RV'03 Preliminary Version

Run-Time Analysis for Atomicity [1]

Liqiang Wang Scott D. Stoller

Computer Science Dept., State University of New York at Stony Brook, USA

Abstract

Writing and debugging concurrent (shared-variable) programs is notoriously difficult. This motivated the development of numerous static analysis and run-time analysis techniques designed to (help) ensure that concurrent programs satisfy common correctness requirements for concurrent programs, such as absence of race conditions and absence of deadlocks. This paper focuses on another common correctness requirement for concurrent programs, namely, atomicity, which requires that any set of concurrent invocations of designated procedures is equivalent to performing those invocations serially in some order. Run-time analysis algorithms for detecting violations of atomicity are presented. The algorithms vary in cost and precision.

1 Introduction

Writing and debugging concurrent (shared-variable) programs is notoriously difficult. Indeed, one could argue that shared variables are an undesirably low-level interaction mechanism, and that concurrent programs should be written using higher-level communication mechanisms. However, many existing concurrent programs use shared variables, and many more programs that use shared variables will surely be written before a paradigm shift occurs. Therefore, research that facilitates the development of correct concurrent (shared-variable) programs will remain valuable for the foreseeable future.

Two properties that many concurrent programs are expected to satisfy are race-freedom (*i.e.*, absence of race conditions) and absence of deadlocks. A *race condition* occurs when two threads concurrently access a shared variable and at least one of the accesses is a write. A *deadlock* occurs when all threads are blocked, each waiting for some action by one of the other threads. Type systems provide an effective way to ensure absence of such errors. For example, the type systems in [FF00,BR01,Gro03] ensure absence of race conditions, and the type system in [BLR02] ensures absence of both race conditions and

[1] This work is supported in part by NSF under Grant CCR-9876058 and by ONR under Grants N00014-01-1-0109 and N00014-02-1-0363. Email: {liqiang,stoller}@cs.sunysb.edu Web: http://www.cs.sunysb.edu/˜{liqiang,stoller}

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

deadlocks. However, these type systems require manual annotation of the program and, like all type systems, are sometimes undesirably restrictive (*i.e.*, the type checker rejects some well-behaved programs as well as all badly-behaved program).

Run-time analysis provides an automatic way to check race condition and deadlock. Run-time analysis can determine whether a given trace (of a concurrent program) satisfies a given property expressed in temporal logic [HR01,HR02]. A *trace* is a sequence of events, usually recorded during execution of a program. Race-freedom and absence of deadlocks are safety properties that can be checked using general-purpose run-time analysis algorithms. Nevertheless, specialized techniques for checking these properties have been developed. For example, the lockset [SBN$^+$97] and goodlock [Hav00] algorithms are specialized run-time analysis algorithms for detecting race conditions and deadlocks, respectively. Run-time analysis is less powerful than type-based approaches, because it cannot ensure correctness of the system in traces other than the observed ones. But it is automatic, which is a significant practical advantage.

The benefit of these specialized algorithms is that they indicate not only whether the given trace violates the property, but also whether the trace contains suspicious behavior which suggests that other traces of the same program might violate the property. For example, the lockset algorithm reports suspicious behavior if some shared variable is not protected consistently by some lock throughout the entire trace. The goodlock algorithm reports suspicious behavior if the order in which locks are acquired and released is not consistent with some partial order on the set of locks; adhering to such a partial order is a well-known technique for deadlock prevention. Thus, these specialized techniques can provide useful feedback (reports of suspicious behavior) even if an actual violation of the property is not encountered. This is a great advantage, because occurrence of a violation is typically much rarer and more schedule-dependent than occurrence of suspicious behavior.

We propose specialized run-time analysis algorithms for checking atomicity. Atomicity is well known in the context of transaction processing, where it is sometimes called *isolation* (the "I" in ACID). [2] Informally, atomicity is the property that every concurrent execution of a set of transactions is equivalent to some serial execution of the same set of transactions. Two executions are equivalent if each read operation has the same predecessor write, and the final write to each shared variable is the same in the two executions. For example, consider transactions t_1 and t_2, where $\mathrm{R}(x)$ and $\mathrm{W}(x)$ denote a read and write of variable x, respectively.

(1) $$t_1 : \mathrm{R}(x)\ \mathrm{W}(x) \qquad t_2 : \mathrm{R}(x)$$

[2] ACID is the abbreviation of Atomicity, Consistency, Isolation and Durability. There is an unfortunate clash in terminology: atomicity in ACID means either all operations of the transaction are reflected properly, or none are. It is quite different from atomicity of this paper.

All feasible traces are

$$
\begin{array}{ll}
 & tr_1 : t_1.\mathrm{R}(x)\ t_1.\mathrm{W}(x)\ t_2.\mathrm{R}(x) \\
(2) & tr_2 : t_1.\mathrm{R}(x)\ t_2.\mathrm{R}(x)\ \ t_1.\mathrm{W}(x) \\
 & tr_3 : t_2.\mathrm{R}(x)\ t_1.\mathrm{R}(x)\ \ t_1.\mathrm{W}(x)
\end{array}
$$

tr_1 and tr_3 are already serial. tr_2 is equivalent to tr_3 because $t_1.\mathrm{R}(x)$ and $t_2.\mathrm{R}(x)$ read the same initial value of x in both traces. Thus, t_1 and t_2 are atomic.

Concurrent programs, like transactions, are often intended to satisfy atomicity properties. For example, consider an implementation of an abstract data type (ADT), whose interface comprises several methods, the execution trace of each method can be regarded as a transaction. If the ADT is designed to support concurrency, the correctness requirements typically include some form of atomicity, *i.e.*, that every set of concurrent invocations of the interface methods is equivalent to performing the invocations serially in some order. For example, linearizability [HW90] is a special case of strict serializability. We do not consider nested transactions: if the implementation of a method m_1 in the API calls a method m_2 in the API, events in such an invocation of m_2 are considered part of the transaction corresponding to the enclosing invocation of m_1. Transactions typically correspond to method invocations, but this is not a requirement: transactions may begin and end at arbitrary points indicated by the user.

Flanagan and Qadeer developed a type system for expressing and ensuring atomicity [FQ03b,FQ03a]. The trade-offs between type systems and run-time analysis for atomicity are the same as for race conditions.[3] Our run-time analysis algorithms do not merely look for violations of atomicity in a given trace. They also attempt to determine whether a violation is possible in other traces.

Our first algorithm is inspired by Flanagan and Qadeer's type system [FQ03b], which in turn is inspired by Lipton's reduction theorem [Lip75]. Our algorithm first determines how locks are used to protect shared variables, uses this information to infer commutativity properties of events, and then checks whether the commutativity properties ensure atomicity.

Our second algorithm is based on the idea of checking whether a violation of atomicity is possible in traces obtained from a given trace by permuting the order of events. Of course, arbitrary permutations of a trace might not be feasible behaviors of the program. Our algorithm considers only the permutations that are consistent with the synchronization events in the trace. For example, if a thread holds a lock m when it executes an event e, then permutations that put e between an acquire and release of m by a different thread are infeasible and are not considered by our algorithm.

[3] Their type system for atomicity does not by itself impose a significant annotation burden on the user, but it must be used together with a type system for preventing race conditions, which does impose a non-negligible annotation burden.

The fact that one rather than another feasible permutation was actually observed reflects vagaries of scheduling and has little significance. Therefore, given a trace, we split it into sub-traces that correspond to “transactions” and then check whether there exists a non-serializable interleaving of these transactions that is consistent with the synchronization. Continuing the ADT example, given a trace containing multiple (possibly concurrent) top-level invocations of methods of the ADT by a client, we extract a set of sub-traces, which we call *transactions*, each comprising the sequence of events that occurred during execution of one such invocation. This set of transactions is the input to our run-time analysis algorithm. Currently, our algorithms take into account only synchronization implemented with locks, but extending them to consider other synchronization mechanisms should not be difficult.

2 Related Properties

Race-freedom and atomicity are incomparable properties, *i.e.*, neither implies the other. For an example that is race-free but not atomic, consider a graphics package with a Rectangle class that offers synchronized methods `setX(int x)` and `setY(int y)` and an unsynchronized method `setXY(int x, int y)` whose body is simply `setX(x); setY(y)`. When thread θ_1 calls `setXY(100,100)`, thread θ_2 calls `setXY(200,200)`, they can interleave in the following way:

$$(3)\qquad \begin{array}{lllll} \theta_1 : & \texttt{setX}(100) & & & \texttt{setY}(100) \\ \theta_2 : & & \texttt{setX}(200) & \texttt{setY}(200) & \end{array}$$

The result is `x=200, y=100`. This is different from the result of any serial execution. Thus, concurrent invocations of `setXY` are not atomic, even though this class is race-free. For an example that is atomic despite having a race condition, consider an unsynchronized method `getX()`. When `getX()` executes concurrently with `setX(int x)`, a race condition occurs, but the invocations are atomic.

Artho *et al.* developed a run-time analysis algorithm for detecting *high-level data races* [AHB03]. Informally, absence of high-level data races seems similar to atomicity. They introduce a concept of *view consistency* which is utilized to detect high-level data races. A *view* is the entire set of shared variables accessed in a synchronized block (*i.e.*, the sequence of events from an acquire to the corresponding release). The *generated views set V(t)* of a thread t is the set of all views generated by t. Thread t_1 and thread t_2 are view consistent if the intersections of all views of V(t_1) with the maximal view of V(t_2) form a chain (with respect to the subset ordering $\subseteq$), and vice versa. View consistency and atomicity are incomparable (*i.e.*, neither implies the other). We demonstrate this with examples. When determining atomicity in these examples, we regard each execution of a thread as a transaction. Let acq(ℓ) and rel(ℓ) denote an acquire and release of lock ℓ, respectively. Consider the following sequences of events executed by threads θ_1 and θ_2. Any feasible trace obtained by interleaving these sequences of events is atomic but not view consistent, because $\mathrm{V}(\theta_1)=\{\{x,y\}\}$, $\mathrm{V}(\theta_2)=\{\{x\},\{y\}\}$, $\{x,y\}\cap\{x\}=\{x\}$, $\{x,y\}\cap\{y\}=\{y\}$, $\{x\}$ and $\{y\}$ do not form a chain.

(4) θ_1 : acq(ℓ) R(x) R(y) rel(ℓ) θ_2 : acq(ℓ) R(x) rel(ℓ) acq(ℓ) R(y) rel(ℓ)

The following example is view consistent but not atomic.

(5) θ_1 : acq(ℓ) W(x) rel(ℓ) θ_2 : acq(ℓ) R(x) rel(ℓ) acq(ℓ) W(x) rel(ℓ)

3 Background

We review the standard notion of serializability [BHG87] and then introduce our notion of atomicity.

A *transaction* is a sequence of events executed by a single indicated thread. A *trace* is a sequence of events, each labeled to indicate which transaction it is part of.

Two traces are *equivalent* (more precisely, *view equivalent*) iff (*i*) they are merges of the same set of transactions, (*ii*) each read operation has the same predecessor write event in both traces (thus, it reads the same value in both traces), and (*iii*) each data item has the same final write event in both traces.

A trace is *serial* if, for each transaction, the operations of that transaction form a contiguous subsequence of the trace.

A trace is *serializable* (more precisely, *view serializable*) if there exists a serial trace that is equivalent to it.

Given a set T of transactions, a trace for T is an interleaving of the events in transactions in T such that (*i*) different transactions of the same thread are not interleaved with each other, and (*ii*) the interleaving is consistent with the synchronization events in the trace. The details of (*ii*) depend on which synchronization mechanisms are considered. Currently, we consider only locks, as described in Section 1.

A set T of transactions is *atomic* if every trace for T is serializable.

4 Reduction-Based Algorithm

This algorithm is inspired by [FQ03b], which is inspired by [Lip75].

An event is a *left-mover* if, whenever it appears immediately after an event of a different thread, the two events can be swapped without changing the resulting state. *Right-mover* is defined similarly. An event is a *both-mover* if it is a left-mover and a right-mover. Events not known to be left or right movers are *non-movers.*

Given an arbitrary interleaving of events in a set T of transactions, if we can move all events of each transaction together (by repeatedly swapping adjacent events in the trace) without changing the results of reads and without changing the final writes, then T is atomic, because the resulting trace is serial and equivalent to the original trace. If some transactions contain two or more non-movers, the non-movers could interleave with non-movers in other transactions, preventing us from moving the events of each transaction together. If each transaction t in T has at most one non-mover e, and each event in t that precedes e can be moved to the right (towards e), and each event in t that follows e can be moved to the left (towards e), then we can move all events of each transaction together. This motivates the following theorem.

Theorem 4.1 *Given a set T of transactions, T is atomic if each transaction has the form $(R+B)^*N^?(L+B)^*$,*[4] *where L, R, B, and N denote the sets of left-movers, right-movers, both-movers, and non-movers, respectively.*

Proof. This is a simple variant of Lipton's reduction theorem [Lip75]. □

Note that the converse does not hold. Thus, this theorem provides an approximate and conservative test for atomicity.

A conservative approximation of the commutativity properties of events can conveniently be obtained based on synchronization. An access to a variable is *race-free* if it is not involved in any race condition, as defined in Section 1.

Theorem 4.2 *Acquire events are right-movers. Release events are left-movers. Race-free reads and race-free writes are both-movers.*

Proof. A proof sketch follows; for details, see [FQ03b]. An acquire event e right-commutes with any immediately following event e' of a different thread; note that e' may be an access to a shared variable or an operation on a different lock but cannot be a successful operation on the same lock (because an acquire would block, and a release would be unsuccessful). For similar reasons, release events are left-movers. Race-free reads and race-free writes are both-movers, because race-freedom implies that an immediately following or immediately preceding event by another thread cannot be an access to the same variable. □

For a simple conservative test of whether an access is race-free, we consider how locks are used. (Other synchronization mechanisms could also be considered.) Let held(e) be the set of locks held by thread θ when it executes event e. The test relies on a notion of "initialization" of a variable. This notion must have the property that concurrent accesses to a variable be impossible during initialization of it.[5] For example, initialization of a heap-allocated object may be defined to end when the object escapes from the thread that allocated it; run-time analysis or static analysis can be used to determine when that occurs.

Theorem 4.3 *A read e_R of variable x is race-free if it is part of initialization of x or, for all writes e_W to x by other threads,* $\mathrm{held}(e_R) \cap \mathrm{held}(e_W) \neq \emptyset$. *A*

[4] This form is a regular expression, $N^?$ means zero or one N, R+B means R or B, $(R+B)^*$ means any number of (R+B).

[5] The notion of initialization used in Eraser [SBN+97] does not satisfy this requirement.

write e_W of x is race-free if it is part of initialization of x or, for all events e that are accesses to x by other threads, $\mathrm{held}(e_W) \cap \mathrm{held}(e) \neq \emptyset$.

Proof. Straightforward. □

The following theorem embodies a less precise and less expensive test based on a common locking discipline [SBN+97].

Theorem 4.4 *Let E be the set of events that access a variable x, excluding accesses during initialization of x. If $\bigcap_{e \in E} \mathrm{held}(e)$ is non-empty, then all accesses to x are race-free.*

Proof. Straightforward. □

To see that this test is less precise, consider the set of transactions

(6) $$t_1 : \mathrm{acq}(\ell)\ \mathrm{W}(x)\ \mathrm{rel}(\ell)\ \mathrm{R}(x) \qquad t_2 : \mathrm{acq}(\ell)\ \mathrm{R}(x)\ \mathrm{rel}(\ell)$$

Using Theorem 4.3, we can conclude that all of these accesses are race-free and that this set of transactions is atomic. Theorem 4.4 is unable to support this conclusion, because t_1 reads x without holding any locks.

Our reduction-based algorithm for conservatively detecting violations of atomicity is as follows: First, determine which variable accesses are race-free (based on Theorem 4.3 or 4.4). Second, classify all events based on Theorem 4.2 (non-race-free reads and non-race-free writes are conservatively classified as non-movers). Third, check whether each transaction has the form in Theorem 3.

Let E be the total number of events in the transactions, and let L be the number of locks. With a simple implementation of sets of locks, the worst-case running time is $O(E^2L)$ if the test embodied in Theorem 4.3 is used, and $O(EL)$ if the test embodied in Theorem 4.4 is used.

5 Block-Based Algorithm

This section presents a more complicated and more expensive but significantly more precise algorithm. It decomposes the problem of checking atomicity of a set of transactions into many smaller problems, each of which requires checking atomicity of two blocks; a block is, roughly, a pair of operations. Simple transactions have the form $(R + B)^* N^? (L + B)^*$ required by Theorem 3, but many serializable transactions do not have this form, causing the reduction-based algorithm to report false alarms. The smallest example of this is

(7) $$t_1 : \mathrm{W}(x)\ \mathrm{W}(x) \qquad t_2 : \mathrm{W}(x)$$

All three events are involved in race conditions. Transaction t_1 has the form NN, so the reduction-based algorithm reports a possible violation of atomicity. The block-based algorithm shows that this set of transactions is atomic.

Two more examples for which the reduction-based algorithm reports a possible violation of atomicity, while the block-based algorithm correctly determines that the set of transactions is atomic, are

(8) $t_1 : \mathrm{acq}(\ell)\ \mathrm{R}(x)\ \mathrm{rel}(\ell)\ \mathrm{W}(y) \qquad t_2 : \mathrm{acq}(\ell)\ \mathrm{R}(x)\ \mathrm{rel}(\ell)\ \mathrm{W}(y)$

(note that both transactions have the form *RBLN*) and

(9)
$$\begin{array}{l} t_1 : \mathrm{acq}(\ell)\ \mathrm{W}(x)\ \mathrm{R}(x)\ \mathrm{rel}(\ell)\mathrm{acq}(\ell)\ \mathrm{W}(x)\ \mathrm{rel}(\ell) \\ t_2 : \mathrm{acq}(\ell)\ \mathrm{W}(x)\ \mathrm{R}(x)\ \mathrm{rel}(\ell)\mathrm{acq}(\ell)\ \mathrm{W}(x)\ \mathrm{rel}(\ell) \end{array}$$

(note that both transactions have the form *RBBLRBL*).

A simple way to combine the two algorithms is to first run the reduction-based algorithm on a given set of transactions. If it indicates possible violations of atomicity, then run the more expensive and more precise block-based algorithm on the same set of transactions. If it also indicates possible violations, then report the possible violations to the user.

We build up to the general block-based algorithm in four steps. First, we present a subroutine to determine feasible interleavings of events. Second, we present a block-based algorithm to determine atomicity of sets of transactions that access a single variable. Third, we present a block-based algorithm for sets containing exactly two transactions, which may access multiple variables. Finally, we present the general block-based algorithm.

5.1 Determining Feasible Interleavings of Events

The block-based algorithms require a subroutine that determines feasible interleavings of events, *i.e.*, interleavings consistent with the recorded synchronization operations. Specifically, a subroutine is needed for the problem: given two events e_1 and e_2 from a transaction t and an event e_3 from another transaction t', determine whether e_3 can occur between e_1 and e_2.

To simplify the problem, we assume that locking is properly nested, *i.e.*, no transaction contains a subsequence of the form like $\mathrm{acq}(\ell_1)\ \mathrm{acq}(\ell_2)\ \mathrm{rel}(\ell_1)\ \mathrm{rel}(\ell_2)$ (ℓ_1 and ℓ_2 are not necessarily contiguous). Java, and other languages with monitor-based synchronization, statically enforce properly nested synchronization. Even in other languages, properly nested synchronization is recognized as good style and is widely used. This assumption is used only to simplify the test for feasibility of interleavings; no other aspect of our algorithms depend on this assumption.

Clearly, a necessary condition is $h_{12} \cap \mathrm{held}(e_3) = \emptyset$, where h_{12} is the set of locks held continuously from before e_1 until after e_2. An interesting observation is that, if t and t' have no potential deadlocks, then the condition $h_{12} \cap \mathrm{held}(e_3) = \emptyset$ is both necessary and sufficient. Transactions t and t' have a *potential for deadlock* [Hav00] if they acquire two locks ℓ_1 and ℓ_2 in different orders without first acquiring some other lock that would prevent their attempts to acquire ℓ_1 and ℓ_2 from being interleaved in a way that causes deadlocks. Note that "potential for deadlock" really means that deadlock appears possible based on the information captured in the trace; aspects of the program behavior not captured in the trace (*e.g.*, synchronization using constructs other than locks) might actually prevent the deadlock.

Theorem 5.1 *Let e_1 and e_2 be events in a transaction t. Let e_3 be an event in a transaction t'. Suppose t and t' do not have a potential for deadlock. A schedule in which e_3 occurs between e_1 and e_2 is feasible iff $h_{12} \cap \mathrm{held}(e_3) = \emptyset$, where h_{12} is the set of locks held continuously from before e_1 until after e_2.*

Proof. See [WS03]. □

Thus, for systems required to be free of potential for deadlocks, we may use the goodlock algorithm [Hav00] to check this condition and, if a potential for deadlock is found, report it to the user and wait for it to be eliminated before checking atomicity. When no potential for deadlock remains, the block-based algorithms for checking atomicity can use the simple and inexpensive test in Theorem 5.1 to determine feasibility of interleavings.

For systems that may contain potential for deadlock, the condition $h_{12} \cap \mathrm{held}(e_3) = \emptyset$ is not sufficient to ensure feasibility of the interleaving, as the following example demonstrates.

$$\begin{array}{l} t : \mathrm{acq}(\ell_1)\ \mathrm{acq}(\ell_2)\ \mathrm{rel}(\ell_2)\ e_1\ e_2\ \mathrm{rel}(\ell_1) \\ t' : \mathrm{acq}(\ell_2)\ \mathrm{acq}(\ell_1)\ \mathrm{rel}(\ell_1)\ e_3\ \mathrm{rel}(\ell_2) \end{array} \qquad (10)$$

Note that $h_{12} = \{\ell_1\}$ and $\mathrm{held}(e_3) = \{\ell_2\}$, so $h_{12} \cap \mathrm{held}(e_3) = \emptyset$. Nevertheless, e_3 cannot occur between e_1 and e_2. To see this, consider two cases. If t' acquires and releases ℓ_1 before t does, then t' must also acquire and release ℓ_2 before t does, so e_3 occurs before e_1. If t' acquires and releases ℓ_1 after t does, then e_3 occurs after e_2.

Intuitively, the situation is that an acquire event e' in t' gets pushed to the side (*i.e.*, away from e_1 and e_2) by a conflict with an acquire event in t, and e_3 gets pulled away together with e, because e' and e_3 they are "bound together" by some lock (in this example, ℓ_1) that is held continuously from e' until e_3 and that conflicts with some other acquire events in t.

Interleaving of e_3 between e_1 and e_2 is feasible iff (*i*) $h_{12} \cap \mathrm{held}(e_3) = \emptyset$ and (*ii*) the kind of situation just described cannot arise. Formulating the exact condition in which this kind of situation cannot arise is conceptually straightforward, but the details are complicated and are left for future work. As a first step, we worked out the details of the condition for a restricted case, namely, when at most two locks are held concurrently by a single thread (*i.e.*, for all events e, $|\mathrm{held}(e)| \leq 2$) [WS03]. The experience in [BLR02, Section 12] suggests that this case is fairly common.

The block-based atomicity algorithms in the following subsections use the subroutine for checking feasibility of interleavings mostly as a black box, except that blocks must be defined so that they contain the synchronization information needed by the selected subroutine. For example, the test in Theorem 5.1 requires that h_{12} be included in the block containing e_1 and e_2, and that $\mathrm{held}(e_3)$ be included in blocks containing e_3. Subroutines embodying more general tests (*i.e.*, tests that do not assume the absence of potential for deadlock) require that additional sets and sequences of locks be included in blocks.

5.2 Algorithm for Transactions that Access One Variable

Unserializable Patterns.
Given a set T of transactions, the algorithm looks for small patterns of events in transactions in T that can be interleaved in an unserializable way. T is atomic iff such a pattern is not found. Specifically, the algorithm checks whether:

UP1. a read from one transaction can occur between two writes in another transaction.

UP2. a write in one transaction can occur between two reads in another transaction.

UP3. a write in one transaction can occur between a write and a subsequent read in another transaction.

UP4. the final write in one transaction can occur between a read and a subsequent write in another transaction.

A trace containing an instance of any of these patterns is not serializable, so T is not atomic. This can be checked by considering both possible serial schedules (a pattern contains events from two transactions t and t', so there are only two possible serial schedules, differing in which transaction occurs first) and verifying that they are not equivalent to the pattern.

The algorithm looks for these unserializable patterns by considering pairs of "blocks". Informally, a block is a pair of read or write events from a single transaction, together with the synchronization information needed to determine possible interleavings involving those events. As mentioned in Section 5.1, the choice of synchronization information depends on which test for feasbility of interleavings will be used. For concreteness, we assume here that the test in Theorem 5.1 is used.

For a read or write event e on variable v, let $\mathrm{var}(e) = v$, and let $\mathrm{op}(e) = \mathrm{R}(v)$ or $\mathrm{op}(e) = \mathrm{W}(v)$, respectively. An *uninitialized read* of a transaction t is a read event e that is not preceded in t by a write event for $\mathrm{var}(e)$.

A *block* for a transaction t is a tuple $\langle op_1, op_2, fw_1, fw_2, h_1, h_2, h_{12}\rangle$ such that t contains read or write events e_1 and e_2 such that

B1. *(a)* If t contains a write to $\mathrm{var}(e_2)$ that precedes e_2, then e_1 is the last write to $\mathrm{var}(e_2)$ that precedes e_2 in t; otherwise, if t contains a read of $\mathrm{var}(e_2)$ that precedes e_2, then e_1 is the last read of $\mathrm{var}(e_2)$ that precedes e_2 in t. Or, *(b)* if e_2 is the final write to $\mathrm{var}(e_2)$ in t, then e_1 is an uninitialized read of $\mathrm{var}(e_1)$ in t.

B2. for $i \in \{1, 2\}$, $op_i = \mathrm{op}(e_i)$ and $h_i = \mathrm{held}(e_i)$. also, $h_{12} = \mathrm{held}(e_1, e_2)$.

B3. for $i \in \{1, 2\}$, boolean fw_i equals true iff e_i is the final write to $\mathrm{var}(e_i)$

in t.

As a special case, if t contains only one event, then we allow a block in which e_1 is that event, e_2 is a dummy event, and h_2 and h_{12} are empty.

For example, the transaction

(11) $$t : \mathrm{acq}(\ell_1)\ \mathrm{R}(x)\ \mathrm{acq}(\ell_2)\ \mathrm{W}(x)\ \mathrm{R}(x)\ \mathrm{rel}(\ell_2)\ \mathrm{rel}(\ell_1)$$

has two blocks namely,

(12) $$\begin{array}{l}\langle R(x), W(x), \mathrm{false}, \mathrm{true}, \{\ell_1\}, \{\ell_1, \ell_2\}, \{\ell_1\}\rangle \\ \langle W(x), R(x), \mathrm{true}, \mathrm{false}, \{\ell_1, \ell_2\}, \{\ell_1, \ell_2\}, \{\ell_1, \ell_2\}\rangle.\end{array}$$

The first block follows from both B1(a) and B1(b); the second block follows only from B1(a).

To explain condition B1(a), we consider cases based on whether the operations are reads or writes. Let $x = \mathrm{var}(e_2)$. Suppose e_2 is a read. We take e_1 to be the last write to x that precedes e_2 in t, if any; this lets the algorithm determine whether a write from another transaction t' can affect the data-flow in t (in the unserializable pattern UP3), by causing e_2 to read a value written by t' instead of the value written by e_1. If t does not contain a write to x that precedes e_2, then we take e_1 to be the last read of x that precedes e_2 in t, if any; this lets the algorithm determine whether a write from another transaction t' can affect the data-flow in t (in the unserializable pattern UP2), by causing consecutive reads in t to return different values.

Suppose e_2 is a write. We take e_1 to be the last write to x that precedes e_2 in t, if any; this lets the algorithm determine whether a read e' from another transaction t' can be affected (in the unserializable pattern UP1) by being interleaved between e_1 and e_2. If t does not contain a write to x that precedes e_2, then we take e_1 to be the last read of x that precedes e_2 in t, if any; this lets the algorithm determine whether a write e'_W from another transaction t' can be interleaved between those two events in the unserializable pattern UP4. As indicated in the definition of UP4, the pattern $e_1 \cdots e'_W \cdots e_2$ is unserializable only if e'_W is the final write to $\mathrm{var}(e_2)$ in t', because this implies that a trace σ containing this pattern is not equivalent to a serial schedule σ' in which t occurs before t', because the final write to $\mathrm{var}(e_2)$ is from t in σ and from t' in σ'; also, σ is not equivalent to a serial schedule σ' in which t' occurs before t, because e_1 reads the value written by e'_W in σ' and reads an older value in σ.

The intuition behind condition B1(b) is that blocks formed from uninitialized reads and final writes represent end-to-end behavior of the transaction that must be preserved when constructing an equivalent serial schedule. As an example of why B1(b) is necessary, consider the following set of transactions.

(13) $$t : \mathrm{acq}(\ell)\mathrm{R}(x)\mathrm{W}_1(x)\mathrm{rel}(\ell)\mathrm{W}_2(x) \qquad t' : \mathrm{acq}(\ell)\mathrm{W}'(x)\mathrm{rel}(\ell)$$

Primes and subscripts merely distinguish different events. $\{t, t'\}$ is not serializable, because the trace in which $\mathrm{W}'(x)$ occurs between $\mathrm{W}_1(x)$ and $\mathrm{W}_2(x)$ is not equivalent to the serial schedule tt' (because the final write to x is different) or the serial schedule tt' (because the value seen by R(x) is different). Our algorithm (given below) detects this unserializability when it observes that UP4 can occur in the interleavings of a block containing W$'(x)$ (and a dummy event) and the block built (based on condition B1(b)) from the uninitialized read R(x) and the final write W$_2(x)$. If condition B1(b) were omitted, the algorithm would incorrectly report that $\{t, t'\}$ is atomic.

Two blocks b and b' for transactions of different threads are *atomic*, denoted isAtomicBlk(b, b'), if the synchronization indicated by the locksets in the blocks prevent the unserializable patterns described above, *i.e.*, no three out of the four events in the two blocks can form one of those patterns. The selected test for feasibility of interleavings is used to determine this.

Putting these ideas together, we obtain the following algorithm.

```
isAtomic-1Var(T) {
  for all transactions $t$ and $t'$ in $T$ with thread($t$) $\neq$ thread($t'$)
    for each block $b$ for $t$
      for each block $b'$ for $t'$
        if $\neg$isAtomicBlk($b, b'$) then return false
  return true
}
```

Correctness of the algorithm is established by the following theorems.

Theorem 5.2 *Let t and t' be transactions with* thread$(t) \neq$ thread(t') *and that access only one common variable.* $\{t, t'\}$ *is atomic iff, for all blocks b for t and all blocks b' for t',* isAtomicBlk(b, b').

Proof. A proof sketch follows; details appear in [WS03]. For the forward implication ($\Rightarrow$), we prove the contrapositive, *i.e.*, if isAtomicBlk(b, b') is false for some pair of blocks b and b', then t and t' are not atomic. This follows easily from the definition of isAtomicBlk. For the reverse implication ($\Leftarrow$), suppose isAtomicBlk(b, b') holds for all pairs of blocks b and b'. Let S be a non-serial trace for $\{t, t'\}$. If neither transaction performs a write (to the common variable), then S is obviously equivalent to a serial trace. Suppose, without loss of generality, that t performs the final write e_{FW} (to the common variable) in S. If t' does not read the value written by e_{FW}, then all reads and writes in t' precede e_{FW} in S, and we can show that S is equivalent to the serial trace in which t' precedes t; the main point is that there is no read event e'_R in t' that reads the value written by a write e_W of t in S, because if there were, then e_W and e_{FW} would form a block b that can be interleaved in an unserializable way with e'_R, so isAtomicBlk(b, b') would be false for some block b' containing e'_R, a contradiction. If t' reads, the value written by e_{FW}, then we can show that all reads and writes in t' appear after e_{FW} in S (because, if one of those events preceded e_{FW}, an unserializable pattern and hence a non-atomic pair of blocks would exist), and that S is equivalent to the serial trace in which t precedes t'. □

Theorem 5.3 *A set T of transactions that access only one common variable is atomic iff every subset of T with cardinality two is atomic.*

Proof. A proof sketch follows; details appear in [WS03]. For the forward implication ($\Rightarrow$), the proof is straightforward, except for details relating to final writes. We prove the reverse implication ($\Leftarrow$) by induction on the number of transactions in T. Let S be a non-serial trace S for T. Let t be the transaction that performs the final write e_{FW} (to the common variable) in S. S is equivalent to a trace S' in which t and the transactions T' that read e_{FW} appear as a serial suffix S_2 of S', after some possibly non-serial prefix S_1. By the induction hypothesis applied to $T \setminus (\{t\} \cup T')$, S_1 is equivalent to some serial trace S_1'. Thus, S is equivalent to the serial trace $S_1' S_2$, where juxtaposition denotes concatenation. $\Box$

The worst-case running time of the algorithm is $O(EL + B^2)$, where E and B are the total numbers of events and blocks, respectively, in all transactions, and L is the maximum number of locks in a single transaction. The summands correspond to the time needed to construct the blocks and the time consumed by the nested loops in the algorithm, respectively.

The number of blocks is at most linear in the number of events (because condition B1 ensures that each event e_2 is combined with at most one preceding event e_1 to form a block). The number of locks is also $O(E)$. Thus, the worst-case running-time of the algorithm is also $O(E^2)$. The bound $O(EL + B^2)$ is more precise, because the numbers of locks and blocks do not necessarily grow linearly with E. For example, the transaction

(14) $$t : \mathrm{acq}(\ell_1)\ \mathrm{R}(x)\ \mathrm{R}(x)\ \mathrm{R}(x)\ \mathrm{R}(x)\ \mathrm{R}(x)\ \mathrm{R}(x)\ \mathrm{rel}(\ell_1)$$

has only one block: $\langle \mathrm{R}(x), \mathrm{R}(x), \mathrm{false}, \mathrm{false}, \{\ell_1\}, \{\ell_1\}, \{\ell_1\}\rangle$.

The rest of this subsection contains additional remarks about condition B1 and about why the algorithm compares two blocks, instead of a block and an event.

In the case where e_2 is a read of x and t does not contain a write to x that precedes e_2, combining e_2 with all preceding reads of x would also be correct, but it is unnecessary. For example, suppose t has the form $R_1(x)\ R_2(x)\ R_3(x) \cdots$ (the subscripts merely distinguish different events). Our algorithm constructs a block from $R_1(x)$ and $R_2(x)$, and another from $R_2(x)$ and $R_3(x)$, but not from $R_1(x)$ and $R_3(x)$, because if a write from another transaction can intervene between $R_1(x)$ and $R_3(x)$, then it occurs either between $R_1(x)$ and $R_2(x)$ (and hence is detected through analysis of that block) or between $R_2(x)$ and $R_3(x)$ (and hence is detected through analysis of that block).

In the case where e_2 is a read of x and e_1 is the last write to x that precedes e_2 in t, if there are reads e_R of x between e_1 and e_2, forming blocks from e_R and e_2 would cause the algorithm to produce false alarms. To see this, consider the atomic set of two transactions

(15) $t : \mathrm{acq}(\ell)\mathrm{W}(x)\mathrm{R}(x)\mathrm{R}(x)\mathrm{rel}(\ell)$ $\quad$ $t' : \mathrm{acq}(\ell)\mathrm{W}(x)\mathrm{rel}(\ell)\mathrm{acq}(\ell)\mathrm{W}(x)\mathrm{rel}(\ell)$

If condition B1 were weakened to allow t to have a block b formed from t's two reads, then when comparing that block with the block b' formed from the two writes in t', the unserializable event pattern UP1 would be encountered, causing the algorithm to say that t and t' are not atomic. In short, the fact that t's read of x gets its value from a write in the same transaction is essential and would be obscured if B1 were weakened. The same problem arises in algorithms based on comparisons of a block and an event, instead of comparisons of two blocks: pattern UP1 arises when a read from t is compared with block b'.

5.3 Algorithm for Two Transactions

In this section, we consider atomicity of sets containing exactly two transactions. In contrast to Section 5.2, each transaction may access multiple variables. As a result, we need to consider blocks that contain operations on two different variables. We call these *2-blocks* (short for "2-variable blocks").

For a transaction t, the set of *final writes* in t, denoted $FW(t)$, is the set of writes e in t such that there is no subsequent write to var(e) in t. The set of *uninitialized reads* in t, denoted $UR(t)$, is the set of reads e in t such that there is no preceding write to var(e) in t. (As an optimization, it suffices to include in $UR(x)$ only the first uninitialized read, if any, for each variable.)

A *2-block* for a transaction t is a tuple $\langle op_1, op_2, h_1, h_2, h_{12}\rangle$ formed from two events e_1 and e_2, both in $FW(t) \cup UR(t)$, such that

2B1. e_1 precedes e_2 in t, and $\text{var}(e_1) \neq \text{var}(e_2)$.

2B2. same as B2.

Two 2-blocks b and b' are *atomic*, denoted isAtomic2Blk(b, b'), if the synchronization indicated by the locksets in the blocks prevent unserializable patterns. All four events in the pair of blocks must be considered when defining unserializable patterns. This significantly increases the number of cases compared to the unserializable patterns for one variable in Section 5.3, which involve only three events. Many cases fall into a few general categories. Two events *conflict* if they access the same variable and at least one of them is a write. Consider 2-blocks b and b'. If they contain no conflicting events, then they are atomic, i.e., isAtomic2Blk(b, b') holds. If they contain exactly one pair of conflicting events (i.e., the other two events do not conflict with those two events or with each other), then isAtomic2Blk(b, b') holds. Suppose they contain two pairs of conflicting events. [6] There are a few dozen specific cases in which the two blocks are not atomic. These cases are enumerated in [WS03]. Here we list a few illustrative cases. The layout of each case shows an unserializable interleaving. b and b' are not atomic if they contain the indicated events and the locksets in b and b' permit that unserializable interleaving. The

[6] The restriction $\text{var}(e_1) \neq \text{var}(e_2)$ in 2B1 ensures that they do not contain three or four pairwise conflicting events.

cases are sorted by the number of reads.

0 reads	$b:$ W(x) W(y)	$b:$ W(x) W(y)
	$b':$ W(x) W(y)	$b':$ W(y) W(x)
1 read	$b:$ R(x) W(y)	$b:$ R(x) W(y)
	$b':$ W(y) W(x)	$b':$ W(y) W(x)
2 reads	$b:$ R(x) W(y)	$b:$ R(x) R(y)
	$b':$ W(x) R(y)	$b':$ W(y) W(x)

Symmetric variants obtained by swapping b and b' have the same atomicity.

We obtain the following algorithm. Let $\pi_x(t)$ denote the projection of transaction t on variable x, *i.e.*, keep synchronization events and accesses to x, and discard accesses to other variables.

```
isAtomic-2Trans($t, t'$) {
  if thread($t$) = thread($t'$) return true;
  for each variable $x$ accessed in $t$ and $t'$
    if ¬isAtomic-1Var($\{\pi_x(t), \pi_x(t')\}$) then return false;
  for each 2-block $b$ for $t$
    for each 2-block $b'$ for $t'$
      if ¬isAtomic2Blk($b, b'$) then return false;
  return true
}
```

Theorem 5.4 *Let t and t' be transactions with* thread(t) $\neq$ thread(t'). $\{t, t'\}$ *is atomic iff (i) for all blocks b for t and all blocks b' for t',* isAtomicBlk(b, b') *and (ii) for all 2-blocks b for t and all 2-blocks b' for t',* isAtomic2Blk(b, b').

Proof. A proof sketch follows; details appear in [WS03]. For the forward implication ($\Rightarrow$), we prove the contrapositive, which follows easily from the definitions of isAtomicBlk and isAtomic2Blk. For the reverse implication ($\Leftarrow$), suppose all pairs of blocks and 2-blocks are atomic. Let S be a non-serial trace for $\{t, t'\}$. We show that S is equivalent to some serial trace. Suppose there exists a variable x written by t and t'. Suppose, without loss of generality, that t' performs the final write $e'_{W(x)}$ to x in S. Let $e_{W(x)}$ denote a write to x in t. We can prove that S is equivalent to the serial trace S' in which t precedes t'. The most interesting part is the proof that no read $e_{R(y)}$ of t sees a value written by a write $e'_{W(y)}$ in t'; if it did, then $e'_{W(x)}$, $e'_{W(y)}$, $e_{W(x)}$, and $e_{R(y)}$ would form 2-blocks b and b' for which isAtomic2Blk(b, b') is false, a contradiction. Suppose no variable is written by both transactions, and at least one transaction contains a write. Without loss of generality, suppose t contains a write e_W. If some read in t' reads the value written by e_W, then we can show that S is equivalent to the serial schedule in which t precedes t'; otherwise, we can show that S is equivalent to the serial schedule in which t' precedes t. If neither transaction contains a write, then S is trivially serializable. $\square$

5.4 General Algorithm

In the presence of multiple variables, a set T of transactions is not necessarily atomic even if all subsets of T with cardinality two are atomic. This is due to cyclic dependencies. For example, consider the set T containing the three transactions

$$\begin{array}{llllll} & t_1: \mathrm{W}(x) & & & & \mathrm{W}(y) \\ (16) & t_2: & \mathrm{R}(x)\ \mathrm{W}(z) & & & \\ & t_3: & & \mathrm{R}(z)\ \mathrm{R}(y) & & \end{array}$$

All three subsets of T with cardinality two are atomic, but T is not atomic, as witnessed by the unserializable trace indicated by the layout in (16). To see that this trace is unserializable, note that an equivalent serial trace would need to satisfy the following unsatisfiable set of constraints: t_1 precedes t_2 (so t_2's R(x) still sees t_1's W(x)), t_2 precedes t_3 (so t_3's R(z) still sees t_2's W(z)), and t_3 precedes t_1 (so t_3's R(y) still sees the initial value of y).

A simple algorithm to check atomicity of a set T of transactions in the general case is as follows. Let UR-FW(T) denote the set of transactions obtained from T by discarding all events other than synchronization events and uninitialized reads and final writes on shared variables.

```
isAtomicTrans($T$) {
  for all transactions $t$ and $t'$ in $T$ with thread($t$) $\neq$ thread($t'$)
    if $\neg$isAtomic-2Trans($t, t'$) then return false;
  if (every feasible interleaving of transactions in UR-FW($T$) is serializable) then
    return true;
  else return false;
}
```

This algorithm is expensive, because the number of possible interleavings may be large. On the positive side, this algorithm considers only interleavings from uninitialized reads and final writes, and hence may be significantly faster than the naive algorithm that considers all interleavings of all events in T. As an optimization, if the set T of transactions can be partitioned into subsets that access disjoint sets of variables, then each subset of T can be analyzed separately.

6 Example and Future Work

As an example of the kind of software defects that this analysis is designed to find, consider `java.util.Hashtable`. The `Hashtable` API supports the creation of Collection views of the contents of a hashtable. For example, `Hashtable.keySet()` returns a `Set` view of the keys in a hashtable. According to the Java 2 Standard Edition (J2SE) API Specification 1.3.1,

> The Iterators returned by the iterator and listIterator methods of the Collections returned by all of Hashtable's "collection view methods" are fail-fast:

> if the Hashtable is structurally modified at any time after the Iterator is created, in any way except through the Iterator's own remove or add methods, the Iterator will throw a ConcurrentModificationException.

However, in the implementation of `Hashtable` in Sun JDK 1.3.1 and 1.4.0, if a context switch occurs at an inopportune moment, a concurrent update to a `Hashtable h` using `h.remove` may cause the `next()` method of an iterator obtained from `h.keySet().iterator()` to return `null` (even though `null` is not a key in `h`) instead of throwing a `ConcurrentModificationException`. In a sense, violation of the J2SE API Specification 1.3.1 is due to a race condition on the field `h.entry`. But even if the race condition is eliminated by using a lock to protect accesses to `h.entry`, if critical sections (*i.e.*, synchronized blocks) of the wrong granularity are used, the resulting race-free program still violates the specification. This suggests that the underlying problem is not merely the race condition but also a violation of atomicity. Note that the iterator behaves correctly if updates to the hashtable are not interleaved with the call to `next()`. With our run-time analysis, the potential violation of atomicity can be detected even in executions (of the original program or a race-free variant) in which inopportune context switches do not occur and the iterator behaves correctly.

Apparently the Java developers became aware of this problem after the release of JDK 1.3.1. Their solution was to include the following paragraph in the J2SE API Specification 1.4.0:

> Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

We have implemented all of the above run-time analysis algorithms. We plan to use these algorithms to check atomicity of methods in Java programs, by implementing a transformation that instruments programs so they produce traces (logs) showing synchronization operations and accesses to shared variables. We will then experiment with the algorithms to explore the trade-off between cost and precision.

Another topic for future work is to develop more efficient algorithms. We have started to investigate optimizations to the general block-based algorithm. The general idea is to recognize common subproblems and avoid re-computing the results for them, defining and recognizing common sub-problems in this context involves significant complications. It is unclear whether an efficient and precise algorithm exists for the general case, considering that the classic problem of determining serializability of a given trace is NP-complete [Pap79].

Acknowledgment
The authors thank the anonymous reviewers for helpful feedback.

References

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *Proc. First International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS)*, April 2003.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems.* Addison Wesley, 1987.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proc. 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 211–230, November 2002.

[BR01] Chandrasekar Boyapati and Martin C. Rinard. A parameterized type system for race-free Java programs. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 56–69. ACM Press, November 2001.

[FF00] Cormac Flanagan and Stephen Freund. Type-based race detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232. ACM Press, 2000.

[FQ03a] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM Press, 2003.

[FQ03b] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 1–12. ACM Press, 2003.

[Gro03] Dan Grossman. Type-safe multithreading in Cyclone. In *Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 13–25. ACM Press, 2003.

[Hav00] Klaus Havelund. Using runtime analysis to guide model checking of java programs. In *Proc. 7th Int'l. SPIN Workshop on Model Checking of Software*, volume 1885 of *Lecture Notes in Computer Science.* Springer-Verlag, August 2000.

[HR01] Klaus Havelund and Grigore Rosu. Monitoring Java programs with Java PathExplorer. In *Proc. First Workshop on Runtime Verification (RV'01)*, volume 55(2) of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2001.

[HR02] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *Proc. International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2280, pages 342–356, April 2002.

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

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[SBN$^+$97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[WS03] Liqiang Wang and Scott D. Stoller. Run-time analysis for atomicity, May 2003. Available at http://www.cs.sunysb.edu/~liqiang/atomicity.html.