Verifying Commit-Atomicity using Model-Checking ⋆

Cormac Flanagan

cormac@cs.ucsc.edu
Department of Computer Science
University of California at Santa Cruz
Santa Cruz, CA 95064

Abstract. The notion that certain procedures are *atomic* provides a valuable partial specification for many multithreaded software systems. Several existing tools verify atomicity by showing that every interleaved execution *reduces* to an equivalent serial execution (in which the actions of each atomic procedure are not interleaved with actions of other threads). However, experiments with these tools have highlighted a number of interesting procedures that, although atomic, are not reducible. This paper presents a more complete technique for verifying atomicity. Essentially, this technique explores non-serial and serial executions of the multithreaded system simultaneously to ensure that every non-serial execution yields the same final state as the corresponding serial execution. Using the SPIN model checker, we have applied this technique to verify the atomicity of a number of irreducible procedures that could not be handled by previous reduction-based tools for checking atomicity.

1 Multithreading and Atomicity

The development and validation of multithreaded software systems is an important yet challenging problem. In particular, standard techniques such as testing and manual code inspection are often inadequate for multithreaded systems, due to the large number of possible thread interleavings. Model checking provides a promising technique for ensuring that a system's implementation satisfies its specification under all possible thread interleavings.

A prerequisite of model checking is developing an appropriate specification. For many interesting software systems, writing a sufficiently-complete specification is non-trivial. As an example, consider the filesystem procedure `create`, which creates a new file. A specification of the exact effect of `create` on the concrete filesystem state would be quite verbose. Alternatively, we could specify the behavior of `create` on an abstraction of the filesystem state, but we would then need an abstraction invariant relating concrete and abstract states, and such abstraction invariants are also quite complex.

⋆ This work was partly supported by the NSF under Grant CCR-03411797 and by faculty research funds granted by the University of California at Santa Cruz.

For many multithreaded procedures such as `create`, the notion of *atomicity* provides a lightweight yet valuable partial specification. Informally, a procedure is atomic if for every (arbitrarily-interleaved) program execution, there is an equivalent execution with the same overall behavior where the atomic procedure is executed serially, that is, the procedure's execution is not interleaved with actions of other threads. This atomicity guarantee reduces the challenging problem of reasoning about the procedure's behavior in a *multithreaded* context to the simpler problem of reasoning about the procedure's *sequential* behavior. The latter problem is significantly more amenable to standard techniques such as testing and manual code inspection. In addition, many programming errors associated with improper synchronization can be detected as atomicity violations.

We formalize this notion of atomicity by modeling multithreaded program execution as a transition system and using two transition relations. The *standard* transition relation $\rightarrow$ interleaves steps of the various threads in an arbitrary manner. The *serial* transition relation $\mapsto$ also interleaves steps of the various threads, provided no thread is executing an atomic procedure. Once a thread enters an atomic procedure, then the serial transition relation executes that procedure to completion, without interleaved steps of other threads.

Reasoning about program behavior is much easier under the serial semantics ($\mapsto$) than under the standard semantics ($\rightarrow$), since each atomic block can be understood sequentially, without the need to consider all possible interleaved actions of concurrent threads. However, standard language implementations only provide the standard semantics ($\rightarrow$), which admits additional transition sequences and behaviors, and a program that behaves correctly according to the serial semantics may still behave erroneously under the standard semantics. Thus, in addition to being correct with respect to the serial semantics, the program should also use sufficient synchronization to ensure the atomicity of each block of code that is intended to be atomic. That is, for any program execution $\sigma_0 \rightarrow^* \sigma$ from the initial state σ_0 (where, for simplicity, we assume no thread is executing an atomic block in σ), there should exist an equivalent serial execution $\sigma_0 \mapsto^* \sigma$. We call this the *atomicity requirement* on program executions, and correctly synchronized programs should satisfy this requirement.

Over the past year, a number of tools have been developed for verifying this atomicity requirement, using techniques such as theorem proving [11], static typing systems [9, 10], dynamic analysis [8, 23], and model checking [13]. All these approaches are based on *reduction*, either Lipton's theory of reduction [16] or partial order reduction [21].

Reduction suffices to verify the atomicity of many procedures with straightforward synchronization, but is often inadequate for procedures that use more subtle synchronization. This paper introduces *commit-atomicity*, which is a more general technique for verifying atomicity. This technique is based on exploring serial and non-serial executions of the program simultaneously, and checking that both executions yield the same final state. Commit-atomicity is capable of verifying the atomicity of many procedures that cannot be handled by existing atomicity-checking tools based on reduction.

The presentation of our results proceeds as follows. The following section reviews reduction and provides an illustration of limitations of that technique. Section 3 introduces a semantics for multithreaded programs that we use as the basis for our formal development. Section 4 describes our technique for verifying commit-atomicity during model checking. Section 5 provides an evaluation of this technique using the SPIN model checker on four benchmark programs. Section 6 discusses related work, and we conclude with Section 7.

2 Reduction

The essential idea behind reduction is to transform an interleaved (non-serial) execution of an atomic procedure into a serial execution of that procedure by commuting adjacent actions of concurrent threads. For example, consider the first execution trace in the diagram below, in which one thread executes a procedure that (1) acquires a lock m, (2) reads a variable x protected by that lock, (3) updates x, and then (4) releases m. The execution of this procedure is interleaved with some actions b_1, b_2, b_3 of a second thread, which do not access x. Hence, the read and write of x by the first thread commute with the operations of the second thread. In addition, the acquire operation right-commutes and the release operation left-commutes with the operations of the second thread, as illustrated by the following diagram. Hence, via reduction, we obtain an equivalent serial execution with the same final state in which the actions of the procedure are not interleaved with operations of other threads. Thus, reduction suffices to prove that the first execution trace is serializable, that is, it has an equivalent serial trace. If every execution trace through the procedure is serializable, we say the procedure is *atomic*.

Reduction example

$$\sigma_0 \xrightarrow{acq(m)} \sigma_1 \xrightarrow{b_1} \sigma_2 \xrightarrow{rd(x,0)} \sigma_3 \xrightarrow{b_2} \sigma_4 \xrightarrow{wr(x,1)} \sigma_5 \xrightarrow{b_3} \sigma_6 \xrightarrow{rel(m)} \sigma_7$$

$$\sigma_0 \xrightarrow{b_1} \sigma'_1 \xrightarrow{acq(m)} \sigma'_2 \xrightarrow{rd(x,0)} \sigma'_3 \xrightarrow{wr(x,1)} \sigma'_4 \xrightarrow{rel(m)} \sigma'_5 \xrightarrow{b_2} \sigma'_6 \xrightarrow{b_3} \sigma_7$$

Reduction suffices to verify the atomicity of many procedures that follow straightforward synchronization disciplines. However, during our experiments with atomicity-checking tools based on reduction, we repeatedly encountered procedures that, although atomic, are not reducible. As an example, the procedure `acquire` shown on the next page uses a combination of busy-waiting and a compare-and-swap (`CAS`) operation to acquire a mutually-exclusive lock m (represented as a boolean). The operation `CAS(m,false,r)` has no effect and returns `false` if `m` $\neq$ `false`. However, if `m` $=$ `false`, then the operation `CAS(m,false,r)` swaps `m` and `r` and returns `true`.

Procedures `acquire` **and** `do_transaction`

```
void acquire() {
    boolean r := true;
    while (r==true) {
        CAS(m,false,r);
    }
}
```

```
void do_transaction() {
    while (true) {
        acquire(mutex);
        int t := data;
        release(mutex);

        // long computation
        int fdata := f(t);

        acquire(mutex);
        if (t==data) {
            data := fdata;
            release(mutex);
            return;
        }
        release(mutex);
    }
}
```

A non-serial execution of `acquire` is shown in column (a) below, in which the acquire operation performed by thread `T1` is interleaved with an operation of thread `T2` that resets `m` to `false`. This execution reduces to the serial execution of column (b), since the operation of `T2` can commute to the start of the execution sequence, before `acquire` begins.

Executions of `acquire`

(a) Non-serial execution	(b) Serial execution	(c) Non-serial execution
`T1: acquire() begins`	`T2: m := false`	`T1: acquire() begins`
`T1: r := true`	`T1: acquire() begins`	`T1: r := true`
`T1: assume r == true`	`T1: r := true`	`T1: assume r == true`
`T2: m := false`	`T1: assume r == true`	`T1: CAS(m,false,r) fails`
`T1: CAS(m,false,r) ok`	`T1: CAS(m,false,r) ok`	`T1: assume r == true`
`T1: assume r != true`	`T1: assume r != true`	`T2: m := false`
`T1: acquire() ends`	`T1: acquire() ends`	`T1: CAS(m,false,r) ok`
		`T1: assume r != true`
		`T1: acquire() ends`

Column (c) shows an alternative non-serial execution of `acquire` in which the `CAS` operation initially fails, and the busy-waiting loop iterates until the `CAS` operation succeeds. Note that since the execution of column (c) contains more instructions in column (b), we clearly cannot commute the execution of column (c) into the serial execution of column (b). Thus, even though the execution of column (c) is equivalent to the serial execution of column (b), in the sense that both executions yield the same final state, reduction is inadequate to verify this equivalence. Thus, the procedure `acquire` is atomic yet not reducible, and current atomicity checking tools based purely on reduction (using either type systems [9, 10], dynamic analysis [8, 23], or model checking [13]) cannot verify the atomicity of `acquire`. The Calvin-R tool [11] uses a combination of iterated abstraction and reduction to verify such atomicity properties, but requires additional programmer annotations to "guide" the right abstraction. Commit-atomicity is intended to verify such atomicity properties automatically.

Another example of an atomic yet irreducible procedure is the procedure `do_transaction` shown above. In this example, the global variable `data` is protected by `mutex`, and the procedure `do_transaction` updates `data` according to `data := f(data)`. However, the calculation of `f(data)` requires a long computation. To avoid holding `mutex` while computing `f(data)`, the procedure acquires `mutex`, reads `data`, and releases `mutex`. The procedure then computes `f(data)`, and, if `data` has not changed, updates `data` with `f(data)`. If `data` has changed, then the transaction is retried.

The procedure `do_transaction` is atomic, in the sense that each execution is equivalent to some serial execution. However, in every serial execution, the procedure returns during the first iteration of the loop, but there are many non-serial executions where the loop iterates many times. Thus, these non-serial executions cannot reduce to the equivalently-behaved serial executions, and so reduction is again inadequate to verify the atomicity of `do_transaction`. In contrast, commit-atomicity is capable of verifying the atomicity of both of these irreducible procedures.

3 Multithreaded Programs

To provide a formal basis for reasoning about atomicity, we start by formalizing an execution semantics for multithreaded programs. In this semantics, a multithreaded program consists of a number of concurrently executing threads, each of which has an associated thread identifier $i \in \mathit{Tid}$. The threads communicate through a shared store $\sigma \in \mathit{Store}$, and system execution starts in an initial store σ_0. The exact structure of the store is left unspecified as it is orthogonal to our development. The behavior of each thread i is specified by a partial function

$$T_i : \mathit{Store} \rightharpoonup \mathit{Store}$$

which performs a single step of that thread.

3.1 Standard semantics

The standard semantics of the entire multithreaded program is defined as a nondeterministic interleaving of steps of the various threads. The transition relation $\sigma \to \sigma'$ performs a single step of an arbitrarily chosen thread. We use $\to^*$ to denote the reflexive-transitive closure of $\to$.

Standard semantics: $\sigma \to \sigma'$

$$\sigma \to \sigma' \quad \text{iff} \quad \exists i \in \mathit{Tid}.\ T_i(\sigma, \sigma')$$

3.2 Serial semantics

We assume that each thread in the multithreaded program contains a number of atomic blocks, and that each atomic block has a particular *commit point* where, from the perspective of other threads, the entire block appears to happen atomically. We assume that for each thread i the function

$$A_i : \mathit{Store} \rightarrow \{\mathit{Outside}, \mathit{PreCommit}, \mathit{PostCommit}\}$$

indicates the *phase* of thread i, that is, whether thread i is

1. outside an atomic block (*Outside*);
2. in the pre-commit phase of atomic block (*PreCommit*);
3. or in the post-commit phase of atomic block (*PostCommit*).

This phase information might be determined by examining the program counter of thread i recorded in the store. We require that no atomic block is active in the initial state; that the phase of one thread is not affected by step of a different thread; and that each thread cannot directly transition from the post-commit phase of one atomic block to the pre-commit phase of a subsequent atomic block (it must have an intermediate state that is outside any atomic block). We formalize these requirements as follows:

- $A_i(\sigma_0) = \mathit{Outside}$ for all $i \in \mathit{Tid}$.
- if $T_i(\sigma, \sigma')$ then $\forall j \neq i.\ A_j(\sigma) = A_j(\sigma')$.
- if $T_i(\sigma, \sigma')$ then $A_i(\sigma) \neq \mathit{PostCommit}$ or $A_i(\sigma') \neq \mathit{PreCommit}$.

The relation $\mathcal{A}(\sigma)$ holds if any thread is inside an atomic block:

$$\mathcal{A}(\sigma) \quad \stackrel{\text{def}}{=} \quad \exists i \in \mathit{Tid}.\ A_i(\sigma) \neq \mathit{Outside}$$

The following *serial* transition relation $\mapsto$ is similar to the standard relation $\rightarrow$, except that a thread cannot perform a step if another thread is inside an atomic block. Thus, the serial relation $\mapsto$ does not interleave the execution of an atomic block with instructions of concurrent threads.

Serial semantics: $\sigma \mapsto \sigma'$

$$\sigma \mapsto \sigma' \quad \text{iff} \quad \exists i \in \mathit{Tid}.\ (T_i(\sigma, \sigma') \wedge \forall j \neq i.\ A_j(\sigma) = \mathit{Outside})$$

Reasoning about program behavior is much easier under the serial semantics ($\mapsto$) than under the standard semantics ($\rightarrow$) that is provided by standard language implementations. However, a program that behaves correctly according to the serial semantics may still behave erroneously under the standard semantics. Thus, in addition to being correct under the serial semantics, the program should also use sufficient synchronization to ensure the atomicity of each block of code that is intended to be atomic. That is, for any program execution $\sigma_0 \rightarrow^* \sigma$ where $\neg\mathcal{A}(\sigma)$, there should exist an equivalent serial execution $\sigma_0 \mapsto^* \sigma$. We call this the *atomicity requirement* on program executions, and correctly synchronized programs should satisfy this requirement. (The restriction $\neg\mathcal{A}(\sigma)$ avoids consideration of partially-executed atomic blocks.)

4 Model Checking Commit-Atomicity

In this section, we present an instrumented semantics that detects violations of the atomicity requirement described above. The instrumented semantics only admits execution sequences that are serializable, and goes wrong on non-serializable sequences. To determine whether a given execution sequence is serializable, the instrumented semantics extends the state space with a *shadow store* $\rho \in State$. Program operations in the pre-commit or post-commit phase of an atomic block operate as expected on the normal store σ, and do not affect the shadow store ρ. However, when an atomic block commits, the entire atomic block is executed in a serial manner on the shadow store. Thus, the shadow store reflects the serial execution of all commited atomic blocks. The shadow store is used to verify the serializability of the given execution sequence.

The instrumented transition relation $(\sigma, \rho) \Rightarrow (\sigma', \rho')$ is defined below. If no atomic block is executing on the shadow store (that is, $\neg\mathcal{A}(\rho)$), then the instrumented semantics performs a step of an arbitrary thread on the normal store. If this step is in the pre-commit or post-commit phase of an atomic block, then no action is performed on the shadow store, via the rules [PRE-COMMIT] and [POST-COMMIT]. However, if the step is the commit action of an atomic block, then the serial execution of that atomic block on the shadow store is initiated via the rule [COMMIT]. As expected, commit actions include transitions from the *PreCommit* to *PostCommit* phase of an atomic block. However, commit actions also include:

- transitions from *Outside* to *PostCommit*, where an action enters and immediately commits an atomic block;
- transitions from *PreCommit* to *Outside*, where an action commits and immediately exits an atomic block; and
- transitions from *Outside* to *Outside*, where the "atomic block" only contains a single action.

Instrumented semantics: $(\sigma, \rho) \Rightarrow (\sigma', \rho')$ **and** $(\sigma, \rho) \Rightarrow$ *wrong*

[PRE-COMMIT]

$$\frac{\neg\mathcal{A}(\rho) \quad T_i(\sigma, \sigma') \quad A_i(\sigma') = \mathit{PreCommit}}{(\sigma, \rho) \Rightarrow (\sigma', \rho)}$$

[POST-COMMIT]

$$\frac{\neg\mathcal{A}(\rho) \quad T_i(\sigma, \sigma') \quad A_i(\sigma) = \mathit{PostCommit}}{(\sigma, \rho) \Rightarrow (\sigma', \rho)}$$

[COMMIT]

$$\frac{\begin{array}{c}\neg\mathcal{A}(\rho) \\ T_i(\sigma, \sigma') \\ A_i(\sigma) \in \{\mathit{PreCommit}, \mathit{Outside}\} \\ A_i(\sigma') \in \{\mathit{PostCommit}, \mathit{Outside}\} \\ T_i(\rho, \rho')\end{array}}{(\sigma, \rho) \Rightarrow (\sigma', \rho')}$$

[SHADOW]

$$\frac{\begin{array}{c}A_i(\rho) \in \{\mathit{PreCommit}, \mathit{PostCommit}\} \\ T_i(\rho, \rho')\end{array}}{(\sigma, \rho) \Rightarrow (\sigma, \rho')}$$

[WRONG]

$$\frac{\begin{array}{c}\neg\mathcal{A}(\sigma) \quad \neg\mathcal{A}(\rho) \\ \sigma \neq \rho\end{array}}{(\sigma, \rho) \Rightarrow \mathit{wrong}}$$

Once the execution of an atomic block on the shadow store is initiated via [COMMIT], then the execution of that atomic block continues in a serial (non-interleaved) manner via the rule [SHADOW] until it completes. Thus, in any reachable instrumented state (σ, ρ), the shadow store reflects the serial execution of all commited atomic blocks. If no thread is currently inside an atomic block (that is, $\neg\mathcal{A}(\sigma)$), then we expect that the operations on the shadow store are a serialization of the operations on the normal store, and hence that $\sigma = \rho$. If the serial execution on the shadow store and the interleaved execution on the normal store yield different results (that is, $\sigma \neq \rho$), then we cannot verify that the execution sequence is serializable, and the instrumented execution *goes wrong* via the rule [WRONG].

Since atomic blocks are executed in a serial manner on the shadow store, two threads should never be simultaneously executing atomic blocks on the shadow store. Hence, we say the shadow store ρ is *well-formed* if

$$\forall i, j \in \mathit{Tid}.(A_i(\rho) \neq \mathit{Outside} \wedge A_j(\rho) \neq \mathit{Outside} \implies i = j)$$

The following lemma states that the instrumented semantics performs a sequence of interleaved operations on the normal store, and a sequence of serial operations on the shadow store.

Lemma 1. *If $(\sigma, \rho) \Rightarrow^* (\sigma', \rho')$ and ρ is well-formed then $\sigma \rightarrow^* \sigma'$ and $\rho \mapsto^* \rho'$ and ρ' is well-formed.*

Proof We prove the case where $(\sigma, \rho) \Rightarrow (\sigma', \rho')$ via a single transition by case analysis on the transition rule used. This proof generalises to longer transition sequences by induction.

- [PRE-COMMIT] or [POST-COMMIT]: Since $T_i(\sigma, \sigma')$, we have $\sigma \rightarrow \sigma'$. In addition, since $\rho' = \rho$, we trivially have $\rho \mapsto^* \rho'$ and ρ' is well-formed.
- [COMMIT]: Since $T_i(\sigma, \sigma')$, we have $\sigma \rightarrow \sigma'$. In addition, since $\neg\mathcal{A}(\rho)$, we have $\mathcal{A}_j(\rho) = \mathit{Outside}$ for all $j \in \mathit{Tid}$. Together with $T_i(\rho, \rho')$, we then have $\rho \mapsto \rho'$. Also from $T_i(\rho, \rho')$, we have $\mathcal{A}_j(\rho') = \mathit{Outside}$ for all $j \neq i$, so ρ' is well-formed.
- [SHADOW] Since $\mathcal{A}_i(\rho) \neq \mathit{Outside}$ and ρ is well-formed, we have $\mathcal{A}_j(\rho) = \mathit{Outside}$ for all $j \neq i$. Together with $T_i(\rho, \rho')$, we then have $\rho \mapsto \rho'$ and that ρ is well-formed. Finally, since $\sigma' = \sigma$, we have $\sigma \rightarrow^* \sigma'$.

□

In addition, the instrumented semantics includes all evaluation sequences possible under the standard semantics, except that the instrumented semantics records additional information in the shadow store. We assume that all atomic blocks terminate, that is, if $A_i(\rho_1) \neq \mathit{Outside}$ then there exists $\rho_2, \ldots, \rho_n$ such that $A_i(\rho_n) = \mathit{Outside}$ and $T_i(\rho_k, \rho_{k+1})$ for all $0 < k < n$.

Lemma 2. *If $\sigma \rightarrow^* \sigma'$ and atomic blocks terminate then for all well-formed ρ there exists ρ' such that $(\sigma, \rho) \Rightarrow^* (\sigma', \rho')$.*

Proof We first prove the base case where $\sigma \rightarrow \sigma'$ via a single transition because $T_i(\sigma, \sigma')$. The proof then generalises to longer transition sequences via induction.

- Suppose $\neg\mathcal{A}(\rho)$ and $\mathcal{A}_i(\sigma') = \mathit{PreCommit}$. In this case $(\sigma, \rho) \mapsto (\sigma', \rho)$ via [PRE-COMMIT].
- Similarly, suppose $\neg\mathcal{A}(\rho)$ and $\mathcal{A}_i(\sigma) = \mathit{PostCommit}$. In this case $(\sigma, \rho) \mapsto (\sigma', \rho)$ via [POST-COMMIT].
- Suppose $\neg\mathcal{A}(\rho)$ and neither of the above cases hold. That is, $\mathcal{A}_i(\sigma') \neq \mathit{PreCommit}$ and $\mathcal{A}_i(\sigma) \neq \mathit{PreCommit}$. Then $(\sigma, \rho) \mapsto (\sigma', \rho')$ via [COMMIT], where $T_i(\rho, \rho')$.
- Suppose $\mathcal{A}(\rho)$. Then there exists $i \in \mathit{Tid}$ such that $\mathcal{A}_i(\rho) \neq \mathit{Outside}$. Since atomic blocks terminate, then there exists $\rho_2, \ldots, \rho_n$ such that $A_i(\rho_n) = \mathit{Outside}$ and $T_i(\sigma, \rho_2)$ and for all $1 < k < n$, we have $A_i(\rho_k) \neq \mathit{Outside}$ and $T_i(\rho_k, \rho_{k+1})$. Hence, $(\sigma, \rho) \mapsto^* (\sigma, \rho_n)$ via a sequence of [SHADOW] transitions to a state (σ, ρ_n) where $\neg A(\rho_n)$, and one of the above cases then applies to this state.

□

Finally, any instrumented execution that does not go wrong satisfies the atomicity requirement.

Theorem 1. *If $\sigma \rightarrow^* \sigma'$ and $\neg\mathcal{A}(\sigma)$ and $\neg\mathcal{A}(\sigma')$ and atomic blocks terminate, then either*

1. *$\sigma \mapsto^* \sigma'$, or*
2. *$(\sigma, \sigma) \Rightarrow^* \mathit{wrong}$.*

Proof Since $\sigma \rightarrow^* \sigma'$, by Lemma 2 there exists ρ' such that $(\sigma, \sigma) \Rightarrow^* (\sigma', \rho')$. Since atomic blocks terminate, there exists ρ'' such that $(\sigma', \rho') \Rightarrow^* (\sigma', \rho'')$ and $\neg\mathcal{A}(\rho'')$. If $\sigma' \neq \rho''$ then $(\sigma', \rho'') \Rightarrow \mathit{wrong}$ via [WRONG], yielding case 2 of this theorem. Otherwise $\sigma' = \rho''$ and $\sigma \mapsto^* \sigma'$ by Lemma 1, yielding case 1 of this theorem. □

Thus, given any standard execution $\sigma_0 \rightarrow^* \sigma'$ (where $\neg\mathcal{A}(\sigma_0)$ and $\neg\mathcal{A}(\sigma')$ and σ_0 is well-formed), we can inspect the corresponding instrumented execution $(\sigma_0, \sigma_0) \Rightarrow (\sigma', \rho')$, which must exist by Lemma 2. If this instrumented execution does not go wrong, then by Theorem 1, we know that the original execution $\sigma_0 \rightarrow^* \sigma'$ is equivalent to some serial execution $\sigma_0 \mapsto^* \sigma'$. Thus, by using model checking to ensure that no instrumented execution goes wrong, we can therefore verify that the program satisfies the atomicity requirement.

5 Evaluation

We have applied commit-atomicity to verify several example programs that could not be handled by earlier atomicity-checking tools based on reduction. This section presents the example programs we used and reports on the performance of our verification technique.

5.1 Busy-Waiting Lock Acquire

Our first benchmark uses the busy-waiting lock acquire function described in Section 2. This benchmark contains an integer variable `data` protected by the mutex `m`. The code for each thread contains a loop that first acquires the mutex, updates `data`, and then releases the mutex. Our correctness specification is that each iteration of the loop should appear to execute atomically (and hence two threads should never update `data` at the same time). This correctness of specification is included in the code via the construct *`atomic`* { ... }. We consider two version of this benchmark in order to calibrate the ability of our technique to handle large procedures. In the first version `acquire1`, the critical section only contains a single line of code, whereas in the second version `acquire2`, the critical section contains 100 lines of code that manipulate `data`.

Busy-waiting lock acquire benchmark

Variables:

```
boolean m;
int data;
```

Initially:

```
m := false;
data := 0;
```

Code for each thread:

```
while (true) {
    atomic {
        acquire(); // see impl in Section 2

        // critical section
        // read-modify-write data

        m := false;
    }
}
```

5.2 Dekker's Mutual Exclusion Algorithm

Our second example is Dekker's algorithm, a classic algorithm for mutual exclusion between two threads that uses subtle synchronization. The critical section of each thread updates a shared variable `data`. Our correctness specification is that, because the mutual exclusion code is correct, the body of the `while` loop of each thread should appear to execute atomically. This specification is expressed using the construct *`atomic`* { ... }.

Dekker's mutual exclusion benchmark

```
Variables:

  boolean a1;
  boolean a2;
  int data;

Initially:

  a1 := false;
  a2 := false;
  data := 0;
```

```
Thread1:
  while (true) {
    atomic {
      a1 := true;
      if (¬a2) {
        // critical section
        // read, write data
      }
      a1 := false;
    }
  }
```

```
Thread2:
  while (true) {
    atomic {
      a2 := true;
      if (¬a1) {
        // critical section
        // read and write data
      }
      a2 := false;
    }
  }
```

5.3 Transaction Retry

Our third benchmark re-uses the procedure `do_transaction` from Section 2, with the requirement that each transaction should be performed atomically.

Transaction benchmark

```
Variables:

  boolean mutex;
  int data;

Initially:

  mutex := false;
  data := 0;
```

```
Code for each thread:

  while (true) {
      atomic {
          do_transaction();
      }
  }
```

5.4 Bluetooth Device Driver

The Bluetooth benchmark is a simplified model of one of the bluetooth device drivers in Windows NT described in [22]. There are two dispatch functions in this simplified device driver: `BCSP_PnpAdd` and `BCSP_PnpStop`. The function `BCSP_PnpAdd` is called by the operating system to perform I/O in the driver. The second dispatch function `BCSP_PnpStop` is called by the operating system to stop the driver. In our benchmark, one thread calls `BCSP_PnpStop`, and all the remaining threads call `BCSP_PnpAdd`.

Our correctness specification is that each dispatch function should execute atomically. In particular, each call to `BCSP_PnpAdd` should either operate normally or return immediately because the device driver is already stopped.

5.5 Experimental Results

We tested each benchmark using various numbers of concurrent threads, as shown in Figure 1. For each of the five benchmarks, we manually generated two Promela programs that capture the semantics of the benchmarks under the standard semantics ($\rightarrow$) and instrumented semantics ($\Rightarrow$), respectively. Figure 1 compares the cost of model checking these benchmarks under these two semantics. For each benchmark/threads/semantics configuration, the figure reports the size of the reachable state space and the memory and time required for model checking. An entry of "-" indicates that the SPIN model checker ran out of memory on that configuration. We performed these experiments under Windows XP on a 1.7GHz Pentium M laptop with 1GB of memory.

For each variable `x` in the original program, we declared two variables `x` and `s_x` in the Promela code for the instrumented semantics, to represent the value of `x` in the normal store and shadow store, respectively. Thus, the size of each state in the Promela code for the instrumented semantics is twice as large as for the standard semantics. In addition to this increase in the size of each state, the experimental results in Figure 1 indicate that the size of the reachable state space for the instrumented semantics is significantly larger than for the standard semantics. That is, the overhead of atomicity checking contributes to the state explosion problem on these benchmarks. However, commit-atomicity does provide a means of verifying atomicity in these benchmarks, which could not be accomplished with previous reduction-based tools. In addition, our results for the `acquire2` benchmark indicate that this technique is capable of handling moderately-large procedures (in this case containing 100 lines of code).

During our experiments, the bluetooth benchmark initially went wrong under the instrumented semantics, revealing the same synchronization bug that was discovered in [22] via an assertion violation. After fixing this bug, none of the benchmarks went wrong under the instrumented semantics, indicating that all these programs satisfy their intended atomicity properties.

6 Related Work

Lipton [16] first proposed reduction as a way to reason about concurrent programs without considering all possible interleavings. Although he focused primarily on checking deadlock freedom, reduction has subsequently been extended to support proofs of general safety and liveness properties [6, 3, 15, 4, 19].

Reduction has been applied to verify atomicity in a static type system for Java programs [10, 9]. This type system for atomicity was inspired by the Calvin-R [11] static checking tool for multithreaded programs, which relates each procedure's specification to its implementation via a combination of simulation and reduction. The Atomizer is a dynamic analysis tool for detecting atomicity violations by running an instrumented version of the program [8]. In recent work, Wang and Stoller [23] also developed several algorithms for checking atomicity dynamically. The use of model checking for verifying atomicity is being explored

Benchmark	Threads	Standard semantics (→) states	space (MB)	time (s)	Instrumented semantics (⇒) states	space (MB)	time (s)
`dekker`	2	3104	1.7	0.02	3601	1.8	0.05
`acquire1`	2	135	1.6	0.02	278	1.6	0.02
`acquire1`	3	468	1.6	0.02	1795	1.7	0.03
`acquire1`	4	4361	1.9	0.05	20935	3.4	0.16
`acquire1`	5	16369	6.0	0.15	118242	16.9	0.99
`acquire1`	6	62806	11.5	0.58	658038	113.4	7.21
`acquire1`	7	299952	70.8	4.42	-	-	-
`acquire2`	2	3335	1.7	0.03	8278	2.0	0.04
`acquire2`	3	12864	2.4	0.09	58795	5.4	0.24
`acquire2`	4	153854	45.6	1.32	714359	96.7	4.91
`acquire2`	5	541601	85.3	5.97	-	-	-
`transaction`	2	836	1.6	0.02	4730	1.9	0.05
`transaction`	3	25557	6.0	0.11	532457	78.1	2.21
`transaction`	4	826627	99.3	4.68	-	-	-
`bluetooth`	2	91	1.6	0.02	116	1.6	0.02
`bluetooth`	3	568	1.6	0.02	1187	1.6	0.03
`bluetooth`	4	4762	1.9	0.05	16383	3.1	0.09
`bluetooth`	5	47163	5.2	0.13	271111	33.1	1.46
`bluetooth`	6	527668	48.6	1.79	-	-	-

Fig. 1. Summary of benchmark programs and model checking performance.

by Hatcliff *et al* [13], and they present two approaches, based on Lipton's theory of reduction and partial order reductions, respectively. Their experimental results suggest that verifying atomicity via model-checking is feasible for unit-testing. All of these approaches can only verify the atomicity of reducible procedures, and thus are insufficient for the examples considered in this paper.

Atomicity is a semantic correctness condition for multithreaded software. It is related to strict serializability [20], a correctness condition for database transactions, and linearizability [14], a correctness condition for concurrent objects. It is possible that techniques for verifying atomicity can be leveraged to develop lightweight checking tools for related correctness conditions.

Many other researchers have proposed using atomicity as a language primitive, essentially implementing the serial semantics $\mapsto$. Lomet [18] first proposed the use of atomic blocks for synchronization. The Argus [17] and Avalon [7] projects developed language support for implementing atomic objects. Persistent languages [1, 2] attempt to augment atomicity with data persistence in order to introduce transactions into programming languages. A more recent approach to supporting atomicity uses lightweight transactions implemented in the run-time system [12]. An alternative is to generate synchronization code automatically from high-level specifications [5].

7 Conclusion

In an effort to avoid errors due to unexpected interactions between concurrent threads, programmers often design procedures that are intended to be atomic. Reduction suffices to verify the atomicity of procedures that use straightforward synchronization, but is often inadequate for more subtle synchronization disciplines.

This paper introduces a novel technique called commit-atomicity for verifying atomicity in multithreaded programs. This technique is based on executing serial and non-serial versions of the programs simultaneously, and checking that both versions yield the same final state. This technique is capable of verifying atomicity of variety of procedures, including procedures that could not be handled using existing atomicity-checking tools based on reduction.

Commit-atomiciy does introduce a significant model checking overhead. An important area for future research is the development of hybrid atomicity-checking tools that use reduction to verify many procedures, but is capable of leveraging commit-atomicity as necessary to verify procedures that use more complicated synchronization disciplines.

References

1. M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, 1981.
2. M. P. Atkinson and D. Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, 1985.
3. R.-J. Back. A method for refining atomicity in parallel algorithms. In *PARLE 89: Parallel Architectures and Languages Europe*, volume 366 of *Lecture Notes in Computer Science*, pages 199–216. Springer-Verlag, 1989.
4. E. Cohen and L. Lamport. Reduction in TLA. In *Proceedings of the International Conference on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1998.
5. X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *International Conference on Software Engineering*, pages 442–452, 2002.
6. T. W. Doeppner, Jr. Parallel program correctness through refinement. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 155–169, 1977.
7. J. L. Eppinger, L. B. Mummert, and A. Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann, 1991.
8. C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, 2004.
9. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 338–349, 2003.
10. C. Flanagan and S. Qadeer. Types for atomicity. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*, pages 1–12, 2003.

11. S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. In *Workshop on Formal Techniques for Java-like Programs*, 2003.
12. T. L. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 388–402, 2003.
13. J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*, 2004.
14. M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
15. L. Lamport and F. B. Schneider. Pretending atomicity. Research Report 44, DEC Systems Research Center, 1989.
16. R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.
17. B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the Symposium on Operating Systems Principles*, pages 111–122, 1987.
18. D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *Language Design for Reliable Software*, pages 128–137, 1977.
19. J. Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Springer-Verlag, 2001.
20. C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, 1986.
21. D. Peled. Combining partial order reductions with on-the-fly model-checking. In D. Dill, editor, *Proceedings of the IEEE Conference on Computer Aided Verification*, Lecture Notes in Computer Science 818, pages 377–390. Springer-Verlag, 1994.
22. S. Qadeer and D. Wu. Debugging concurrent programs with sequential analysis. 2003. Submitted for publication.
23. L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Proceedings of the Workshop on Runtime Verification*, volume 89(2) of *Electronic Notes in Computer Science*. Elsevier, 2003.