

# An Annotation Assistant for Interactive Debugging of Programs with Common Synchronization Idioms

Tayfun Elmas   Ali Sezgin   Serdar Tasiran

Koç University, İstanbul, Turkey  
{telmas, asezgin, stasiran}@ku.edu.tr

Shaz Qadeer

Microsoft Research, Redmond, WA  
qadeer@microsoft.com

## Abstract

This paper explores an approach to improving the practical usability of static verification tools for debugging synchronization idioms. Synchronization idioms such as mutual exclusion and readers/writer locks are widely-used to ensure atomicity of critical regions. We present an annotation assistant that automatically generates program annotations. These annotations express non-interference between program statements, ensured by the synchronization idioms, and are used to identify atomic code regions. This allows the programmer to debug the use of the idioms in the program. We start by formalizing several well-known idioms by providing an abstract semantics for each idiom. For programs that use these idioms, we require the programmer to provide a few predicates linking the idiom with its realization in terms of program variables. From these, we automatically generate a proof script that is mechanically checked. These scripts include steps such as automatically generating assertions and annotating program actions with them, introducing auxiliary variables and invariants. We have successfully shown the applicability of this approach to several concurrent programs from the literature.

**Categories and Subject Descriptors** D.2.5 [Software Engineering]: Testing and Debugging — debugging aids, diagnostics; D.2.4 [Software Engineering]: Software/Program Verification — assertion checkers, formal methods; D.1.3 [Programming Techniques]: Concurrent Programming — parallel programming

**General Terms** Languages, Theory, Verification

**Keywords** Concurrent Programs, Atomicity, Synchronization Idioms

## 1. Introduction

The fundamental difficulty in reasoning about multithreaded programs is the need to reason about concurrent execution of fine-grained atomic actions. Atomicity has been used as a key tool to circumvent this difficulty in many contexts, ranging from model checking [8] to static verification [4]. Atomicity is also a useful concept in programming; programmers tend to think of program executions as interleavings of atomic blocks, and do sequential reasoning on atomic blocks.

This paper presents an annotation assistant to debug synchronization idioms used in the program. In this study, we build on the interactive verification tool QED [4]. QED is a verification method for checking assertions in concurrent programs. This proof method is supported by a tool also called QED. In the QED setting, annotations and non-interference analysis are used to prove that code blocks are atomic and the verification task is progressively reduced to assertion checking in sequential code blocks. In this paper, we introduce a technique used in QED that, given simple hints about the use of a synchronization idiom, automatically generates annotations for identifying atomic blocks ensured by the idiom. The generated annotations confirm that a synchronization idiom performs the expected functionality, thus help the programmer to diagnose incorrect or insufficient uses of the idiom.

Our notion of atomicity is based on Lipton's theory of reduction [11]. Reduction 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. This concept of atomicity (e.g., [4, 9]) has been used successfully on many concurrent programs, even very intricate ones. However, the use of these tools has been limited to verification experts. In this paper, we attempt to make interactive static analysis tools for concurrent programs more easily usable by programmers.

For sequential programs, several key ideas have been instrumental in making automated reasoning about programs practical [7, 1]. Property checking on industrial-scale designs, for example, is viable if users provide annotations which allow tools to decompose the verification task. In practice, coming up with program annotations for this purpose is error-prone and difficult. This has led to research on automated approaches to generating annotations [6]. For concurrent programs, there is little work along these lines. Roughly stated, one key goal of the work presented here is to provide a similar automated annotation capability for concurrent programs.

The issue of proving non-interference between certain program actions is central to methods that check atomicity of code blocks for concurrent programs. A programmer who wants to show that a certain code block is atomic must (i) make use of synchronization mechanisms that ensure the desired non-interference in his program, and (ii) provide the program annotations needed to convince the atomicity checker that the desired non-interference has indeed been accomplished. The first task is aided by the use of well-known synchronization idioms. The second one is often more difficult and error prone. One typically needs to introduce auxiliary variables, program invariants, and assertions in order to reason about programs using these idioms. It is easily possible to write annotations that are too strong or too weak, and, when the check fails, it is difficult to diagnose whether the program contains synchronization bugs or whether the annotations are not satisfactory. We address

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PADTAD '09, July 19-20, 2009, Chicago, Illinois, USA.  
Copyright © 2009 ACM 978-1-60558-655-7/09/07...\$5.00

this issue by automating the generation of program annotations required for these scripts to succeed.

In our approach, synchronization idioms serve as contracts and proof hints to a static verification tool. For each synchronization idiom we explored, we provide an abstract semantics expressed in terms of predicates, which serve as the connection between the idiom and its realization within the program. The programmer indicates which synchronization idioms he makes use of and supplies the required predicates in terms of program variables. For example, for the readers/writer lock idiom, we require two predicates in terms of program variables indicating when the readers (alternatively, writers) lock is held. Based on the idiom and the predicates, we produce a proof script, and an iterative, automated method for generating the program annotations required to make the proof template succeed. If the proof script succeeds, the assertions in the original program have been validated. Otherwise, the way in which the script fails allows us to distinguish between i) incorrect implementation of an idiom, ii) incorrect use of the idiom that violates semantics of the idiom, or iii) the idiom not being the appropriate one to guarantee the assertions in the program. From this point of view, the annotation generation serves as a debugging tool for the concurrency protocol of the program.

This paper makes the following contributions:

- For several common synchronization idioms, we provide an abstract semantics and, using programmer-provided predicates, a way to link this semantics to its realization in a program.
- For each synchronization idiom we generate a proof script, driven by programmer-provided predicates. The script automatically generates the program annotations required to make the script referred to above succeed. The algorithm takes over the derivation of the annotations, and thus reduces the user’s annotation burden to simply providing the synchronization idiom and a few predicates.
- We provide an interactive tool using which the programmer can identify and correct synchronization errors. We have successfully used our tool on a number of examples from the literature.

## 2. Related work

There is an analogy between the use of QED for generating annotations and the Houdini tool [6]. Procedure pre- and post-conditions are formulas that make the sequential verification modular, by specifying the effect of the procedure at call points without needing its body. The validity of the specification is separately checked using the body of the procedure. Houdini infers, from a set of candidate conditions, pre- and post-conditions of procedures. The inference algorithm ensures that it outputs the strongest valid formulas as post-conditions for each candidate pre-condition. The inferred pre- and post-conditions confirm the programmer’s thinking about the procedure’s operation. Similarly, the techniques presented in the paper infer strong and valid annotations for proving atomicity of code blocks. The annotations confirm the programmer’s expectations about the atomic computations in the program.

Our work is built on many of the same ideas reported in [9], [5] and [8]. These studies also automatically apply reduction and take as input hints or specifications of how access to program variables is regulated, e.g., by specifying the locks protecting accesses to particular data variables. Our techniques are more widely applicable, since our modeling of synchronization idioms allows us to uniformly handle a wider variety of idiom implementations. More importantly, our methods are more easily usable because we do not require the user to list the program data variables protected by a particular idiom. Given a particular idiom, the user simply provides a few predicates in terms of program synchronization variables, and

```

increment():           increment():
1: acquire();          1: <acquire(); m := tid;>
2: lx := x;           2: <assert m==tid; lx := x;>
3: lx := lx + 1;      2: <assert m==tid; lx := lx + 1;>
4: x := lx;           2: <assert m==tid; x := lx;>
5: release();         2: <assert m==tid; release(); m := 0;>

```

Figure 1. The increment example

we automatically infer which program variables and statements are protected by the synchronization mechanism.

## 3. Motivation and overview

This section gives motivation for the techniques presented in the following sections, through an overview of how annotating programs helps in proving atomicity, and how this procedure can be utilized to debug the synchronization in the program.

### 3.1 Background

When talking about program executions, we will let *action* denote the program statements that are executed atomically, in one step. By an execution of the program we refer to a sequence of actions by different threads.

In order to illustrate our technique consider the code on the left in Fig. 1. Variable *x* is global and *lx* is local. Each line in the procedure denotes a separate action. We assume that only the actions in `increment` access *x* within the entire program. We would like to show that the procedure `increment` atomically increments *x*. Notice that this actually is the case, since the lock prevents threads running `increment` from being interleaved. Thus, every execution of the program is equivalent to an execution in which `increment` appears to run sequentially, and `increment` is said to be atomic.

Our core reasoning method for showing the atomicity of `increment` is based on the popular reduction theory due to Lipton [11]. In this approach, actions in `increment` are classified according to their mover types. An action  $\alpha$  is a right mover if it commutes to the right of all actions  $\beta$ , by other threads, in the program that can potentially follow  $\alpha$ . Left movers are defined similarly. Then, two actions sequentially composed can be combined into a single action if they have compatible mover types. Compound actions, by virtue of being an action themselves, will also have a mover type and the reduction will iteratively try to combine more and more actions with the ultimate goal of representing the body of `increment` as a single atomic action.

By the above definition, an action  $\alpha$  having a mover type indicates that for any action  $\beta$  in the program (including  $\alpha$ ), either  $\beta$  does not follow  $\alpha$ , or  $\alpha$  does not interfere with  $\beta$ . The lock ensures the former condition when  $\alpha$  and  $\beta$  are both actions of `increment`. Expressing this feature of the lock in the program is essential in showing that the actions guarded by the lock are movers. We accomplish this by annotating the program with assertions. The annotated version of `increment` is seen in Fig. 1 on the right.

The annotations are inserted to atomic actions as assertions and assignments, which refer to an auxiliary variable, *m*. The domain of *m* is  $Tid \cup \{0\}$  where *Tid* is the set of all thread ids ( $0 \notin Tid$ ). The annotations add auxiliary information to the state of the mutex lock: If *m*=0 the lock is free, and otherwise *m* holds the id of the thread that has acquired the lock. Variable *tid* is a special global variable: It stores the id of the currently executing thread. The assertion *m*==*tid* is used to explicitly mark the lines that a thread executes while holding the lock. Therefore, it is made explicit that two actions with these assertions may not follow each other. Then, it is proved that actions of `increment` have proper mover types to combine all to a single action.

Notice that the assertions are added to the program as extra proof obligations. If the synchronization idiom is properly used

```

-----
T1: <x := 0; v := 0;>
-----
T2:      T3:      T4:      T5:
Update(1); || Update(2); || (x1,v1):= Read(); || (x2,v2):= Read();
-----
T1: assert (v1 == v2) ==> (x1 == x2);

Update(k : int):      Read():
1: x := k;            1: lx := x;
2: lv := v;           2: lv := v;
3: v := lv + 1;       3: return (lx,lv);

```

**Figure 2.** Threads accessing the  $(x, v)$  pair

in the program, these assertions are eventually proved as valid sequentially after all the actions are merged into one action. One of the following two outcomes indicates a possible bug in the program: 1) Actions that are expected to be protected by a synchronization mechanism are not annotated with any assertion that expresses this fact. 2) Assertions that are added to actions in a code block by the annotation generation do not help in making that code block atomic by a subsequent reduction. As an example to the latter, assume that there exists a `decrement` procedure that accesses `x` but does not use the lock used by `increment`. In this case, our automatic annotation method provides no annotation for the actions of `decrement` because of the missing synchronization. Failing to annotate `decrement` even though `increment` is annotated makes actions of `increment` non-movers, which prevents `increment` from being reduced to a single action. As a result, the assertions in `increment` will not be validated since this requires having `increment` as a single action. The failing mover analysis between actions of `increment` and `decrement` can easily diagnose the missing synchronization in `decrement`.

Our annotation technique improves the approach demonstrated above in two directions. First, we can handle variety of idioms including mutual-exclusion, readers/writer lock and events using similar kind of annotations. Second, the annotation generation scheme for an idiom, such as mutual-exclusion, can be applied to different implementations of that idiom in a common way automatically. In the following we will give more information about our method on a larger example, elaborating on these directions.

### 3.2 The example

Throughout this paper, we will use several versions of the program given in Fig. 2 to demonstrate our techniques. We denote parallel composition with “||”. Dashed lines are used to visually separate sequential statements. Sequences of actions surrounded by “<...>” represent an atomic action whose effect is accomplished by executing the sequence of actions without interruption.

In the example, the global variables `x` and `v` stand for a shared data variable and its version number, respectively. Thread T1 initializes `x` and `v` and forks T2-T5. When T2-T5 all terminate, T1 checks whether the snapshots of the  $(x, v)$  pair read by T4 and T5 are consistent. The `assert` statement makes the execution go wrong if the given condition evaluates to `false`.

Procedure `Update` writes to `x` and increases `v`, and procedure `Read` takes a snapshot of the  $(x, v)$  pair. The variables `lx` and `lv` are procedure-local. Each line in the procedures denotes a separate atomic action. The procedures are written so that each action does at most one global variable read or write.

If the bodies of `Update` and `Reads` in Fig. 2 were executed sequentially, it would be trivial to prove that the assertion at the end of the program is not violated in any execution. However, in a concurrent execution, the lines of these procedures may get interleaved. In order to ensure that every call to `Read` returns the correct version number for `x`, the programmer must guarantee that

```

Update(k : int):      Read():
1: AcqWrite();        1: AcqRead();
2: x := k;            2: lx := x;
3: lv := v;          3: lv := v;
4: v := lv + 1;      4: RelRead();
5: RelWrite();        5: return (lx,lv);

AcqWrite():          AcqRead():
1: AcqMutex()        1: AcqMutex();
2: assume reads==0 && write==false; 2: assume write==false;
3: write := true;    3: reads := reads + 1;
4: RelMutex();       4: RelMutex();

RelWrite():          RelRead():
1: AcqMutex();       1: AcqMutex();
2: write := false;   2: reads := reads - 1;
3: RelMutex();       3: RelMutex();

AcqMutex():          RelMutex():
1: <assume held == false; 1: <held := false;>
2: held == true;>

```

**Figure 3.** The procedures used in Fig. 2

`Update` and `Read` by different threads do not interfere with each other, i.e. `Update` and `Read` are atomic. In other words, no `Read` is allowed to execute between the writes to `x` and `v` by `Update`, and no `Update` is allowed to execute between reads of `x` and `v` by `Read`. If this is the case, then `Update` and `Read` can be analyzed as if each runs sequentially and the assertion can be verified with little effort.

In order to ensure the aforementioned non-interference requirement, the programmer employs the readers/writer (r/w) lock idiom. Fig. 3 shows the new versions of `Update` and `Read` using the r/w lock idiom. The figure also shows the implementation of the r/w lock, which uses the mutual-exclusion (mutex) idiom to obtain the correct functionality of a r/w lock.

While proving that `Update` and `Read` are atomic, we show that the bodies of `Update` and `Read` are composed of actions of certain mover types. When the code in Fig. 3 is considered as is, the conflicting actions accessing `x` and `v` cannot be shown to be movers. However, the r/w lock ensures that these lines cannot run consecutively in any execution, i.e. they never interfere with each other. Since the mover analysis involves local commutativity checks on individual actions, it misses this global information. By annotating code with assertions, we encode that these conflicting actions run at different synchronization states and thus cannot follow each other in any execution.

We prove that `Update` and `Read` are atomic in two steps. First, by generating annotations for the mutual-exclusion idiom, we prove that procedures `AcqRead/RelRead` and `AcqWrite/RelWrite`, which implement the r/w lock operations, are composed of certain movers, thus are atomic. Second, we reason about the r/w lock to show that `Update` and `Read` are atomic. At each step, our method generates the assertions for the regarding idiom automatically, given a simple hint about how the idiom is implemented in the program. In the following, we will first introduce the implementations of these idioms, and then explain how the annotation engine is informed of these idioms through simple hints.

### 3.3 Synchronization idioms

Fig. 3 shows the procedures that implement the readers/writer and mutex lock idioms. While the former idiom is used to access the  $(x, v)$  pair atomically in `Update` and `Read`, the latter is used to implement the readers/writer lock correctly.

**Readers/writer lock.** Procedures `AcqRead` and `RelRead` (alternatively, `AcqWrite` and `RelWrite`) are used to acquire and release the read (write) lock, respectively. The implementation ensures that no code between `AcqRead` and `RelRead` is executed concurrently with code between `AcqWrite` and `RelWrite`. However, code be-

```

Update(k : int):
1: <AcqWrite(); w := tid;>
2: <assert w==tid; x := k;>
3: <assert w==tid; lv := v;>
4: <assert w==tid; v := lv + 1;>
5: <assert w==tid; RelWrite();
   w := 0;>

AcqWrite():
1: <AcqMutex(); m := tid;>
2: <assert m==tid;
   assume reads==0 && write==false;>
3: <assert m==tid;
   write := true;>
4: <assert m==tid; RelMutex();
   m := 0;>

RelWrite():
1: <AcqMutex(); m := tid;>
2: <assert m==tid;
   write := false;>
3: <assert m==tid; RelMutex();
   m := 0;>

Read():
1: <AcqRead(); w := 0;
   r[tid] := true;>
2: <assert w==0; lx := x;>
3: <assert w==0; lv := v;>
4: <assert w==0 && r[tid]==true;
   RelRead(); r[tid] := false;>
5: return (lx,lv);

AcqRead():
1: <AcqMutex(); m := tid;
2: <assert m==tid;
   assume write==false;>
3: <assert m==tid;
   reads := reads + 1;>
4: <assert m==tid; RelMutex();
   m := 0;>

RelRead():
1: <AcqMutex(); m := tid;>
2: <assert m==tid;
   reads := reads - 1;>
3: <assert m==tid; RelMutex();
   m := 0;>

```

**Figure 4.** The annotated version of the procedures in Fig. 3

tween `AcqRead` and `RelRead` can be executed by multiple threads concurrently. Thus, the r/w lock ensures that `Update` and `Read` may not run concurrently, and allows concurrent execution of multiple `Read` calls.

The integer variable `reads` stores the number of currently reading threads, and the boolean variable `write` indicates whether a thread is currently writing. The `assume` statement models waiting on a condition; `assume e` terminates when `e` holds, otherwise it blocks<sup>1</sup>.

The following invariant expresses the implementation of the r/w lock in the program: While `read>0` and `write==false`, the read lock is held, and while `reads==0` and `write==true` one thread has held the write lock. For the program to implement readers/writer correctly, the pair `(reads, write)` must be accessed atomically. The programmer ensures this using the mutual-exclusion idiom.

**Mutual-exclusion.** Procedures `AcqMutex` and `RelMutex` are used to acquire and release the mutex lock. We assume that `AcqMutex` and `RelMutex` have atomic bodies. The implementation ensures that no code pieces between `AcqMutex` and `RelMutex` are executed concurrently.

The programmer uses the boolean `held` variable to indicate whether the lock is currently held by a thread or not. The following invariant expresses the implementation of the mutual-exclusion in the program: While `held==false`, the lock is free, and while `held==true` one thread has held the lock. The lock is acquired and released, by setting `held` to `true` and `false`, respectively.

### 3.4 Annotating procedures

Figure 4 shows the version of procedures in Fig. 3 annotated by our technique. We now describe the annotations generated by our tool and explain how the assertions indicate noninterference while proving that the procedures `Update` and `Read` are atomic.

Let us first consider how the mutual-exclusion is expressed in the annotations. The programmer provides the annotation tool the predicate `held==true`. The predicate indicates at which states the lock is held. The tool automatically determines the acquire and the release operations and generates the annotations as assignments and assertions referring to an auxiliary variable `m`.

<sup>1</sup>The uses of `assume` at line 2 of `AcqWrite` and `AcqRead` in Figure 3 may cause deadlocks. We deliberately used `assume` to simplify the presentation. A deadlock-free implementation can be obtained by replacing `assume e` in both lines with `while(!e){RelMutex();AcqMutex();}`

First, notice that the lines 1 and 4 of `AcqRead` and `AcqWrite` are annotated with assignments to `m`. This is because the annotation engine detects, by considering the body of `AcqMutex` and `RelMutex`, that these lines acquire and release the mutex lock, respectively, thus `m` must be updated to preserve the correspondence between the value of `m` and the lock state. Then observe that the assertions make it explicit that lines 2-3 of `AcqRead` and lines 2-3 of `AcqWrite` run at different synchronization states of the mutex lock and thus do not interfere. To see how this makes it possible to prove actions to be movers, consider the following action pair executed by different threads:

```

AcqRead():
2: <assert m==tid;
   assume write==false;>

AcqWrite():
3: <assert m==tid;
   write := true;>

```

During the mover analysis, this action pair is considered as if `tid` in each action refers to a different thread id. The assertions express that this pair of actions cannot run consecutively, since they are executed when the same mutex lock is held by different threads. By using the assertions we prove that the actions in `AcqRead` and `AcqWrite` that write to and read from the `(reads, write)` pair do not interfere with each other, thus these actions are movers. Consequently, `AcqRead` and `AcqWrite` are both atomic. A similar reasoning holds for `RelRead` and `RelWrite`. Once this is provided, we can now reason about the use of the readers/writer lock in `Update` and `Read`.

In order to generate the annotations for the r/w lock, the programmer provides the predicates `reads>0 && write==false` and `reads==0 && write==true`. These predicates indicate at which states the read and write lock are held, respectively. The variable `w`, with domain  $Tid \cup \{0\}$ , is used to add auxiliary information about whether the write lock is being held (`w==tid`) or not (`w==0`). In addition, the variable `r` encodes a boolean predicate with domain  $Tid$ , and indicates whether the read lock is being held by the current thread or not. The lines that read from and write to the `(x, v)` pair are annotated with assertions that indicate whether the write and read locks are free or not, respectively.

By considering the proved-atomic bodies of `AcqWrite` and `AcqRead`, the first lines of `Update` and `Read` are detected to acquire write and the read lock, respectively, so are annotated with assignments to `w` and `r`. Proper annotations are done for the actions that release the locks. Now consider lines 2-3 of `Read` and lines 2-4 of `Update`. While the former run while the read lock is held, the latter run while the write lock is held. Therefore these lines run at different synchronization states and thus do not interfere. To see why these actions are movers, consider the following action pair executed by different threads:

```

Update(k : int):
2: <assert w==tid; x := k;>

Read():
2: <assert w==0; lx := x;>

```

Notice that since the thread id stored in `tid` cannot have the value 0, the assertions conflict with each other. Therefore, this pair of actions cannot get executed consecutively.

In addition, the write lock provides mutual-exclusion for concurrent runs of `Update`. To see how this is expressed through assertions, consider the following pair of actions, again executed by different threads:

```

Update(k : int):
2: <assert w==tid; x := k;>

Update(k : int):
2: <assert w==tid; x := k;>

```

During the mover analysis, this action pair is considered as if `tid` in each action refers to a different thread id. By a similar reasoning to mutual-exclusion above, the assertions make this action pair trivially satisfy the mover check.

### 3.5 Debugging the synchronization idiom

While helping to prove atomicity, the generated annotations also indicate for which parts of the program the synchronization idiom is effective, and thus are useful for debugging applications of the idiom. We will now show several bug examples and explain how our annotation method helps to detect and diagnose these bugs.

**Incorrect use of the idiom.** Although the programmer correctly implements a synchronization idiom, she may not obey the usage rules enforced by the idiom. For example, the readers/writer lock idiom enforces that a thread must first acquire the write lock before using it, and every release must match a previous acquire by the same thread. If the program contains a path that violates this rule, our technique is able to detect this. We do this by encoding the usage rules of the idiom as verification conditions and checking these conditions before the annotation starts. The verification condition for each usage rule has two parts:  $\mathbb{R}$  (a rely condition) and  $\mathbb{G}$  (a guarantee condition). These  $(\mathbb{R}, \mathbb{G})$  pairs are generated automatically given simple hints, referred to above, about the synchronization idiom. Each procedure is checked against these conditions. During the check,  $\mathbb{R}$  expresses an assumption that no other thread running concurrently with the procedure under the check violates the rule, while  $\mathbb{G}$  asserts that every action of the current procedure obeys the rule. For example,  $\mathbb{R}$  for the r/w lock idiom expresses that if the lock is held by the current thread, no other thread may acquire it before the current one releases it. The corresponding  $\mathbb{G}$  asserts that an action does not acquire a lock held by another thread. If it is detected that a procedure may release a lock before acquiring it, this is detected and programmer is informed about the missing acquisition. The annotation engine runs only after the program is proved to satisfy these verification conditions.

**Missing synchronization.** If the mover check for an action  $\alpha$  protected by a synchronization idiom fails, there are two possible reasons. First, annotations may be incorrect or weak to show the non-interference. Sec. 3.6 gives an example where the annotations are weak. However, our annotation scheme generates strong enough assertions to prove for atomicity reduction. The second reason is that there is a true interference between  $\alpha$  and another action in the program. This is the case if, although the program conforms to the usage rules of the idiom, the programmer omits using the idiom for an action that conflicts with  $\alpha$ . In this case the actual use of the idiom does not fit the intended use of the idiom.

For example, suppose that the example in Figure 2, using the procedures defined in Figure 3, contains an extra thread  $T_6$ , in parallel to  $T_2$ – $T_5$ , that updates the  $(x, v)$  pair without calling `AcqWrite` and `RelWrite`. In this case, the checking of the idiom’s usage rules does not fail, because  $T_6$  completely ignores using the r/w lock, which does not violate the rules of the r/w lock. However, the actions in  $T_6$  will not be annotated with assertions, which indicates missing synchronization in  $T_6$ . As a result, the blocks surrounded by `AcqWrite` and `RelWrite` will not get reduced to atomic blocks, since there is interference between the actions in these blocks and the actions of  $T_6$ ; this interference cannot be ruled out by the assertions in the former. The failing mover checks and the missing assertions help the user to identify the missing synchronization.

**Incorrect implementation of the idiom.** Missing annotations indicate either an incorrect implementation of the idiom, or insufficient atomicity level for the program. For example, suppose that the readers/writer lock implementation does not use mutual-exclusion. Therefore, the pair  $(reads, write)$  is not read and updated atomically. In this case, the annotation engine will not find the acquire and release operations, which must be atomic actions in the current program, and will not annotate the program. The missing annotations in `Update` and `Read` indicate that the procedures `AcqRead` and `AcqWrite` do not implement the intended lock operations.

```

AcqWrite():                               AcqRead():
1: <AcqMutex(); m := true;>                1: <AcqMutex(); m := true;
2: <assert m==true;                          2: <assert m==true;
   assume reads==0 && write==true;>         assume write==false;>
3: <assert m==true;                          3: <assert m==true;
   write := true;                            reads := reads + 1;
4: <assert m==true; RelMutex();              4: <assert m==true; RelMutex();
   m := false;>                               m := false;>

```

Figure 5. The annotated procedures with weaker assertions

```

Update(k : int):                           Read():
1: <AcqWrite(); w := tid;>                  1: <AcqRead(); w := tid;>
2: <assert w==tid; x := k;>                  2: <assert w==tid; lx := x;>
3: <assert w==tid; lv := v;>                  3: <assert w==tid; lv := v;>
4: <assert w==tid; v := lv + 1;>             4: <assert w==tid; RelRead();
5: <assert w==tid; RelWrite();                w := 0;>
   w := 0;>                               5: return (lx,lv);

```

Figure 6. The annotated procedures with stronger assertions

### 3.6 Naïve attempts

We conclude this section by showing two failing attempts to annotating our example program despite the fact that the example is correct. These illustrate that coming up with the right program annotations is non-trivial, and indicates the essence of automating the annotation process.

**Assertions too weak.** Let us now use a boolean auxiliary variable  $m$  to perform the annotation for the mutual-exclusion idiom as shown in Fig. 5. Predicates  $m==true$  (alternatively,  $m==false$ ) indicate the mutex lock is being held (alternatively, free). In the new scheme, the lines that call `AcqMutex` and `RelMutex` assign true and false to  $m$ , respectively.

Although the assertions are valid, this annotation is not enough to show that actions of `AcqRead` and `AcqWrite` are not simultaneously enabled, since both actions seem to run at the same synchronization state, where  $m==true$  holds for both actions. Thus, the mover check for these actions fails in this case, since the assertions do not suffice to express the fact that the lock is held by different threads. In contrary, the assertions  $m==tid$  inserted by our annotation scheme are strong enough to express this fact properly.

**Assertions too strong.** Fig. 6 gives an annotation scheme where `Update` and `Read` are both annotated with the same assertion  $w==tid$ . This makes pairs of actions having elements from `Update` and `Read` trivially satisfy the mover check.

However the assertions  $w==tid$  in `Read` indicate that no two thread may run `Read` simultaneously. In fact, these assertions are not valid as the readers/writer lock allows multiple lines of `Read` to run concurrently, violating the assertion. Consequently, these assertions will not be proved in later stages of the proof. At this point, it is difficult to distinguish whether the program contains a bug or whether the annotations contained erroneous reasoning. Our annotation scheme always generates valid assertions, which can be discharged later in the proof, preventing this difficulty. In fact, we will show in Sec. 5 that while the assertions added by our tool help in a subsequent reduction step, they are proved valid after the critical regions protected by the idiom are proved to be atomic.

## 4. Preliminaries

### 4.1 The ActionPL language

Following [4], we use a language called ActionPL to describe programs formally. The syntax of ActionPL is given in Fig. 7.

**Syntax.** An atomic statement (*Atomic* of Fig. 7) in ActionPL is expressed as a *gated action*  $\varphi \triangleright \tau$ , abbreviated as *action*. The store predicate  $\varphi$  is the *gate* of the action and represents the set of program states from which this action can execute without “going

<i>Atomic</i> :	$\alpha$	::=	$\varphi \triangleright \tau$
<i>Stmt</i> :	$s$	::=	$\alpha \mid \rho() \mid s ; s \mid s \parallel s \mid s \square s \mid s^\circ$
<i>Dynamic</i> :	$d$	::=	$\text{skip} \mid \text{error} \mid t : s \mid d ; d \mid d \parallel d$

**Figure 7.** The syntax of the ActionPL language.

wrong”. In other words, the gate represents an assertion on the pre-states of the action and going wrong corresponds to violating the assertion. The transition predicate  $\tau$  represents the set of state transitions allowed by this gated action.  $Atoms(\mathcal{P})$  returns the set of all gated actions in the program  $\mathcal{P}$ . We sometimes express transition predicates in the compact notation:  $\langle \tau \rangle_M \equiv \tau \wedge (\forall x \in Var \setminus M. x' = x)$  where  $M$  is list of variables written by  $\alpha$  and  $x'$  is a variable that refers to the value of  $x$  after the transition described by  $\tau$  is taken.

Statements in ActionPL (*Stmt* of Fig. 7) are either atomic statements, procedure calls ( $\rho()$ ) or are built from them by sequential ( $;$ ) or parallel ( $\parallel$ ) composition, non-deterministic choice ( $\square$ ) or looping ( $s^\circ$ ). We model argument passing using global variables. Dynamic statements (*Dynamic*) are used for formalizing the semantics and will be explained below.

**Semantics.** Execution of the program follows interleaving semantics in which only one atomic statement is executed at a time without being interrupted. Statements may refer to the current thread id through the special variable  $\text{tid} \in Var$ , whose domain is  $Tid$ . To represent the fact that a statement  $s \in Stmt$  is executed by a thread  $t \in Tid$ , we use the *dynamic statement*  $t : s$ . Execution of a gated action  $\varphi \triangleright \tau$  by thread  $t$  may result in one of two outcomes: if the current store satisfies the gate  $\varphi$ , the store is modified atomically consistent with the transition  $\tau$ . Otherwise, the execution *goes wrong*. An *execution* of  $s$  is a sequence of transition steps. A *terminating execution* is one that ends in special dynamic statements skip or error. An execution is said to succeed if it ends in skip and to fail (or *go wrong*) if it ends in error. The program goes wrong from  $\mathcal{I}$ , if there exists a failing execution of the program with the initial store  $\sigma$  satisfying  $\mathcal{I}$ . The complete operational semantics of ActionPL can be found in [4].

## 4.2 The QED method

The QED method [4] provides a set of proof rules and related tools for proving assertions in concurrent programs. A proof in QED consists of rewriting the input program, denoted  $\mathcal{P}_1$ , iteratively using abstraction and reduction so that, in the limit, one arrives at a program, denoted  $\mathcal{P}_n$ , that can be verified by sequential reasoning methods. Reduction, due to [11], creates coarse-grained atomic statements from fine-grained ones. Abstraction of a statement allows us to reason that it does not interfere with other atomic statements.

Formally, the proof is expressed as  $\mathcal{P}_1, \text{true} \dashrightarrow \mathcal{P}_2, \mathcal{I}_2 \dashrightarrow \dots \dashrightarrow \mathcal{P}_n, \mathcal{I}_n$  where the pair  $\mathcal{P}_i, \mathcal{I}_i$  denotes a proof context, which is defined by the current program  $\mathcal{P}_i$  and program invariant  $\mathcal{I}_i$ . Each proof step  $\mathcal{P}_i, \mathcal{I}_i \dashrightarrow \mathcal{P}_{i+1}, \mathcal{I}_{i+1}$  is governed by a proof rule which either rewrites the program, as an application of reduction or abstraction, or changes the program invariant. The proof terminates when it reaches a proof context  $\mathcal{P}_n, \mathcal{I}_n$  where the assertions in every atomic statement in  $\mathcal{P}_n$  can be discharged separately by sequential reasoning. The soundness of the method is stated as follows (see [3] for the proof): If  $\mathcal{P}_n$  does not go wrong from  $\mathcal{I}_n$ , then  $\mathcal{P}_1$  does not go wrong from  $\mathcal{I}_1$ .

The application of reduction in QED aims to enlarge the atomic code blocks in the program by combining multiple gated actions to single gated actions. For each kind of statement in Fig. 7, a separate proof rule governs the application of reduction. Each rule requires that the statement is composed of gated actions of certain mover

types. For example, a statement  $\alpha_1; \alpha_2$  can be combined to a single gated action if  $\alpha_1$  is a right-mover or  $\alpha_2$  is a left-mover.

We define the abstraction relation to define mover types.  $\varphi_2 \triangleright \tau_2$  abstracts  $\varphi_1 \triangleright \tau_1$  from  $\mathcal{I}$ , denoted  $\mathcal{I} \vdash \varphi_1 \triangleright \tau_1 \preceq \varphi_2 \triangleright \tau_2$ , if the following two conditions hold:

$$1) \models (\mathcal{I} \wedge \varphi_2) \Rightarrow \varphi_1 \quad 2) \models (\mathcal{I} \wedge \varphi_2 \wedge \tau_1) \Rightarrow \tau_2$$

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  $\mathcal{I}$ .

Figure 8 gives the rule for right-movers; the rule for left-movers is similar. In the rule,  $Concur(\alpha)$  returns the set of all actions that may execute concurrently with  $\alpha$ .  $\alpha[t]$  denotes the action  $\alpha$  where  $\text{tid}$  is substituted by  $t$  both in the gate and the transition. The premise states that, from states satisfying  $\mathcal{I}$ , a subsequent execution of  $\alpha$  and  $\beta$  by different threads is simulated by their execution in the reversed order. The judgment  $\mathcal{P}, \mathcal{I} \vdash \alpha : m$  expresses the mover type of  $\alpha$  in the proof context  $\mathcal{P}, \mathcal{I}$  where  $m \in \{\mathbb{R}, \mathbb{L}\}$ . In the rule  $\alpha \circ \beta$  stands for atomic sequential composition of  $\alpha$  and  $\beta$ .

$$\frac{\text{RIGHT-MOVER} \quad \forall t, u \in Tid. \forall \beta \in Concur(\alpha) : \quad (t \neq u) \Rightarrow (\mathcal{I} \vdash (\alpha[t] \circ \beta[u]) \preceq (\beta[u] \circ \alpha[t]))}{\mathcal{P}, \mathcal{I} \vdash \varphi \triangleright \tau : \mathbb{R}}$$

**Figure 8.** The right mover rule

Intuitively, the right-mover rule states that in any execution of the program, commuting  $\alpha$  to the right of any other following action  $\beta$  yields a new execution that is either equivalent to the original execution or one that goes wrong.

A central theme in the approach presented in this paper is the use of assertions to help reduction. Adding an assertion to an action  $\alpha$  is an abstraction, and replacing actions with more abstract actions is sound for assertion checking [4]. We use assertions to eliminate apparent interference between actions which in reality does not exist due to synchronization idioms used in the program. Adding conflicting assertions to a pair  $(\alpha, \beta)$  indicates that  $\alpha$  and  $\beta$  run at different synchronization states. As a result, the sequential composition of  $\alpha$  and  $\beta$  always goes wrong while doing the simulation check in Fig. 8 which makes the pair  $(\alpha, \beta)$  trivially satisfy the checks in Fig. 8.

In the following sections we investigate several common synchronization idioms and explain how we provide proof assistance for programs using these idioms.

## 5. The annotation method

Recapitulating Sec. 3, one of the main objectives of this paper is to have the program text and annotations reflect the synchronization idiom employed to control concurrency. Such an objective necessarily starts with the formal specification of the idiom. The specification provides the type of the idiom and some information on how each program line relates to the implementation of this idiom. The outcome of this step is a template which can be used to correctly annotate the program. The annotation can either be done by the user manually, following a proof script or automatically by a theorem prover. In this section, we will explain each step in detail, illustrating the main ideas via the running example.

### 5.1 Describing Synchronization Idioms

A synchronization idiom stands for any well-known pattern to restrict the amount of concurrency in the execution of a program. For instance, readers/writer lock is an idiom denoting the collection of implementation methods which distinguish the lines of code,

$$\begin{array}{l}
\boxed{\mathcal{S}_{\text{rwlock}}(\phi_R, \phi_W)} \\
\mathbb{V} = \{r, w\} \quad \text{Dom}(w) = \text{ Tid} \cup \{0\} \\
\quad \quad \quad \text{Dom}(r) = \wp(\text{ Tid} \times \{\text{true}, \text{false}\}) \\
\mathbb{I} = \bigwedge \left( \begin{array}{l} \phi_w \Rightarrow \neg\phi_r \\ \neg\phi_w \Leftrightarrow w = 0 \\ \phi_r \Leftrightarrow \exists t. r(t) = \text{true} \end{array} \right) \\
\mathbb{A} = \bigwedge \left( \begin{array}{l} \forall t \in \text{ Tid}. \quad t \neq \text{tid} \Rightarrow \quad r'(t) = r(t) \\ \phi_w \Leftrightarrow \phi'_w \Rightarrow \quad w' = w \\ \phi_r \Leftrightarrow \phi'_r \Rightarrow \quad r(\text{tid})' = r(\text{tid}) \\ (\neg\phi_w \wedge \phi'_r) \Rightarrow \quad r'(\text{tid}) = \text{true} \\ (\neg\phi_r \wedge \phi'_w) \Rightarrow w' = \text{tid} \wedge r'(\text{tid}) = \text{false} \\ (\phi_w \wedge \neg\phi'_w) \Rightarrow \quad w' = 0 \end{array} \right) \\
\mathbb{P} = \{ w = \text{tid}, w = 0, w \neq 0, w \neq \text{tid}, \\ \quad r[\text{tid}] = \text{true}, r[\text{tid}] = \text{false} \} \\
\mathbb{R} = \bigwedge \left( \begin{array}{l} (w = \text{tid}) \Rightarrow (w' = w) \\ r(\text{tid}) = r'(\text{tid}) \end{array} \right) \\
\mathbb{G} = \bigwedge \left( \begin{array}{l} (w \neq 0 \vee w \neq \text{tid}) \Rightarrow w' = w \\ \forall t. t \neq \text{tid} \Rightarrow r(t) = r'(t) \end{array} \right)
\end{array}$$

**Figure 9.** The idiom template for the r/w lock idiom

the critical sections, for reading and writing part of memory that cannot be executed concurrently. We formalize the idea of a synchronization idiom by specifying a set of *idiom formulas*. For the mutual-exclusion idiom, this set contains one formula:  $\phi$  over program variables. Intuitively,  $\phi$  is the predicate which becomes true only when there is at least one thread that has the mutex lock. For the readers/writer lock idiom, the set contains two formulas:  $\phi_r$  and  $\phi_w$  both of which are predicates over program variables. While  $\phi_r$  becomes true only when there is at least one thread that has a reader's lock,  $\phi_w$  becomes true exactly when there is (exactly) one thread holding the writer's lock.

The idiom formulas express how the idiom is implemented in the program. Recalling our running example, the use of the mutual-exclusion in Fig. 3 is specified by the predicate  $\phi = (\text{held} == \text{true})$ . This means the program contains atomic actions that acquire and release the lock by setting and resetting  $\phi$  to true and false, respectively. The procedures `AcqMutex` and `RelMutex` contain these actions. Our method is able to recognize these actions and do the required annotation. Similarly, the use of the r/w lock is specified by the predicates  $\phi_r = (\text{reads} > 0 \wedge \text{write} == \text{false})$  and  $\phi_w = (\text{write} == \text{true})$ . Each of procedures `AcqRead`, `RelRead`, `AcqWrite` and `RelWrite` performs an operation that corresponds to either acquiring or releasing either the read or the write lock.

## 5.2 Relating the Idiom to the Program

The idiom formulas divide the state space of the program. For example, the mutual exclusion idiom divides into two, states in which the lock is acquired and states in which the lock is free. In order to apply static proof methods to the program, the structure of the state space must be linked to the program text. We introduce now a structure, *idiom template*, establishing this link.

The idiom template is a tuple  $\langle \mathbb{V}, \mathbb{I}, \mathbb{A}, \mathbb{P} \rangle$ . The first component  $\mathbb{V}$  is the set of auxiliary variables used to relate the idiom formulas to the gated actions of the program.  $\mathbb{I}$  is the global invariant that is expected to hold if the idiom is correctly implemented and used.  $\mathbb{A}$  defines the transition relation of the auxiliary variables in  $\mathbb{V}$ . This transition relation is defined using the truth values of the idiom formulas. Finally,  $\mathbb{P}$  is a set of predicates over variables in  $\mathbb{V}$ . Each predicate in  $\mathbb{P}$  is a candidate for the final annotation of the actions in the program due to the idiom under consideration. These predicates over the program variables and the variables in  $\mathbb{V}$  encode the restrictions on concurrent execution enforced by the idiom.

The template for the r/w lock will be denoted by  $\mathcal{S}_{\text{rwlock}}(\phi_r, \phi_w)$ . The subscript and the parameters used for naming the template

show the idiom and its formulas. The user has to specify both the idiom, e.g., that it is the r/w lock, and its associated formulas,  $\phi_r$  and  $\phi_w$  for the r/w lock idiom. Figure 9 shows the elements of the constructed idiom template.

The set of the auxiliary variables,  $\mathbb{V}$ , contains two variables:  $w$  and  $r$ . The variable  $w$  can hold any value in the set  $\text{ Tid} \cup \{0\}$ . Intuitively, its value defines the state of the writer's lock: if 0, no thread has the lock; if  $t$ , then the thread with id  $t$  holds the lock. The variable  $r$  is a unary predicate over  $\text{ Tid}$ . Intuitively, the value  $r(t)$  is true only when the thread with id  $t$  holds a reader's lock. As can be seen, these variables depict whether a lock is held and if so, by which thread.

The invariant  $\mathbb{I}$ , and the transition predicate  $\mathbb{A}$  that annotates each action in the program together establish the link between the auxiliary variables, program state and transitions by making use of the idiom formulas  $\phi_w, \phi_r$ .

$\mathbb{I}$  is a conjunction of three terms. The first conjunct specifies that whenever the condition for the writer lock is satisfied, the condition for the reader lock is not. The other conjuncts relate the auxiliary variables to the idiom formulas, formalizing what we have described in the previous paragraph. The variable  $w$  is 0 iff the writer lock condition  $\phi_w$  is not satisfied. The last conjunct states that the reader lock condition  $\phi_r$  is satisfied iff there is at least one  $t$  such that  $r(t)$  holds.

Whereas in  $\mathbb{I}$ , state invariants were given,  $\mathbb{A}$  will provide the transition counterpart; that is, how auxiliary variables  $w, r$  change their values based on the change in the idiom formulas  $\phi_w, \phi_r$ . For example, the fourth conjunct states that whenever the current thread acquires the readers lock, expressed by  $\phi_w$  being false in the current state and  $\phi_r$  being true in the next state, the auxiliary variable valuation should reflect this by setting the next value of  $r(\text{tid})$  to true. Similarly, the fifth conjunct states that whenever the current thread acquires the writers lock, expressed by  $\phi_r$  being false in the current state and  $\phi_w$  being true in the next state, the auxiliary variables should reflect this by setting the next state value of  $w$  is set to  $\text{tid}$ .

The candidate annotation set  $\mathbb{P}$  contains four predicates. Each predicate represents some information about the writers lock. The first two are precise:  $w = \text{tid}$  means the writers lock is held by the current thread;  $w = 0$  means the writers lock is not held by any thread. The third and fourth are less precise: the former states that the lock is held by some unspecified thread; the latter states that the current thread is not holding the writers lock. We have found that these predicates are strong enough to correctly annotate the program code in order to prove the desired non-interference targeted by the r/w lock idiom.

Recalling our running example, each of procedures `AcqRead`, `RelRead`, `AcqWrite` and `RelWrite` performs an operation that corresponds to a transition specified in one of the lines in  $\mathbb{A}$ . As a result of the annotation, the predicates  $w == \text{tid}$  and  $w == 0$  are inserted to the code blocks in which the write lock is held and the read lock is held, respectively.

## 5.3 Annotating Programs - Template for Manual Insertion

An idiom template  $\langle \mathbb{V}, \mathbb{I}, \mathbb{A}, \mathbb{P} \rangle$  generated as explained in the previous section can be used as the basis of a QED proof script. In this section, we present a script (Fig. 10) in which the user guesses the program annotations. We do this for ease of exposition. In the next section, we explain how we automatically generate annotations.

The first phase, line (1), temporarily removes existing assertions in the program. These assertions will be restored at the end of the script in line (8). They are temporarily removed so that the proof effort in the script concentrates solely on assertions relating to the synchronization idiom under study. The added annotations should be only about the idiom; they will be checked in phase (7).

(1)	pushasserts	
(2)	auxannotate	$\mathbb{V}, \mathbb{A}$
(3)	invariant	$\mathbb{I}$
(4)	specify	$\rho_1   label_1, pre_1, post_1$
	...	...
	specify	$\rho_n   label_n, pre_n, post_n$
(5)	assert	$label_1, p_1$
	...	...
	assert	$label_m, p_m$
(6)	reduce	$\rho_1, \dots, \rho_n$
(7)	check	$\rho_1, \dots, \rho_n$
(8)	popasserts	

**Figure 10.** The proof script for adding annotations

Line (2) introduces the auxiliary variables  $\mathbb{V}$  and their transition relation  $\mathbb{A}$ . This transition relation is embedded into the program by replacing every gated action  $\text{true} \triangleright \tau$  in  $\text{Atoms}(\mathcal{P})$  with  $\text{true} \triangleright (\tau \wedge \mathbb{A})$ . This step also annotates the action with proper assignments to the auxiliary variable, as implied by the transition predicate of the action and  $\mathbb{A}$ . For example, the atomic body of `AcqWrite` together with  $\mathbb{A}$  to the `r/w` lock implies that `w==tid` at the end, so the assignment `w := tid` is appended to the actions that call `AcqWrite`.

Line (3) introduces the invariant  $\mathbb{I}$ . If  $\mathbb{I}$  is not found to be an invariant of the program, this indicates either the idiom is not correctly implemented or it is too early in the proof to apply the proof script: the program invariant is not strong enough, or the atomic blocks are not large enough.

In line (4), the user provides specifications for the procedures and the loops of the program as they concern this synchronization idiom. As such, each specification is a pair of formulas taken from the set  $\mathbb{P}$ : the precondition  $pre_i$ , which should hold when the procedure or the loop starts execution and the post-condition  $post_i$  which should hold when the execution of the said part completes. These formulas are what the user thinks will hold about the idiom under consideration. Recall that each element of  $\mathbb{P}$  is only about the synchronization idiom. This might be better pictured as a projection of the specifications for procedures and loops onto idiom relevant information. If no specification is given, both formulas are assumed to be equal to the invariant.

Line (5) is the main annotation step. The user annotates the code by specifying a gated action and how its gate should be strengthened. That is, once the user guesses  $p_i$  among those contained in  $\mathbb{P}$  as an annotation for a particular action  $\varphi \triangleright \tau$  with label  $label_i$ , `assert  $label_i, p_i$`  adds the predicate  $p_i \in \mathbb{P}$  to the gated action  $\varphi \triangleright \tau$ , i.e. replaces the action with  $(\varphi \wedge p_i) \triangleright \tau$ .

Line (6) applies Lipton's reduction iteratively. At the end of this phase, certain code blocks are combined into compound atomic actions. Line (7) checks whether the assertions inserted in line (5) are indeed correct. For each gated action in  $\rho_i$ , `check` performs a sequential analysis that verifies the gate of the action by considering the program invariant as the pre-condition. Disposing assertions by sequential reasoning in this way is made possible by the reductions performed in line (6) over the newly annotated program. If no assertion fails, the atomic blocks computed in line (6) are indeed atomic and the guessed annotations are correct. The assertions might fail either because the idiom is not correctly implemented or because the assertions are too weak, too strong or simply wrong. This failure is likely to provide valuable insight to the user about the program and the implementation of the idiom.

#### 5.4 Annotating Programs - Algorithm

The successful validation of the annotations introduced in line 5 of the preceding section depends on the usefulness of the annota-

```

tactic (instrument  $\mathbb{P}, \mathbb{R}, \mathbb{G}$ )( $\mathcal{P}, \mathcal{I}$ ):
1 // check the program without adding assertions
2  $\Psi := \text{GenerateVC}(\mathcal{P}, \mathcal{I}, \mathbb{R}, \mathbb{G})$ 
3 if ( $\Psi$  is not valid) then
4   Fail and return the original program  $\mathcal{P}$ 
5 // do instrumentation per predicate in  $\mathbb{P}$ 
6 foreach ( $p \in \mathcal{S}.\mathbb{P}$ ) do
7   // add  $p$  to gated actions as assertion
8   foreach ( $\varphi \triangleright \tau \in \text{Atoms}(\mathcal{P})$ ) do
9     if ( $(\varphi \wedge p)$  is satisfiable)
10      Replace  $\varphi \triangleright \tau$  with  $(\varphi \wedge p) \triangleright \tau$  in  $\mathcal{P}$ 
11 // remove invalidated assertions
12 while (true) do
13    $\Psi := \text{GenerateVC}(\mathcal{P}, \mathcal{I}, \mathbb{R}, \mathbb{G})$ 
14   break if ( $\Psi$  is valid)
15    $\text{Failed} := \mathcal{Z}_3(\Psi)$ 
16   foreach ( $(\varphi \wedge p) \triangleright \tau \in \text{Failed}$ ) do
17     Replace  $(\varphi \wedge p) \triangleright \tau$  with  $\varphi \triangleright \tau$  in  $\mathcal{P}$ 
18 Return the current program  $\mathcal{P}$ 

```

**Figure 11.** The inference algorithm

tions generated. As explained, failure by the user to come up with appropriate annotations during that phase is likely to be the norm as there are too many pitfalls. In this section, we will provide a remedy: we will explain a new approach which makes assertion generation completely automatic.

We modify the proof script of Fig. 10 by replacing line 5 with the following tactic:

(5) `instrument  $\mathbb{R}, \mathbb{G}$`

The tactic `instrument` expects as parameters two transition formulas,  $\mathbb{R}$  and  $\mathbb{G}$ . Intuitively, these formulas specify which state transitions related to the synchronization idiom are allowed. Both are actually automatically generated once the idiom formulas are supplied; their values are given in Fig. 9 for the `r/w` lock idiom. We have included them as parameters to explicate their import to the instrument tactic.

The formula  $\mathbb{R}$  gives a global condition that a thread can assume about the actions executed by other threads. In the case of the `r/w` lock, it consists of two conditions. The first one states that the status of a held writers lock cannot change as long as only those threads not holding the lock are executing. The second states that the readers lock cannot be assigned to or taken from a thread by some other thread.

The formula  $\mathbb{G}$ , on the other hand, gives a local condition that a thread has to satisfy each time it executes an action. Again referring to our example given in Fig. 9, there are two conditions that have to be satisfied. The first one states that if the action is executed by a thread that does not hold the writers lock which is held by some thread, the state of the writers lock cannot change. The second one states that the running thread cannot change the readers lock status of any other thread.

The pseudo-code explaining how the tactic instrument operates is given in Fig. 11. The extra parameters seen in the pseudo-code are the predicate component  $\mathbb{P}$  of the idiom template and the current proof context  $(\mathcal{P}, \mathcal{I})$ . As will be explained below, the proof context can be modified by the tactic.

The tactic instrument starts with a sanity check for the idiom formulas (lines 2-4). A verification condition (VC) generator is called for the current proof context along with the formulas  $\mathbb{R}$  and  $\mathbb{G}$ . We will skip the details of how the VC generator, `GenerateVC`, operates. Briefly, it is similar to [1], except for the following: while the weakest-precondition of each gated action  $\varphi \triangleright \tau$  is computed as  $\text{wp}(\varphi \triangleright \tau, \psi) = \forall \text{Var}'. \varphi \wedge (\tau \Rightarrow \psi')$  in the sequential case, we compute it in the concurrent case as follows:

$$\text{wp}(\varphi \triangleright \tau, \psi) = (\forall \text{Var}'. \varphi \wedge (\tau \Rightarrow \mathbb{G})) \wedge (\forall \text{Var}'. (\mathbb{R} \circ \tau \circ \mathbb{R}) \Rightarrow \psi')$$

This new form of the weakest-precondition expresses that each gated action assumes  $\mathbb{R}$  and asserts  $\mathbb{G}$ . The tactic instrument is expected to forward the VC to a theorem prover, which, in our framework, is the Z3 SMT solver [2].

The output of the VC generation routine called at line 2 is the formula  $\Psi$ . If the idiom formulas match with the idiom they specify,  $\Psi$  comes out as a valid formula. This is why a non-valid  $\Psi$  causes an abortion of the overall annotation script much like (3) of Fig. 10.

In the remaining part of the instrument (lines 6-17), a suitable annotation for each gated action is sought. First, a predicate  $p$  from the set  $\mathbb{P}$  is chosen. The only requirement for  $p$  is that it should not be weaker than another predicate not yet chosen. This is to ensure that always the strongest possible annotation is used for gated actions. Then, each gated action whose gate does not contradict with  $p$  is annotated with  $p$  (lines 8-10). Then, until a valid VC is generated, those gated actions whose modified assertions may be violated (elements of the *Failed* set at line 15, returned by Z3) are restored to the values they had at the start of the outer loop iteration (lines 16-17). This iterative removal of annotations is bound to terminate since at the very worst all the gated actions will be restored to the values they had before having been annotated by  $p$  which by a simple inductive argument can be proved to have a valid VC.

The assertions added, at the end of instrument, as annotations to the gated actions of the program carry information about the idiom. The candidate annotation predicates given by  $\mathbb{P}$  appeal to the understanding of these idioms. Furthermore, in the light of the experimental evidence (Sec. 6), we conjecture that the absence of interference targeted by the synchronization idiom is correctly captured by the assertions added by instrument.

## 5.5 Mutual-exclusion

$$\boxed{S_{\text{mutex}}(\phi)}$$

$$\begin{aligned} \mathbb{V} &= \{m\} \quad \text{Dom}(m) = \text{tid} \cup \{0\} \\ \mathbb{I} &= (\neg\phi) \Leftrightarrow (m = 0) \\ \mathbb{A} &= \bigwedge \left( \begin{array}{l} (\neg\phi \wedge \phi') \Rightarrow (m' = \text{tid}) \\ (\phi \wedge \neg\phi') \Rightarrow (m' = 0) \\ (\phi \Leftrightarrow \phi') \Rightarrow (m' = m) \end{array} \right) \\ \mathbb{P} &= \{m = \text{tid}, m = 0, m \neq 0, m \neq \text{tid}\} \\ \mathbb{R} &= (m = \text{tid}) \Rightarrow (m' = m) \\ \mathbb{G} &= (m \neq 0 \vee m \neq \text{tid}) \Rightarrow m' = m \end{aligned}$$

**Figure 12.** The idiom template for the mutual-exclusion idiom

The policy template for the mutual-exclusion idiom is given in Fig. 12. In the mutual-exclusion idiom, threads acquire locks to get exclusive access to blocks of code. In the template we represent the lock conceptually with the predicate  $\phi$ . Similarly to the r/w lock,  $\phi$  is true whenever the lock is acquired. The variable  $m$  stores either 0 or a thread id, indicating whether the lock is free or held by a thread, respectively.  $\mathbb{I}$ ,  $\mathbb{R}$ , and  $\mathbb{G}$  associate  $\phi$  with  $m$  and define the semantics of the mutual-exclusion idiom. The template  $S_{\text{mutex}}(\phi)$  is used in the proof script given Fig. 10 to generate annotations for mutual-exclusion policies. Recalling the example in Fig. 3, the idiom is implemented by procedures `AcqMutex` and `RelMutex`. Since this implementation guarantees that `held==true` holds within the critical sections, the policy  $S_{\text{mutex}}(\text{held==true})$  is provided to annotate the program.

Variations of mutual exclusion can also be specified in a common form. For example, the template in Fig. 13 can be used in cases where the conditions that hold in different critical regions are not complements of each other as in the above template.

$$\boxed{S_{\text{mutex2}}(\phi_1, \phi_2)}$$

$$\begin{aligned} \mathbb{V} &= \{m\} \quad \text{Dom}(m) = \text{tid} \cup \{0\} \\ \mathbb{I} &= (\phi_1 \Rightarrow (\neg\phi_2)) \wedge ((\phi_1 \vee \phi_2) \Leftrightarrow (m \neq 0)) \\ \mathbb{A} &= \bigwedge \left( \begin{array}{l} (\neg\phi_1 \wedge \phi'_1) \Rightarrow (m' = \text{tid}) \\ (\neg\phi_2 \wedge \phi'_2) \Rightarrow (m' = \text{tid}) \\ (\phi_1 \wedge \neg\phi'_1) \Rightarrow (m' = 0) \\ (\phi_2 \wedge \neg\phi'_2) \Rightarrow (m' = 0) \\ (\phi_1 \Leftrightarrow \phi'_1) \Rightarrow (m' = m) \\ (\phi_2 \Leftrightarrow \phi'_2) \Rightarrow (m' = m) \end{array} \right) \\ \mathbb{P} &= \{m = \text{tid}, m = 0, m \neq 0, m \neq \text{tid}\} \\ \mathbb{R} &= (m = \text{tid}) \Rightarrow (m' = m) \\ \mathbb{G} &= (m \neq 0 \vee m \neq \text{tid}) \Rightarrow m' = m \end{aligned}$$

**Figure 13.** The idiom template for a variation of the mutual-exclusion idiom

```
AcqReentrant():                               RelReentrant():
1: < if(owner == tid) {                         1: < count := count - 1;
2:   count = count + 1;                         2:   if(count == 0) {
3: } else {                                     3:   owner := 0;
4:   assume owner == 0;                         4: } >
5:   count := 1;
6:   owner := tid;
7: } >
```

**Figure 14.** Example implementation of reentrant locks

## 5.6 Reentrant locks

The policy template for the reentrant locks is given in Fig. 15. A reentrant lock idiom is similar to a mutex lock, except that it can be acquired and released by the same thread multiple times as long as acquires and the releases are properly nested. Reentrant locks are widely-used to implement monitors in modern languages like Java and C#. We provide a simple reentrant lock implementation in Fig. 14.

The implementation must keep a counter to remember how many times the lock has been acquired by the same thread, and decrease the counter at each release. In the template we represent the state of the counter with the expression  $\phi_c$ . If  $\phi_c > 0$  holds then the lock is acquired, and if  $\phi_c = 0$  then the lock is free. In the template the thread id that acquired the lock is represented by the auxiliary variables  $m$ . The template  $S_{\text{reentrant}}(\phi_c)$  is used in the proof script given Fig. 10 to generate annotations for reentrant locks policies. When the implementation in Fig. 14 is used, then the template is instantiated by providing  $\phi_c = \text{count}$ .

$$\boxed{S_{\text{reentrant}}(\phi_c)}$$

$$\begin{aligned} \mathbb{V} &= \{m\} \quad \text{Dom}(m) = \text{tid} \cup \{0\} \\ \mathbb{I} &= ((\phi_c > 0) \Leftrightarrow (m = 0)) \wedge (\phi_c \geq 0) \\ \mathbb{A} &= \bigwedge \left( \begin{array}{l} (\phi'_c = \phi_c + 1) \Rightarrow (m' = \text{tid}) \\ (\phi'_c = 0) \Rightarrow (m' = 0) \\ (\phi'_c = \phi_c) \Rightarrow (m' = m) \end{array} \right) \\ \mathbb{P} &= \{m = \text{tid}, m = 0, m \neq 0, m \neq \text{tid}\} \\ \mathbb{R} &= (m = \text{tid}) \Rightarrow (m' = m) \\ \mathbb{G} &= (m \neq 0 \vee m \neq \text{tid}) \Rightarrow (m' = m) \end{aligned}$$

**Figure 15.** The idiom template for the reentrant locks

## 5.7 Event synchronization

Different forms of the event synchronization appears in concurrent programs. A common form is the following:

$$(s_1; S; s_2) \parallel (s_1^1; W; s_2^1) \parallel \dots \parallel (s_1^k; W; s_2^k) \parallel \dots$$

```

T1: x := 0; v := 0; InitEvent();
-----
T2                                T3
1: Update(1);                      1: WaitEvent();
2: SetEvent();                      2: (x1,v1) := Read();
-----
T1: assert (x1 == 1) && (v1 == 1);

InitEvent():      SetEvent():      WaitEvent():
1: done := false; 1: done := true;  1: assume (done==true);

```

**Figure 16.** Synchronizing threads with events

$S_{\text{event}}(\phi)$

$$\begin{aligned}
\mathbb{V} &= \{e, s\} \quad \text{Dom}(e) = \text{Tid} \cup \{0\} \quad \text{Dom}(s) = \{\text{true}, \text{false}\} \\
\mathbb{I} &= \phi \Leftrightarrow ((s = \text{true}) \wedge (e = 0)) \\
\mathbb{A} &= \bigwedge \left( \begin{array}{l} (\neg\phi \wedge \phi') \Rightarrow (e' = 0 \wedge s' = \text{true}) \\ (\phi \Leftrightarrow \phi') \Rightarrow (e' = e \wedge s' = s) \end{array} \right) \\
\mathbb{P} &= \{s = \text{true}, s = \text{false}\} \\
\mathbb{R} &= ((e = \text{tid} \wedge s = \text{false}) \vee (e = 0 \wedge s = \text{true})) \Rightarrow \\
&\quad (e' = e \wedge s' = s) \\
\mathbb{G} &= ((e \neq \text{tid} \wedge e \neq 0 \wedge s = \text{false}) \vee (e = 0 \wedge s = \text{true})) \Rightarrow \\
&\quad (e' = e \wedge s' = s)
\end{aligned}$$

**Figure 17.** The idiom template for the event synchronization idiom

In this context, an event is a condition, say  $\phi$ , that is false before the event is set and is true after the event is set. The statement  $s_1; S; s_2^i$  sets the event, where statement  $S$  makes  $\phi$  true, and the statement  $(s_1^i; W; s_2^i)$  waits for the event, where statement  $W$  waits for  $\phi$  to be true, i.e.  $W$  terminates only when the event is set. We will refer to  $(s_1; S), s_1^1, \dots, s_1^k$  and  $s_2, (W; s_2^1), \dots, (W; s_2^k)$  by *before* and *after* phases of the statements, respectively.

As an example, consider the program in Fig. 16, which uses event synchronization implemented using the variable *done*. The “before” phase of T2 updates the pair  $(x, v)$  and it sets *done* to true. Then the “after” phase of T3 reads the  $(x, v)$  pair. The event prevents T2 from reading the initial values of  $x$  and  $v$ .

We handle event synchronization in  $s$  through the following steps:

1. We first annotate the gated actions in the parallel statement  $s$  with assertions using the technique described in Sec. 5.3. The policy template for event synchronization is given in Fig. 17. The predicate  $\varphi$  in the template represents the condition over the program variables representing the event. For the program in Fig. 16, for example,  $S_{\text{event}}(\text{done}==\text{true})$  is used for annotation generation. The variables  $e$  and  $s$  store the owner of the event and whether the even has been set, respectively.
2. We do reduction using the reduce all tactic, which applies a sequence of reduction rules until no more rule are applicable.
3. We then use the proof rules HOIST-L-MOVER and HOIST-R-MOVER, given in Fig 18, to hoist before and after phases out of the parallel statement  $s$ . If the before phase of a setter statement is a left-mover, HOIST-L-MOVER hoists this phase before the parallel statement. Similarly, if the after phase of a waiting thread is a right-mover, HOIST-R-MOVER hoists this phase after the parallel statement. This makes the before and after phases in  $s$  sequentially composed with each other, which is in fact enforced by the event synchronization at runtime.

Now recall the example in Fig. 16. Using the idiom template  $S_{\text{event}}(\text{done}==\text{true})$ , we add the assertion  $s==\text{false}$  to lines 1-2 of T2 and the assertion  $s==\text{true}$  to lines 1-2 of T3. The assertions allow us to show that actions of T2 are left-movers and ac-

$$\frac{\text{HOIST-L-MOVER} \quad S_1 = (\alpha; s_1) \parallel s_2 \quad \mathcal{P}, \mathcal{I} \vdash \alpha : \mathbb{L} \quad S_2 = \alpha; (s_1 \parallel s_2)}{\mathcal{P}, \mathcal{I} \dashv\vdash \mathcal{P}[S_1 \mapsto S_2], \mathcal{I}}$$

$$\frac{\text{HOIST-R-MOVER} \quad S_1 = (s_1; \alpha) \parallel s_2 \quad \mathcal{P}, \mathcal{I} \vdash \alpha : \mathbb{R} \quad S_2 = (s_1 \parallel s_2); \alpha}{\mathcal{P}, \mathcal{I} \dashv\vdash \mathcal{P}[S_1 \mapsto S_2], \mathcal{I}}$$

**Figure 18.** Rules for hoisting actions out of a parallel statement

```

FindSlot(x:int)                      InsertPair(x:int, y:int)
returns r:int                          returns r:bool
1 for (i=0; i<N; i++) {                1 i := FindSlot(x);
2 acq(M[i]);                            2 if (i == -1) {
3 if (M[i].elt==nil && !(M[i].vld)){    3 r := false; return;
4 M[i].elt := x; rel(M[i]);            4 }
5 r := i; return;                      5 j := FindSlot(y);
6 } else { rel(M[i]); }                6 if (j == -1) {
7 } r := -1; return;                  7 M[i].elt = nil;
                                        8 r := false; return;
                                        9 }

LookUp(x:int)                          returns r:bool
returns r:bool                          10 acq(M[i]);
1 for (i=0; i<N; i++) {                11 acq(M[j]);
2 acq(M[i]);                            12 M[i].vld = true;
3 if (M[i].elt==x && M[i].vld){       13 M[j].vld = true;
4 rel(M[i]; r := true; return;         14 rel(A[i]);
5 } else { rel(M[i]); }                15 rel(A[j]);
6 } r := false; return;                16 r := true; return;

```

**Figure 19.** The multiset implementation

tions of T3 are both-right-movers. Using HOIST-L-MOVER for T2 and HOIST-R-MOVER for T3, we obtain a sequential composition of the initialization by T1, T2, T3 and the assertion by T1.

## 6. Experience

In this section we give evidence about the usefulness of our techniques using programs from the literature. We applied our techniques to these programs mechanically or manually.

**Multiset.** Figure 19 shows a concurrent multiset of integers with `InsertPair` and `LookUp` 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  $(M[i].\text{lock})$ , the lock of cell  $i$ . We implemented two versions of multiset, one with the mutex locks (given in Figure 19) and another with the readers/writer lock. In the readers/writer version, procedures `FindSlot` and `InsertPair` acquire the write lock, whereas `LookUp` acquires the read lock. For this version, we used the same lock implementation in Figure 3 except for using a separate  $(\text{readers}, \text{write})$  pair for each cell  $M[i]$  in the multiset. All the annotations required for the blocks protected by locks were generated mechanically.

For the version using mutex locks, we used the policy  $S_{\text{mutex}}(M[i].\text{lock}==\text{true})$  to reduce the code blocks between calls to `acq` and `rel` to atomic actions. As for the version using the readers/writer lock, we used the policy  $S_{\text{rwlock}}((M[i].\text{readers}>0 \ \&\& \ M[i].\text{writer}==\text{false}), (M[i].\text{writer}==\text{true}))$ . As a result 36 assertions were added to the actions between `acq` and `rel` (and their  $r/w$  lock versions). The loop in Fig. 11 took 6 iterations for the mutex and 8 iterations for the  $r/w$  lock.

In both versions of multiset, we also used the policy  $S_{\text{mutex}}(M[i].\text{elt} \neq \text{nil})$ . This allowed us to capture the property that, once `FindSlot` returns an allocated slot, its `elt` and `vld` fields are not modified by other threads. Using this policy we annotated the blocks at lines 1-3 and 5-9 of `InsertPair` as acquires of the conceptual lock for  $M[i]$  and  $M[j]$  and the block at lines 10-16 as the protected blocks. `InsertPair` into a single atomic action. For more information about other details of the proof see [4].

**Boxwood.** The Boxwood system [12] makes use of the both mutex and readers/writer locks. For example, the allocation server uses the `ReadersWriterLock` class for different purposes including preventing contention on a primary mutex lock and accessing its custom cache. The blinktree uses the same class to prevent writes to its cache buffer. The implementation of the r/w lock in `ReadersWriterLock` is specified by the policy

```
S_rwlock(this.numReaders>0 && this.hasWriter==false,
this.numReaders==0 && this.hasWriter==true)
```

where `this` refers to the lock object. We found that annotating the code using a r/w policy with these formulas as parameters will help in showing that the code regions protected by `ReadersWriterLock` atomic.

**Readers/writer lock implementations.** Programs using existing implementations of the readers/writer locks idiom can be handled by our techniques. In addition to mutex locks, the `pthread` library also provides a readers/writer lock implementation. The implementation uses the `num_active` and `owner` variables similar to our `readers` and `write` variables in Figure 3; a use of the lock in a program can be specified by the policy  $S_{\text{rwl}}((\text{num\_active}==0 \ \&\& \ \text{owner}==0), (\text{num\_active}>0 \ \&\& \ \text{owner}==1))$ . The implementations in the ACE framework [14] and in [10], can be encoded similarly.

**Philo.** We found that in the `philo` benchmark from [15] the use of the mutual exclusion policy allows us to prove the atomicity of the methods. The benchmark is written in Java and uses Java monitors. Each object has a separate monitor that is locked to synchronize accesses to the fields of the object. We model this using an array `objlock` of booleans indexed by the object reference. The policy  $S_{\text{mutex}}((\text{objlock}[o]==\text{true}))$  specifies that a code block surrounded by `synchronized(o){...}` is protected by the lock of object `o` and can be annotated accordingly.

In the `philo.java` implementation, the synchronized `putForks` procedure is trivially proved to be atomic with the predicate `objlock==tid`. The justification for this predicate is that there is a single shared object from class `Table` and both its methods are synchronized. The case for the other method, `getForks`, is more involved since the call to `wait()` present in the method is an implicit release of the object lock. Thus, the initial mutex annotation reduces the method to three atomic blocks, each block starts with acquiring the implicit object lock and ends with the release of the lock. This has the side benefit of making explicit the hidden concurrency and can aid the programmer to better understand the trade-offs implied.

**Two-lock queues.** The concurrent queues in [16] and [13] use two locks, one for the head and one for the tail of the queue to reduce contention on the lock. The operations (`enqueue/dequeue`) in these implementations can be proved atomic by specifying the two locks with two separate mutual exclusion policies. In these implementations locks are modeled conceptually without giving an implementation. In this case, we model the acquire and release operations by associating the auxiliary variable `mutex` and the actions in `AcqMutex` and `RelMutex` in Section 5.5 with the corresponding conceptual lock operations.

**Device drivers.** In [4] the authors used a simple example illustrating a device driver that copies from the device registers to a cache and from the cache to a user buffer. The example had two uses of the mutual exclusion idiom. First, accesses to driver's fields were protected by a mutex lock. Second, it is ensured that at most one thread transfers data from the device registers to the cache by setting and checking a condition atomically. As a result of using the policy that specifies the mutex locks, we inserted 15 assertions, after 4 iterations of the loop in Fig. 11.

**Object initialization.** Concurrency patterns in which objects are initialized and used in parallel can be implemented using events.

A simple example is given in Section 5.7, where the variable `s` indicates whether the object has not been initialized (`NULL`) or being initialized (`INIT`) before the initialization and `FULL` after the initialization. The policy template  $S_{\text{event}}(s==\text{FULL})$  can be used to capture the synchronization for the object initialization in this case.

## References

- [1] 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*, 2005.
- [2] L. M. de Moura and N. Björner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
- [3] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. Technical Report MSR-TR-2008-99, Microsoft Research Redmond, July 2008.
- [4] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *In POPL '09: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, 2009. ACM.
- [5] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005.
- [6] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for `esc/java`. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 500–517, London, UK, 2001. Springer-Verlag.
- [7] 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.
- [8] C. Flanagan and S. Qadeer. Transactions for software model checking. *Electronic Notes in Theoretical Computer Science*, 89, 2003.
- [9] C. Flanagan and S. Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12, New York, NY, USA, 2003. ACM.
- [10] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. A fair fast scalable reader-writer lock. In *ICPP '93: Proceedings of the 1993 International Conference on Parallel Processing*, pages 201–204, Washington, DC, USA, 1993. IEEE Computer Society.
- [11] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
- [12] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.
- [13] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, New York, NY, USA, 1996. ACM.
- [14] D. C. Schmidt and S. D. Huston. *C++ Network Programming: Resolving Complexity Using Ace and Patterns (C++ in-Depth Series)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [15] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 70–82, New York, NY, USA, 2001. ACM Press.
- [16] E. Yahav and S. Sagiv. Automatically verifying concurrent queue algorithms. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.