrightarrow P_n, \mathcal{I}_n$ be a sequence of proof steps. Let $\mathrm{V} = P_1.\mathit{Var}$ and $\mathrm{X} = P_n.\mathit{Var} \backslash P_1.\mathit{Var}$. If P_n cannot go wrong from $\mathcal{I}_n$, then P_1 cannot go wrong from $\mathcal{I}_n$ and $\mathit{Good}(P_1, \exists X.\, \mathcal{I}_n) \subseteq \mathit{Good}|_{\mathrm{V}}(P_n, \mathcal{I}_n)$.*

The $P_1, \mathcal{I}_1$ notation implicitly requires that $\mathcal{I}_1$ be an invariant of program P_1. Furthermore, any transformation $P_1, \mathcal{I}_1 \dashrightarrow P_2, \mathcal{I}_2$ has the property that $\mathcal{I}_2 \Rightarrow \mathcal{I}_1$. This connection between $\mathcal{I}_1$ and $\mathcal{I}_2$ becomes significant while proving Theorem 1.

In the rest of this section, we present the proof rules of our system, which are the building blocks of the higher-level proof tactics presented in Section 5. We anticipate that most users of QED will reason at the level of these higher-level, combined steps. In this section, we have made an attempt at motivating each proof rule by providing forward pointers to Section 5.

4.1 Invariants

The second component of the proof context, $\mathcal{I}$, is a store predicate that specifies the stores from which each execution of P preserves $\mathcal{I}$ for each atomic transition of P. A gated action $\varphi \triangleright \tau$ *preserves* $\mathcal{I}$, denoted $\varphi \triangleright \tau \vdash \mathcal{I}$, if from stores satisfying $\mathcal{I}$, $\varphi \triangleright \tau$ either goes wrong or preserves $\mathcal{I}$ at the post-store, i.e., $(\varphi \wedge \tau) \Rightarrow (\mathcal{I} \Rightarrow \mathcal{I}')$. Here, for a store predicate ψ, the store predicate ψ' is obtained by replacing each free occurrence of each $v \in \mathit{Var}$ with v'. Then $\mathcal{I}$ is an *invariant* of a program P, denoted $P \vdash \mathcal{I}$, if all gated actions in $\mathit{Atoms}(P)$ preserve $\mathcal{I}$. Each proof starts from the proof context P, true. New program invariants are introduced by strengthening the current invariant by using INVARIANT.

4.2 Auxiliary variables

The rule AUX-ANNOTATE adds a fresh auxiliary variable a to the program and replaces each gated action $\varphi^i \triangleright \tau_1^i$ in P with a new gated action $\varphi^i \triangleright \tau_2^i$. We require that $\varphi^i \triangleright \tau_2^i$ preserves the current invariant. The new action τ_2^i specifies how a is updated by the gated action (or left unmodified) for each value of a in the current state, but preserves the effect of τ_1^i on other variables. In particular, the introduction of auxiliary variables cannot cause an action to block at a state s if the original action did not. This proof rule is typically used to annotate gated actions with synchronization information; see Section 5.2 for details.

4.3 Simulation

The simulation relation, defined in this section, enables us to decide when SIMULATE can replace a gated action with another. Let $\mathcal{I}$ be a store predicate. $\varphi_2 \triangleright \tau_2$ *simulates* $\varphi_1 \triangleright \tau_1$ from $\mathcal{I}$, denoted $\mathcal{I} \vdash \varphi_1 \triangleright \tau_1 \preceq \varphi_2 \triangleright \tau_2$, if the following two conditions hold:

1. $\models \mathcal{I} \wedge \varphi_2 \Rightarrow \varphi_1$
2. $\models \mathcal{I} \wedge \varphi_2 \wedge \tau_1 \Rightarrow \tau_2$

The first condition above states that whenever $\varphi_1 \triangleright \tau_1$ goes wrong from $\mathcal{I}$, so does $\varphi_2 \triangleright \tau_2$. The second condition states that whenever $\varphi_2 \triangleright \tau_2$ does not go wrong from $\mathcal{I}$, it simulates succeeding runs of $\varphi_1 \triangleright \tau_1$ from $\mathcal{I}$.

Common ways of rewriting a gated action $\varphi_1 \triangleright \tau_1$ to $\varphi_2 \triangleright \tau_2$ are strengthening the gate, i.e., $\varphi_2 \Rightarrow \varphi_1$ and $\tau_1 = \tau_2$, or weakening the transition predicate, i.e., $\tau_1 \Rightarrow \tau_2$ and $\varphi_1 = \varphi_2$. In both cases, the new action is an abstraction of the old one. While the former adds extra failing behaviors to the gated action by adding new assertions for the pre-store, the latter adds extra succeeding behaviors.

Borrowing assertions. A special case of the simulation proof rule is the annotation of gated actions with assertions. When a gated action $\varphi_1 \triangleright \tau_1$ is annotated with an assertion ψ, it is transformed to the gated action $(\varphi_1 \wedge \psi) \triangleright \tau_1$. We use assertions to enable later applications of the simulation rule. Consider the case where we would like to replace $\varphi_1 \triangleright \tau_1$ with $\varphi_2 \triangleright \tau_2$ but it is not the case that $\mathcal{I} \vdash \varphi_1 \triangleright \tau_1 \preceq \varphi_2 \triangleright \tau_2$ because the invariant $\mathcal{I}$ is not strong enough. Instead of having to strengthen $\mathcal{I}$ by reasoning about the entire program, we simply express our belief about the program state when $\varphi_1 \triangleright \tau_1$ is reached using an assertion ψ. The new simulation check $\mathcal{I} \vdash (\varphi_1 \wedge \psi) \triangleright \tau_1 \preceq (\varphi_2 \wedge \psi) \triangleright \tau_2$ is more likely to pass. We often insert assertions to express beliefs about synchronization mechanisms; see Section 5.2 for details.

Our method allows program transformations to proceed while the verification of the introduced assertion is deferred to later proof stages. Borrowing assertions in this manner (instead of proving them right away) is a powerful tool, since in the transformed program with larger atomic actions, validation of assertions can often be done automatically.

Validating gated actions. Another special case of SIMULATE is using the invariant to validate gated actions, i.e. proving their their gates (assertions) are valid. We use the following derived rule RELAX for the purpose of proving and eliminating the gates.

$$\textsc{relax}\quad \frac{\models \mathcal{I} \Rightarrow \varphi}{P, \mathcal{I} \dashrightarrow P[\varphi \triangleright \tau \mapsto \mathsf{true} \triangleright \tau], \mathcal{I}}$$

A program is proved correct when all the gates of $\mathit{Atoms}(P)$ are replaced with true. The rule emphasizes the fact that validating assertions in $\varphi \triangleright \tau$, in essence, is nothing more than sequential reasoning where the invariant is used as the pre-condition to discharge the assertions in sequential code represented by τ. Here, φ is obtained by propagating the assertions in the sequential code to the beginning of the code block using weakest preconditions [7].

4.4 Reduction

$$\boxed{P, \mathcal{I} \vdash s : m}$$

$$\textsc{left-mover}\quad \frac{\begin{array}{c} \alpha = \varphi \triangleright \tau \\ \forall t, u \in \mathit{Tid}.\, \forall \beta \in \mathit{Atoms}(P) : \\ (t \neq u) \Rightarrow (\mathcal{I} \vdash (\beta[u] \circ \alpha[t]) \preceq (\alpha[t] \circ \beta[u])) \end{array}}{P, \mathcal{I} \vdash \varphi \triangleright \tau : \mathbb{L}}$$

$$\textsc{right-mover}\quad \frac{\begin{array}{c} \alpha = \mathsf{true} \triangleright (\varphi \wedge \tau) \\ \forall t, u \in \mathit{Tid}.\, \forall \beta \in \mathit{Atoms}(P) : \\ (t \neq u) \Rightarrow (\mathcal{I} \vdash (\alpha[t] \circ \beta[u]) \preceq (\beta[u] \circ \alpha[t])) \end{array}}{P, \mathcal{I} \vdash \varphi \triangleright \tau : \mathbb{R}}$$

Figure 9. Right and left movers

Our reduction rules are based on Lipton's theory [18]. Figure 9 shows how we determine whether an action is a right or a left mover using the judgment $P, \mathcal{I} \vdash s : m$. These judgments are used for reducing sequential composition, nondeterministic choice, loops, and parallel composition.

The rules LEFT-MOVER and RIGHT-MOVER define the mechanism by which we label atomic actions in the program as a left or right mover. These rules correspond to performing a pairwise commutativity check with all other atomic actions and are performed automatically by QED. The rule LEFT-MOVER is straightforward; an action $\varphi \triangleright \tau$ is a left mover if it commutes to the left of every action in the program. The rule RIGHT-MOVER is similar; however, to establish whether $\varphi \triangleright \tau$ is a right mover, we check the commutativity of $\mathsf{true} \triangleright (\varphi \wedge \tau)$ to the right of actions in the program. We have introduced this asymmetry deliberately to increase the applicability of the RIGHT-MOVER rule in the case when the program contains blocking actions. To see why this was needed, observe that an action $\varphi \triangleright \tau$ that may go wrong cannot commute to the right of an action β that blocks.

Sequential composition. The REDUCE-SEQUENTIAL makes use of the $\circ$ operator defined in terms of $\mathsf{wp}(\tau, \varphi)$, the (sequential) weakest-precondition of φ with respect to the transition τ. It is applicable to $\alpha_1; \alpha_2$ if either α_1 is a right mover or α_2 is a left mover.

$$\boxed{P_1, \mathcal{I}_1 \dashrightarrow P_2, \mathcal{I}_2}$$

INLINE-CALL

$$\frac{\mathit{Body}(\rho) = s}{P, \mathcal{I} \dashrightarrow P[\rho() \mapsto s], \mathcal{I}}$$

REDUCE-RECURSIVE

$$\frac{\begin{array}{c}\mathcal{M} = \{\rho_1, ..., \rho_n\} \text{ is closed under call} \\ \text{There exists an annotation function } \mathit{Annot} \text{ where for each } \rho_i \in \mathcal{M}: \quad \mathit{Annot}(\rho_i) = (m_i, \varphi_i, \tau_i) \\ \forall \rho_i \in \mathcal{M}: \qquad \varphi_i \triangleright \tau_i \vdash \mathcal{I} \qquad P, \mathcal{I}, \mathit{Annot} \vdash \mathit{Body}(\rho_i) : m_i \qquad P, \mathcal{I}, \mathit{Annot} \vdash \{\varphi_i, \tau_i\}\rho_i \qquad \mathit{Finalizable}(\rho_i, \varphi_i)\end{array}}{P, \mathcal{I} \dashrightarrow P[\mathit{Body}(\rho_1) \mapsto \varphi_1 \triangleright \tau_1, \ldots, \mathit{Body}(\rho_n) \mapsto \varphi_n \triangleright \tau_n], \mathcal{I}}$$

Figure 10. Rules for procedures

$$\begin{array}{rcl}\varphi_1 \triangleright \tau_1 \circ \varphi_2 \triangleright \tau_2 & = & (\varphi_1 \wedge \mathsf{wp}(\tau_1, \varphi_2)) \triangleright (\tau_1 \circ \tau_2) \\ \mathsf{wp}(\tau, \varphi) & = & \forall \mathit{Var}'.\, \tau \Rightarrow \varphi' \\ \tau_1 \circ \tau_2 & = & \exists \mathit{Var}''.\, \tau_1[\mathit{Var}''/\mathit{Var}'] \wedge \tau_2[\mathit{Var}''/\mathit{Var}]\end{array}$$

Nondeterministic choice. The REDUCE-CHOICE makes use of the $\oplus$ operator defined as follows.

$$\varphi_1 \triangleright \tau_1 \oplus \varphi_2 \triangleright \tau_2 \quad = \quad (\varphi_1 \wedge \varphi_2) \triangleright (\tau_1 \vee \tau_2)$$

Loops. REDUCE-LOOP reduces a loop $\alpha^{\circlearrowleft}$ to a single gated action $\varphi \triangleright \tau$. Intuitively, φ is a predicate that is true at the beginning of the loop and τ specifies a relation between the beginning of the loop and the end of any iteration. The conditions needed to apply this rule are the following: 1) $\varphi \triangleright \tau$ preserves the current invariant, 2) the body of the loop is a right or left mover, and 3) $\varphi \triangleright \tau$ simulates by zero or more iterations of the loop body α.

Parallel composition. We reduce parallel statements composed of gated actions to sequential compositions using three rules. The first rule EXPAND-PARALLEL eliminates the parallel composition by explicitly enumerating the two possible interleavings of the gated actions. The following derived rules REDUCE-PARALLEL-I/II exploit the gated actions being right/left movers to directly eliminate the parallel composition.

REDUCE-PARALLEL-I

$$\frac{\begin{array}{c}P, \mathcal{I} \vdash \alpha_1 : \mathbb{L} \quad \text{or} \quad P, \mathcal{I} \vdash \alpha_2 : \mathbb{R} \\ \alpha_3 = \alpha_1[\mathit{left}(\mathit{tid})/\mathit{tid}] \qquad \alpha_4 = \alpha_2[\mathit{right}(\mathit{tid})/\mathit{tid}]\end{array}}{P, \mathcal{I} \dashrightarrow P[\alpha_1 \| \alpha_2 \mapsto \alpha_3; \alpha_4], \mathcal{I}}$$

REDUCE-PARALLEL-II

$$\frac{\begin{array}{c}P, \mathcal{I} \vdash \alpha_1 : \mathbb{R} \quad \text{or} \quad P, \mathcal{I} \vdash \alpha_2 : \mathbb{L} \\ \alpha_3 = \alpha_1[\mathit{left}(\mathit{tid})/\mathit{tid}] \qquad \alpha_4 = \alpha_2[\mathit{right}(\mathit{tid})/\mathit{tid}]\end{array}}{P, \mathcal{I} \dashrightarrow P[\alpha_1 \| \alpha_2 \mapsto \alpha_4; \alpha_3], \mathcal{I}}$$

4.5 Procedures

In this section, we show how our proof method deals with procedures. The rules are given in Figure 10. The rule INLINE-CALL is particularly simple; it simply inlines $\mathit{Body}(\rho)$ at the call site. All the other rules discussed previously can be used to transform the body of a procedure iteratively making it smaller and simpler. In the limit, the procedure body could be transformed to a single atomic action. Once the procedure body has been simplified enough, it can be inlined at call sites without any significant increase in the program size.

The rule INLINE-CALL is inadequate if there is recursion in the program. In this case, we use the proof rule REDUCE-RECURSIVE, which takes a set $\mathcal{M}$ of procedures that are closed under the call relation and gives a mechanism whereby the body of each method in $\mathcal{M}$ is replaced by a gated action. To break circular dependencies, REDUCE-RECURSIVE requires an annotation function Annot that provides a specification for each procedure ρ_i in $\mathcal{M}$ and for each loop occurring in a ρ_i as explained later in this section.

$$\boxed{P, \mathcal{I} \vdash s : m}$$

BOTH-MOVER

$$\frac{P, \mathcal{I} \vdash s : \mathbb{L} \qquad P, \mathcal{I} \vdash s : \mathbb{R}}{P, \mathcal{I} \vdash s : \mathbb{B}}$$

SUP-MOVER

$$\frac{P, \mathcal{I} \vdash s : n \qquad n \sqsubseteq m}{P, \mathcal{I} \vdash s : m}$$

ATOMIC-NON-MOVER

$$\frac{}{P, \mathcal{I} \vdash \varphi \triangleright \tau : \mathbb{A}}$$

CALL-MOVER

$$\frac{\mathit{Annot}(\rho) = (m, \varphi, \tau)}{P, \mathcal{I} \vdash \rho() : m}$$

COMBINE-MOVERS

$$\frac{P, \mathcal{I} \vdash s_1 : m \qquad P, \mathcal{I} \vdash s_2 : n \qquad \bullet \in \{;, \square, \|\}}{P, \mathcal{I} \vdash s_1 \bullet s_2 : m \bullet n}$$

LOOP-MOVER

$$\frac{P, \mathcal{I} \vdash s : m}{P, \mathcal{I} \vdash s^{\circlearrowleft} : m^{\circlearrowleft}}$$

Definitions of $m; n$, $m \square n$, $m \| n$, and $m^{\circlearrowleft}$

$m; n$	$\mathbb{B}$	$\mathbb{R}$	$\mathbb{L}$	$\mathbb{A}$		$m \square n$	$\mathbb{B}$	$\mathbb{R}$	$\mathbb{L}$	$\mathbb{A}$		$m \| n$	$\mathbb{B}$	$\mathbb{R}$	$\mathbb{L}$	$\mathbb{A}$		$m^{\circlearrowleft}$	
$\mathbb{B}$	$\mathbb{B}$	$\mathbb{R}$	$\mathbb{L}$	$\mathbb{A}$		$\mathbb{B}$	$\mathbb{B}$	$\mathbb{R}$	$\mathbb{L}$	$\mathbb{A}$		$\mathbb{B}$	$\mathbb{B}$	$\mathbb{R}$	$\mathbb{L}$	$\mathbb{A}$		$\mathbb{B}$	$\mathbb{B}$
$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{A}$	$\mathbb{A}$		$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{A}$	$\mathbb{A}$		$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	-	-		$\mathbb{R}$	$\mathbb{R}$
$\mathbb{L}$	$\mathbb{L}$	-	$\mathbb{L}$	-		$\mathbb{L}$	$\mathbb{L}$	$\mathbb{A}$	$\mathbb{L}$	$\mathbb{A}$		$\mathbb{L}$	$\mathbb{L}$	-	$\mathbb{L}$	-		$\mathbb{L}$	$\mathbb{L}$
$\mathbb{A}$	$\mathbb{A}$	-	$\mathbb{A}$	-		$\mathbb{A}$	$\mathbb{A}$	$\mathbb{A}$	$\mathbb{A}$	$\mathbb{A}$		$\mathbb{A}$	$\mathbb{A}$	-	-	-		$\mathbb{A}$	-

Figure 11. Rules for deciding movers

To define the procedure specifications provided by Annot, we must first introduce two new mover types—atomic non-mover ($\mathbb{A}$) and both-mover ($\mathbb{B}$). The partial order $\sqsubseteq$ defines a lattice over movers as specified as follows: $\mathbb{B} \sqsubseteq \mathbb{L} \sqsubseteq \mathbb{A}$ and $\mathbb{B} \sqsubseteq \mathbb{R} \sqsubseteq \mathbb{A}$ Figure 11 extends the judgment $P, \mathcal{I} \vdash s : m$ to all statements and all mover types [12]. The symbol $-$ indicates that no conclusion about the mover type can be reached and rules in Figure 11 cannot be applied. The rule CALL-MOVER allows us to use the mover type annotation for the body of a procedure at the call site. We call the statement s *atomic* if $P, \mathcal{I} \vdash s : m$ is proved for any mover $m \sqsubseteq \mathbb{A}$. The function Annot is defined as follows:

- For each $\rho_i \in \mathcal{M}$, $\mathit{Annot}(\rho_i)$ returns a tuple (m_i, φ_i, τ_i), where $m_i \in \{\mathbb{A}, \mathbb{L}, \mathbb{R}, \mathbb{B}\}$ specifies the mover type of the body of ρ_i, and φ_i and τ_i are the pre-condition and the post-condition of ρ_i, respectively. If the rule REDUCE-RECURSIVE succeeds, then it is sound to replace the body of each procedure $\rho_i \in \mathcal{M}$ with $\varphi_i \triangleright \tau_i$. While φ_i is a store predicate for the call point of the procedure, τ_i is a transition predicate and specifies a relationship between the call and return points of the procedure. For soundness of the REDUCE-RECURSIVE rule, we require that every execution of ρ_i from φ_i can be extended to a blocking execution. We express this requirement using the predicate $\mathit{Finalizable}(\rho_i, \varphi_i)$. We currently assume that the programmer has ensured that this requirement is met and do not provide a mechanical check for it in our tool. This condition is similar to a termination requirement, in fact, termination is a sufficient condition for being finalizable. We leave the mechanization of this check to future work.

- For each loop $s^{\circlearrowleft}$ in the body of a procedure $\rho_i \in \mathcal{M}$, $Annot(s^{\circlearrowleft})$ returns a store predicate φ. φ is a loop invariant for $s^{\circlearrowleft}$.

The above specifications are written only for loops and procedures that are proved to be atomic as required by REDUCE-RECURSIVE. Therefore, the problem of deriving these specifications is the same as providing loop invariants and procedure specifications for sequential code.

Given an annotation function $Annot$, REDUCE-PROCEDURE checks each procedure ρ_i in three phases. First, it checks that the specification of ρ_i indicates an action that preserves the invariant. In the rule, this is described by the condition $\varphi_i \triangleright \tau_i \vdash \mathcal{I}$. Second, it checks that given the mover types for procedures in $Annot$, the body of ρ_i in P conforms to its mover type provided by $Annot$. In the statement of the rule, this is expressed with the condition $P, \mathcal{I}, Annot \vdash Body(\rho_i) : m_i$. The rules in Figure 11 are used in this phase; for brevity, we have elided the implicit argument $Annot$ to all the judgments. Third, it verifies that the body of ρ_i implements the specification described by the pre-condition φ_i and the post-condition τ_i. This is described by the condition $P, \mathcal{I}, Annot \vdash \{\varphi_i, \tau_i\}\rho_i$. Formally, $P, \mathcal{I}, Annot \vdash \{\varphi_i, \tau_i\}\rho_i$ if for all $t \in Tid$, the following two conditions hold: (1) $(\sigma_1, \sigma_2) \models \tau_i$ for all $(\sigma_1, \sigma_2) \in Good(t, Body(\rho_i), \varphi_i)$, and (2) $Bad(t, Body(\rho_i), \varphi_i) = \emptyset$.

The third requirement described above is verified by generating a verification condition (VC) and checking its validity using an automatic theorem prover. Since the body of ρ_i is atomic, checking whether ρ_i implements its specification requires sequential reasoning only. We use a VC generation technique based on weakest preconditions, similar to that implemented in the Boogie verifier [2]. To handle the use of the parallel composition operator within procedure bodies, we define the weakest precondition of $s_1 \| s_2$ with respect to a post-store φ as follows:

$$\mathsf{wp}(s_1 \| s_2,\ \varphi) \ = \ \mathsf{wp}(s_1[\mathit{left}(\mathit{tid})/\mathit{tid}],\ \mathsf{wp}(s_2[\mathit{right}(\mathit{tid})/\mathit{tid}],\ \varphi))$$

Notice that the weakest precondition for the parallel composition is similar to the one for the sequential composition [2]. Since $s_1 \| s_2$ is part of a procedure body that has been proved to be atomic, $s_1 \| s_2$ is atomic as well. The correctness of the weakest precondition of parallel composition crucially depends on this observation.

5. Implementation with high-level tactics

The proof rules introduced in Section 4 are low-level rules that are the building blocks of proofs. In this section, we will present higher-level, more intuitive-to-use proof tactics: coarser-grained proof rules built out of lower-level ones. Figure 12 summarizes the tactics, their usage and the low-level rules justifying the application of the tactic. By construction, each tactic preserves the soundness of assertion checking.

We implemented our proof method in an interactive tool called QED. QED uses the Boogie framework [2] as its front end and forwards the validity checks to the Z3 SMT solver [6]. Our tool accepts as input a multithreaded program written in an extension of the Boogie programming language [2] and a proof script. The transformed program after the application of each proof rule or tactic is available for the user to examine. If required, the tool automatically generates the verification conditions necessary to prove the antecedents of proof rules and checks them using Z3. The commands are rejected if this check fails.

Proof strategy. We view a proof of a concurrent program as a sequence of steps, each applying a tactic presented in this section. We have found the proof strategy sketched below to be a good start:

```
while exist unverified assertions :
    while not done { do reduction with reduce stmt/loop/proc }
    eliminate assertions with check
    if exist unverified assertions :
        repeat { do abstraction with abstract or mutex }
        repeat { introduce a specification with invariant, annot }
```

In each iteration we first apply reduction and then verify assertions. When there are still unverified assertions, we do abstraction and introduce specifications, which allows reduction to obtain coarser atomic actions at the next iteration. In the following sections we elaborate on the operations of the tactics referred to above and give examples of how these tactics are used during the proof in Section 2.1.

5.1 Introducing invariants and specifications

The tactic invariant ψ introduces a new invariant into the proof context. The current invariant $\mathcal{I}$ is replaced with the conjunction $\mathcal{I} \wedge \psi$. The tactic fails to change the invariant if any gated action in $Atoms(P)$ does not preserve $\mathcal{I} \wedge \psi$.

The tactics annot pre, annot post and annot inv allow us to introduce specifications for loops and procedures. The specifications can be introduced partially, where newly introduced specifications are conjoined with the existing specifications. This allows the annotation function referred to in Section 4.5 to be defined partially while the proof progresses. While annot pre ρ, φ adds a new pre-condition, an assertion, φ, annot post ρ, τ adds a new post-condition τ to the specification of procedure ρ. The tactic annot mover ρ, m (needed for REDUCE-PROCEDURE) specifies that the body of ρ is of mover type m. The tactic[2] annot inv $s^{\circlearrowleft}, \varphi$ adds a loop invariant φ to the specification of the loop $s^{\circlearrowleft}$.

Example: The specification for `Read` in Figure 4 is introduced in step **S7** by the tactics annot pre `Read,0 <= start && 0 <= size` and annot post `Read,(forall x:int. start <= x && x < start + bytesread ==> buffer[x] == device[x])`.

5.2 Abstraction

Abstraction is applied by adding either extra transitions or extra assertions to a gated action. We use the abstract tactic for the former by having a gated action read or write a nondeterministic value from/to a variable. The tactic mutex is used for the latter to infer extra assertions to the gated actions using synchronization information.

Read and write abstractions. A read abstraction is performed by the tactic abstract read $x, \varphi \triangleright \tau$. It makes the gated action $\varphi \triangleright \tau$ read a nondeterministic value from x at the beginning of the action thus making the operation of the action independent of the initial value of x when its execution starts. For this purpose, abstract read $x, \varphi \triangleright \tau$ replaces the given action $\varphi \triangleright \tau$ with the following:

$$\begin{array}{rcl} P, \mathcal{I} & \dashrightarrow & P[\varphi \triangleright \langle\tau\rangle_M \mapsto \varphi \triangleright \overline{\tau}], \mathcal{I} \\ \overline{\tau} & = & \begin{cases} \varphi \triangleright \langle \exists x. \tau \rangle_M & \text{if} \quad x \in M \\ \varphi \triangleright \langle x = x' \wedge \exists x. \tau \rangle_M & \text{if} \quad x \notin M \end{cases} \end{array}$$

A write abstraction is performed through the tactic abstract write x,$\varphi \triangleright \tau$. It makes the gated action $\varphi \triangleright \tau$ write a nondeterministic value to x at the end of the action. For this purpose, abstract write $x, \varphi \triangleright \tau$ replaces the given action $\varphi \triangleright \tau$ with the following:

[2] In our implementation, we assign each statement a unique "label". The tactics that require a statement as a parameter are given the label of the statement.

Tactic	Usage	Low-level rules
invariant ψ	Add a new invariant to the proof context.	INVARIANT
annot pre ρ, φ	Introduce a pre-condition specification to a procedure.	–
annot post ρ, τ	Introduce a post-condition specification to a procedure.	–
annot mover ρ, m	Specify that the body of procedure ρ is of mover type m.	–
annot inv $s^{\circlearrowleft}, \varphi$	Introduce a loop invariant specification to a loop.	–
reduce stmt	Apply reduction iteratively on the program body and procedure bodies.	REDUCE-SEQUENTIAL/CHOICE/PARALLEL
reduce loop $\alpha^{\circlearrowleft}, \varphi \rhd \tau$	Reduce the loop to its previously given specification.	REDUCE-LOOP
reduce proc $\rho_1, \ldots, \rho_n$	Reduce the procedures to their previously given specifications and do inlining.	REDUCE-PROCEDURE, INLINE-CALL
inline ρ	Inline the body of the procedure ρ at all call sites.	INLINE-CALL
abstract read $x, \varphi \rhd \tau$	Abstract the value of x at the entry of the action $\varphi \rhd \tau$.	SIMULATE
abstract write $x, \varphi \rhd \tau$	Abstract the value of x at the exit of the action $\varphi \rhd \tau$.	SIMULATE
assert $\psi, \varphi \rhd \tau$	Add new assertion by strengthening the gate of the action $\varphi \rhd \tau$	SIMULATE
mutex $\phi, x_1, \ldots, x_n$	Add assertions for a mutual exclusion access policy for variables $x_1, \ldots, x_n$	AUX-ANNOTATE, INVARIANT, SIMULATE
check ρ	Validate assertions in the procedure body using sequential analysis and $\mathcal{I}$.	RELAX

Figure 12. The high-level proof tactics.

$$\begin{array}{rcl} P, \mathcal{I} & \dashrightarrow & P[\varphi \rhd \langle \tau \rangle_M \mapsto \varphi \rhd \overline{\tau}], \mathcal{I} \\ \overline{\tau} & = & \langle \exists x'.\tau \rangle_{M \cup \{x\}} \end{array}$$

By the definition of simulation, both read and write abstractions are sound by construction. We do not allow abstractions that violate the program invariant $\mathcal{I}$.

Example: We do two read abstractions in our running example.

First, in step **S4**, we abstract the read of `newsize` for the action corresponding to the branch of the `if` statement at lines 5-7 of Figure 4. This allows `Read` to take this branch even though `newsize ==currsize` holds, as a result, reading from the cache the available bytes (fewer than the initially requested `size`) and returns them, although it could have fetched all the requested bytes from `device`.

Second, in step **S5**, we abstract `size` at the beginning of the action that spans lines 3-7 of Figure 4 (after reducing the first two branches of `if` to a single atomic action by REDUCE-CHOICE). This allows `Read` to return fewer bytes than the original `size`. The assumption `(size == 0 || start + size <= i)` obtained from the condition of `if` guarantees that the abstraction leaves `size` still in the safe bounds.

These two abstractions allow us to prove that the branches of `if` are all right-movers and reduce the entire statement to the one given at line 1 of Figure 5. Notice that, neither abstraction breaks the specification of `Read` and each corresponds to a behavior that could have occurred in a different interleaving. □

Adding assertions. We use the tactic assert $\psi, \varphi \rhd \tau$ to add the assertion p to the gated action to yield $\psi \wedge \varphi \rhd \tau$. In the following, we describe a tactic that, using hints about mutual exclusion synchronization in the program, infers appropriate assertions for gated actions.

The tactic mutex $\phi, x_1, \ldots, x_n$ takes a store predicate ϕ and a set of program variables $x_1, \ldots, x_n \in \mathit{Var}$. The hint communicated through this tactic is that ϕ specifies a mutual exclusion condition that holds at all the program states from which a gated action reads from or writes to a variable x_i. In this regard, ϕ can be thought of as a high-level description of a locking discipline that can be implemented by any program variables. The tactic automatically adds to the program a fresh auxiliary variable a with domain $\mathit{Tid} \cup \{0\}$[3], and generates the following invariant over $\mathit{Var} \cup \{a\}$:

$$\mathcal{I} \quad = \quad (a \neq 0) \Leftrightarrow \phi$$

$\mathcal{I}$ associates the auxiliary variable with ϕ. Intuitively, ϕ is true whenever it is acquired and a stores the id of the thread that acquired ϕ, and ϕ is false whenever ϕ is not acquired by any thread and a stores 0. The operation of the tactic includes 1) by AUX-ANNOTATE, adding a to the set of program variables, 2) by INVARIANT, adding the invariant $\mathcal{I}$, and then 3) by annotating gated actions as follows:

1. Replace every gated action $\varphi \rhd \tau$ such that $\models (\varphi \wedge \tau) \Rightarrow (\neg\phi \wedge \phi')$ with $\varphi \rhd (\tau \wedge (a' = \mathit{tid}))$.
2. Replace every gated action $\varphi \rhd \tau$ such that $\models (\varphi \wedge \tau) \Rightarrow \neg\phi'$ with $(\varphi \wedge (a = \mathit{tid})) \rhd (\tau \wedge (a' = 0))$.
3. Replace every gated action $\varphi \rhd \tau$ such that $\models (\varphi \wedge \tau) \Rightarrow (\phi \Leftrightarrow \phi')$ with $\varphi \rhd (\tau \wedge (a' = a))$.
4. Replace every gated action $\varphi \rhd \tau$ that read from or write to the variable x_i with $(\varphi \wedge (a = \mathit{tid})) \rhd \tau$.

The above operations are justified by the rules AUX-ANNOTATE and SIMULATE. The assertion $a = \mathit{tid}$ is the key to showing that actions annotated with $a = \mathit{tid}$ are movers in later reduction steps because they are non-conflicting.

Example:
1. The application of mutex in our running example of Section 2.1 (in step **S1**) expresses the fact that `newsize` and `currsize` are protected by a variable `lock`, which is modified by `acquire` and `release` primitives. We used the tactic mutex `lock == true, currsize,newsize`. The assertion $a = \mathit{tid}$ were added to the lines accessing `currsize` and `newsize` between `acquire` and `release`.
2. In order to reduce the code in Figure 5 into one atomic action, it is crucial to prove that the lines between `READ_DEVICE` and `COPY_TO_BUFFER` commute over each other. This enabled the reasoning in **S2** and **S3**. In fact, only one thread can execute these lines. Recall that the synchronization mechanism in `Read` allows only the thread that establishes `currsize < newsize` to access the device. In order to prove that this is the case and to use this fact in later reductions, we use the tactic mutex `currsize < newsize, device`. As a result, the assertion $a = \mathit{tid}$ is added to the actions spanning the block of code between the labels `READ_DEVICE` and `COPY_TO_BUFFER`. □

5.3 Reduction

Reducing statements. The tactic reduce stmt is used to compute coarser atomicities in the program by iteratively applying reduction rules in Figure 8.

Example: In our running example, we apply reduce stmt twice. The first reduction (in step **S1**), after using the mutex tactic as described in Section 5.2, combines branches of the `if` statement between lines 3-4, 5-7 and 9-10 of Figure 4 to separate atomic blocks, each having the code at line 1-2 at the beginning. Then it merges the branches of `if` to a single atomic action. In addition,

[3] We assume that 0 is not an element of Tid, and represents “no thread”.

the block at lines 14-17 is also reduced to a single action. Figure 5 shows the state of the program at this point. The second reduction (in step **S6**), using the assertions at lines 3 and 8 reduces the loops as described below and combines the entire body into a single action. □

Reducing loops. The tactic reduce loop is used to reduce an entire loop to a gated action given with the same tactic. reduce loop $s^{\circlearrowleft}$ $\varphi \rhd \tau$ uses REDUCE-LOOP to reason about possibility of reducing $s^{\circlearrowleft}$ to $\varphi \rhd \tau$.

Example: The loops in our running example at lines B-D and G-I of Figure 5 are reduced to single actions using the tactic reduce loop. After steps **S2** and **S3**, the body of the first loop is a right-mover, and the body of the latter is a left-mover. We use the specification `(currsize <= i)` $\rhd\langle$ `((i <= i') && forall x:int. ((i'-1) <= x <= start + size)==>(cache'[x] == device[x]))` $\rangle_{(\mathtt{i,cache})}$ for the first loop and `(start + i < currsize)` $\rhd\langle$ `(i <= i') && forall x:int. ((i'-1) <= x <= size)==>(buffer'[x] == cache[start + x])` $\rangle_{(\mathtt{i,buffer})}$ for the latter. Notice that, these gated actions specify the copying from `device` to `cache` and from `cache` to `buffer` properly. In addition, these actions are right- and left- movers themselves, and are used later in the reduce stmt tactic. □

Reducing procedures. For the cases where the body of a procedure is small, we provide the tactic inline ρ, which replaces all the calls to ρ with its body after doing proper substitutions for formals. In other cases, the tactic reduce proc $\rho_1, ..., \rho_n$ is used to eliminate calls to the procedures $\rho_1, ..., \rho_n$, which are closed under call. reduce proc does the checks specified in the rule REDUCE-PROCEDURE where $\mathcal{M} = \{\rho_1, ...\rho_n\}$, and the existing specifications for loops and procedures given as described in Section 5.1 define the annotation function $Annot$. The tactic fails if a procedure or loop does not implement its given specification, or the body of a procedure is not of the given mover type. If all the checks pass, it replaces all the calls to $\rho_1, ..., \rho_n$ with their corresponding specifications.

Example: Suppose that the loops at lines B-D and F-I of Figure 5 are implemented as separate procedures, and the loops are replaced with the calls `copy_from_device(start,size)` and `buffer := copy_from_cache(start,size)`, respectively. In this case, we could still reduce the bodies of the procedures `copy_from_device` and `copy_from_cache`, the loops, to single actions by reduction. In this case, the specifications to be inlined at the call sites in `Read` will be similar to the loop specifications given above while describing reduce loop on `Read`. □

6. Experience

In this section we present our experience with several benchmark algorithms[4]. We were able to prove the benchmarks, except for the non-blocking stack, using our tool QED. All the proof steps driven by high-level tactics were fully mechanized; QED required only a few seconds to finish each proof. We have generated and verified the verification conditions for the non-blocking stack manually; currently, we are trying to mechanize this proof as well. Our experience with the benchmarks is summarized in the following conclusions:

- Our proofs did not require complicated global invariants that capture possible interference at each interleaving point. Instead, we attempted to achieve the correct level of atomicity by reasoning locally about the effect of synchronization on individual actions. The iterative use of reduction and abstraction was crucial in this endeavor.
- The benchmarks, while operating with fine-grained concurrency, ensure a coarse level of atomicity through a variety of sophisticated synchronization protocols. We were able capture these protocols with few uses of abstraction through the assert, mutex and rwlock tactics.
- In many benchmarks, after aggressive use of reduction and abstraction, the atomic blocks obtained were large enough so that global program invariants stated for the transformed program were almost as simple as invariants for sequential programs.

In the remainder of this section, we discuss the proof of each benchmark at a high level.

6.1 Purity benchmarks

```
apply_f()
 1 var x, fx : int;
 2 acquire(m);
 3 x := z;
 4 release(m);
 5 while(true) {
 6   fx := f(x);
 7   acquire(m);
 8   if(x == z) {
 9     z := fx; release(m);
10     break;
11   } else {
12     x := z; release(m);
13   }
14 }
```

Figure 13. Optimistic concurrency control using transaction retry

We verified the examples in [10], thereby demonstrating that our approach generalizes existing work on enforcing atomicity through abstractions. We were able to handle all the examples by simple applications of abstract read and abstract write on variables that are accessed but left unmodified in the pure blocks. Consider an example (Figure 13) from Section 5 of [10]. This program reads a shared variable `z` in one critical section, performs a long computation `f` on its value, and then attempts to writes back the result into `z` in another critical section. The tactic "mutex `m==true, z` " allows the code block 2-4 and both branches of the `if` statement to be proved atomic. The reads of `z` at line 3 and 12 are unnecessary for correctness and could therefore be abstracted.

Notice that the loop in this example and other examples in this section contain the `break` statement. Such loops cannot be translated directly to the loop statement in Figure 6. Instead of rewriting the programs to eliminate `break`, we provide the user tactics to hoist the final, successful iteration out of the loop. In this particular example, we hoist the successful `if` branch at lines 8-10 out of the loop. This is a sound operation since every terminating execution of the loop contains these lines once at the end. We apply similar transformations to the loops in other examples given below.

The rest of the loop after the above transformation does not touch global variables, and also implements a simple "skip" operation. Thus this portion of the program is a both-mover. Finally we apply reduction and convert the body of `apply_f` into a single atomic action that assigns `f(z)` to `z`.

6.2 Multiset

Figure 14 shows a concurrent multiset of integers with `InsertPair` and `Delete` operations. The implementation contains an array `M` of cells for storing the multiset elements; the `elt` field of the cell stores the element and the `vld` field indicates whether the value stored in `elt` is valid. Procedures `acq` and `rel` acquire and release

[4] We will use pseudocode for brevity; the verified versions of the benchmarks in the Boogie programming language can be found at: *http//home.ku.edu.tr/∼telmas/popl09bench.tar.gz*

```
FindSlot(x:int)
  returns r:int
for (i=0; i<N; i++) {
 acq(M[i]);
 if (M[i].elt==nil && !(M[i].vld)){
  M[i].elt := x; rel(M[i]);
  r := i; return;
 } else { rel(M[i]); }
} r := -1; return;
```

```
Delete(x:int)
  returns r:bool
for (i=0; i<N; i++) {
 acq(M[i]);
 if (M[i].elt==x && M[i].vld){
  M[i].elt:=nil; M[i].vld:=false;
  rel(M[i]; r := true; return;
 } else { rel(M[i]); }
} r := false; return;
```

```
InsertPair(x:int, y:int)
  returns r:bool
 1 i := FindSlot(x);
 2 if (i == -1) {
 3  r := false; return;
 4 }
 5 j := FindSlot(y);
 6 if (j == -1) {
 7  M[i].elt = nil;
 8  r := false; return;
 9 }
10 acq(M[i]);
11 acq(M[j]);
12 M[i].vld = true;
13 M[j].vld = true;
14 rel(A[i]);
15 rel(A[j]);
16 r := true; return;
```

Figure 14. The multiset data structure

(M[i].lck), the lock of cell i. In [9], we proved that InsertPair and Delete are atomic using an abstraction map from M to an abstract specification variable S representing the multiset contents. The variable S was required to abstract away from the concrete values of the indices of M into which multiset elements are stored. Using our method, we transformed the bodies of InsertPair and Delete implementations to atomic actions, indicated by [...], given below. Thus, we replaced a proof based on abstraction mappings with a simpler, layered correctness proof that ends with sequential reasoning.

```
Delete(x:int)
  returns r:bool {
var i : int;
[havoc i;
 if(*) {
   r := false; return;
 } else {
   assume 0 <= i < N;
   assume M[i].elt == x;
   assume M[i].vld;
   M[i].elt:=nil; M[j].vld:=false;
   r := true; return;
 }]
}
```

```
InsertPair(x:int, y:int)
  returns r:bool {
var i,j : int;
[havoc i,j;
 assume 0 <= i,j < N;
 if(*) {
  r := false; return;
 } else {
  assume M[i].elt == nil;
  assume M[j].elt == nil;
  assume !(M[i].vld);
  assume !(M[i].vld);
  M[i].elt:=x; M[i].vld:=true;
  M[j].elt:=y; M[j].vld:=true;
  r := true; return;
 }]
}
```

We first proved the following specification for FindSlot and inlined the specification at call points in InsertPair. This proof required reasoning similar to the example in Figure 13: we hoisted the last iteration of the loop in FindSlot, which allocates an empty slot, outside and performed abstractions on the other iterations. The abstractions were done on the elt and vld fields of M[i]. Then we used the tactic "mutex (M[x].lck==true), M[x].elt, M[x].vld" to reduce the code blocks between calls to acq and rel to atomic actions, including the succeeding iteration in FindSlot.

```
FindSlot(x:int)
  returns r:int {
[havoc r;
 if(*) {
   r := -1; return;
 } else {
   assume (0 <= r < N && M[r].elt == nil && !(M[i].vld));
   M[i].elt := x;   return;
 }]
}
```

The tactic "mutex (M[x].elt != nil && !M[x].vld), M[x].elt, M[x].vld" allowed us to capture the property that, once FindSlot returns an allocated slot, its elt and vld fields are not modified by other threads. This fact is essential for proving that the atomic actions spanning the blocks 1-3 and 5-9 of InsertPair are right-movers. Then we were able to merge the three atomic blocks (lines 1-3, 5-9 and 10-16) in InsertPair into a single atomic action. We also introduced the invariant (forall x:int. 0 <= x && x < N && M[x].vld ==> M[x].elt != nil) at the very end of the proof. It is crucial to introduce this invariant only after the body of InsertPair has been transformed into a single action because the individual actions in InsertPair do not preserve this invariant. The correctness proof for a sequential multiset implementation would have required the same simple invariant.

6.3 Non-blocking algorithms

Our experience with the following collection of non-blocking algorithms demonstrates that our method can also be used to verify highly-concurrent algorithms from the literature.

```
rightpush(v) returns r:bool
 1 while(true) {
 2  k := oracle(right);
 3  prev := A[k-1];
 4  cur := A[k];
 5  if(prev.val != RN && cur.val = RN) {
 6   if (k = MAX + 1) { r := false; return; }
 7   if (CAS(&A[k-1], prev, <prev.val,prev.ctr+1>))
 8    if (CAS(&A[k], cur, <v,cur.ctr+1>))
 9     { r := true; return; }
10  }
11 }
```

Figure 15. Example operation of obstruction-free deque

Obstruction-free deque. The double-ended queue (deque) from [14] provides four operations –rightpop, rightpush, leftpop, and leftpush– operation with similar designs, to insert/remove elements to/from both ends of the queue. Figure 15 shows the rightpush operation, which inserts an element into the right end of the deque. After reading two consecutive array elements in lines 3-4 to local variables prev and cur, rightpush inserts the given value v if CAS operations at lines 7-8 both succeed. Because of the reads at lines 3-4, the CAS operations cannot be proved to be movers in the original program. We abstracted these reads using the "abstract read" tactic, which made prev and cur point to arbitrary elements. In addition, we abstracted the write to k at line 2, using the "abstract write tactic. This abstraction does not affect the correctness of the operation, since the function oracle can return any index from the deque; its purpose is to return the optimum index for having fewer failing attempts.

The above proof steps allowed us to reason about the operation of two consecutive CAS operations. We then hoisted the part of the loop body with the successful CAS operations out of the loop. Then it was easy, by using the specification of CAS, to prove that the remaining part of the loop was a skip operation. By adding the deque invariants indicated in the algorithm description in [14], we were able to prove that the CAS operations, which were hoisted out of the loop, are left-movers in the abstracted program. This gave us an atomic block that contains two CAS operations with the desired behavior. We proved the other operations using a similar approach.

Non-blocking stack. Figure 16 shows the pop and push operations from Michael's non-blocking stack algorithm [19]. The operations make use of a hazard pointer per thread in order to solve the well-known ABA problem. [22] gave a proof of this algorithm using concurrent separation logic. Since the algorithm uses fine-grained concurrency, the proof in [22] required reasoning about invariants at each interleaving point throughout the code. Using our method, we performed a simpler proof that transforms the bodies of pop and push to the following atomic actions:

```
pop()                           push(b:ref)
  returns r:ref                   returns r:bool
 1 var t,n : ref;                1 var t : ref, n : int = 1;
 2 while (true) {                2 while(n <= THREADS) {
 3   t := TOP;                   3   if (H[n] == b) {
 4   if(t == null) break;        4     r := false; return;
 5   H[tid] := t;                5   } n := n + 1;
 6   if(t != TOP) continue;      6 }
 7   n := TL[t];                 7 while (true) {
 8   if(CAS(&TOP, t, n)) break;  8   t := TOP;
 9 }                             9   TL[b] := t;
10 H[tid] := null;              10   if(CAS(&TOP, t, b)) break;
11 r := t; return;              11 } r := true; return;
```

Figure 16. Michael's algorithm with hazard pointers

```
pop()                              push(b:ref) {
  returns r:ref {                  var m : ref,  i : int;
var m, n : ref;                    [assert b != nil;
[havoc m,n;                         if(*) {
 if(*) {                              r := false;
   assume (TOP == nil);             } else {
   r := nil;                          tl[b] = TOP;
 } else {                             TOP := b; r := true;
   assume (m != nil && m == TOP);   }]
   n := tl[m]; TOP := n; r := m;   }
 }]
}
```

The proof of the non-blocking stack required the application of 11 tactics, in three of which we introduced invariants. Our proof strategy was aimed at making the successful executions of the code at lines 7-8 of `pop` and 9-10 of `push` atomic. We performed read abstractions on `TOP` at lines 4 and 6 of `pop` and 8 of `push` since these reads do not affect correctness. For each loop, we hoisted the loop iteration containing a successful `CAS` operation outside the loop. There is an ownership protocol implicit in the use of the stack. A thread should only push into the stack an element that it owns. The push operation transfers the ownership of the element to the stack. Ownership is transferred back to the thread that manages to pop that element from the stack. We used an auxiliary variable `owner` , a map from stack elements to thread ids, to capture this ownership transfer protocol. We also introduced assertions into the code to check that the ownership protocol is not being violated. As with other examples, we delayed the introduction of invariants until the atomic actions in the program had become coarse enough; all invariants were consequently fairly simple.

```
bakery(i:int)
 1 choosing[i] := 1;
 2 number[i] := 1 + max(number[1],...,number[N]);
 3 choosing[i] := 0;
 4 for(i:=1; i<=N ; ++i) {
 5  while(choosing[j] != 0) /*wait*/;
 6  while(number[j] != 0 && (number[j],j)<(number[i], i))/*wait*/;
 7 }
 8 c := c + 1;
 9 assert c == 1; // critical section
10 c := c - 1
11 number[i] := 0; return;
```

Figure 17. The bakery algorithm

The bakery algorithm. The bakery algorithm (Figure 17) provides non-blocking critical sections [17]. We encoded the mutual exclusion property by adding a global variable `c` that counts the number of threads in the critical section and adding the assertion `c == 1` in the critical section. We proved the property by obtaining a single action that spans the code between lines 1-7 that is enabled only when `(number[i],i)` is greater that `(number[k],k)` for all `k` different from `i`. We abstracted the waiting iterations of the `for` loop, which allowed us to eliminate the loop from the code. We applied the mutex tactic twice. The first application encoded the fact that that `choosing[i]==0` prohibits the values of `number[i]` that are smaller than `number[k]` to be read by a different thread `k` at line 6. The second application of the mutex tactic modeled lines 1-7 as an acquire for a conceptual lock, and line 11 as the release of this lock. This conceptual lock protects `c` so that the code between lines 8-10 is atomic and the assertion never fails.

7. Related work

In this section, we compare our work with other approaches for compositional verification of multithreaded programs. In a nutshell, our method is orthogonal and complementary to existing methods that do not make direct use of reduction and abstraction, and subsumes others that do.

In the Owicki-Gries approach [21], each potential interleaving point in a program must be annotated with an invariant that is valid under interference from other concurrently-executing actions. Rely-guarantee methods [25, 5, 4] make this approach more modular by obviating the need to consider each pair of concurrent statements separately. Instead, the guarantee and rely conditions of a thread provide a summary for transitions taken by this thread and the transitions taken by environment threads, respectively. Both these methods require the programmer to reason about interleavings of fine-grained actions; consequently, the required annotations are complex. Concurrent separation logic [20, 3] has the ability to maintain separation between shared and local memory dynamically. Similarly to our method, it enables sequential reasoning for multithreaded programs. By converting the original program into a simpler program that uses coarse-grained atomic actions, our method enhances the applicability of all of these approaches. At the same time, our method can benefit from these approaches as well. For example, the ability of concurrent separation logic to reason about dynamic ownership transfer of heap objects could be useful for establishing that heap accesses are non-conflicting, thereby enabling the key step of reduction in our method. The same symbiotic relationship applies to the approach in [15] where method contracts and object invariants are used to specify sharing and ownership constraints on Spec# objects.

Several verification approaches in the literature use reduction as a key ingredient [12, 24, 10, 23]. These approaches are different from ours in that (i) they are limited to simple synchronization disciplines and (ii) can only reason about commutativity of accesses that are not simultaneously enabled. [13] addresses the first issue by using auxiliary variables and access predicates to enable wider application of reduction. In addition to regular locking primitives, [23] applies mover-analysis to non-blocking synchronization primitives `LL`, `SC` and `CAS` under certain execution patterns of these primitives. As demonstrated in this paper, our method supports intricate synchronization mechanisms naturally, with moderate annotation burden. Further, our check for mover types is general and able to deduce that certain simultaneously enabled accesses commute. Abstractions have been used as a mechanism to prove atomicity in the work on purity [10, 23]. The abstraction step in our method subsumes purity and allows us to deduce pure code blocks through simple variable abstractions.

8. Conclusion

We introduced a proof method that iteratively simplifies a program by rewriting it in terms of coarser-grained atomic actions. When applied iteratively, reduction and abstraction enable further use of each other and significantly simplify programs. Our tool QED automates the proof steps in our approach, and our experience suggests that our approach provides a useful strategy to simplify the verification of assertions in concurrent programs. Future work includes extending our framework to verify programs written in

C and Spec#, developing tactics to support more synchronization idioms such as barriers and events, and applying our method to larger verification problems.

Acknowledgments

This research was supported by a career grant (104E058) from the Scientific and Technical Research Council of Turkey, the Turkish Academy of Sciences Distinguished Young Scientist Award (TUBA-GEBIP), and a research gift from the Software Reliability Research group at Microsoft Research, Redmond, WA. We would like to thank the Spec# team, particularly Rustan Leino and Mike Barnett, for their support with using the Boogie/Spec# framework.

References

[1] E. A. Ashcroft. Proving assertions about parallel programs. *J. Comput. Syst. Sci.*, 10(1):110–135, 1975.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. *FMCO '05: 4th International Symposium on Formal Methods for Components and Objects*, pages 364–387, 2005.

[3] S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.

[4] J. W. Coleman and C. B. Jones. Guaranteeing the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841, 2007.

[5] F. S. de Boer, U. Hannemann, and W.-P. de Roever. A compositional proof system for shared variable concurrency. In *FME'97: 4th International Symposium of Formal Methods Europe*, volume 1313, pages 515–532. Springer-Verlag, 1997.

[6] L. M. de Moura and N. Björner. Z3: An efficient SMT solver. In *TACAS '08: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[7] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[8] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. Technical Report MSR-TR-2008-99, Microsoft Research, 2008.

[9] T. Elmas, S. Tasiran, and S. Qadeer. VYRD: Verifying concurrent programs by runtime refinement-violation detection. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 27–37, New York, NY, USA, 2005. ACM Press.

[10] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. *IEEE Trans. Softw. Eng.*, 31(4):275–291, 2005.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[12] C. Flanagan and S. Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 1–12, New York, NY, USA, 2003. ACM.

[13] S. Freund and S. Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, 2004.

[14] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529, Washington, DC, USA, 2003. IEEE Computer Society.

[15] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. *Electron. Notes Theor. Comput. Sci.*, 174(9):23–47, 2007.

[16] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981.

[17] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

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

[19] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[20] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 268–280, New York, NY, USA, 2004. ACM.

[21] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[22] M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. *SIGPLAN Not.*, 42(1):297–302, 2007.

[23] L. Wang and S. D. Stoller. Static analysis for programs with non-blocking synchronization. In *PPoPP '05: Proceedings of the ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming*, pages 61–71. ACM Press, June 2005.

[24] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, Feb. 2006.

[25] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.