Monitoring Atomicity in Concurrent Programs

Azadeh Farzan[1] and P. Madhusudan[2]

[1] Carnegie Mellon University (afarzan@cs.cmu.edu)
[2] Univ. of Illinois at Urbana-Champaign (madhu@cs.uiuc.edu)

Abstract. We study the problem of monitoring concurrent program runs for atomicity violations. Unearthing fundamental results behind scheduling algorithms in database control, we build space-efficient monitoring algorithms for checking atomicity that use space polynomial in the number of active threads and entities, and independent of the length of the run monitored. Second, by interpreting the monitoring algorithm as a *finite automaton*, we solve the model checking problem for atomicity of finite-state concurrent models. This establishes (for the first time) that model checking finite-state concurrent models for atomicity is decidable, and remedies incorrect proofs published in the literature. Finally, we exhibit experimental evidence that our atomicity monitoring algorithm gives substantial time and space benefits on benchmark applications.

1 Introduction

Correct concurrent programs are way harder to write than sequential ones. Sequential bug-free programs are already hard to write, maintain, and test, though the tremendous effort over the last 20 years in finding errors in programs has yielded certain tractable approaches and tools to assure correctness. The advent of multi-core technologies and the increasing use of threads and communicating modules in software design has brought all concurrency issues to the forefront. Consequently, one of the most important problems in software analysis is to understand concurrency idioms used in practice, and leverage the understanding to build testing and verification tools.

While programming for a multicore (shared-memory) architecture to exploit concurrency, a useful mechanism to have is the ability to parallelize tasks such that there is controlled interaction amongst them. For instance, the proposal of transactional memory (and software transactional memory [23]) introduces such an atomicity construct in a programming language. Programmers writing in current programming languages (such as Java or C with Pthreads) implicitly need such a construct, but since it is not available, implement their own concurrency control mechanism (say using locks) in order to mutually exclude their threads from accessing shared data. A large number of errors in these concurrent programs are due to mismanaged atomicity. For instance, a recent study on bug characteristics in real-world concurrent programs revealed that more than 68% of concurrency bugs were due to atomicity violations (where blocks of code were intended to be atomic but the mechanisms did not ensure atomicity) [16].

The above intuition motivates a remarkable *generic specification* (a specification common across applications) for concurrent programs called *atomicity*.

Consider a concurrent program where certain blocks of code are annotated as transaction blocks, capturing the *intention* of the programmer that they be atomically executed. We declare an interleaved run to be atomic if it is semantically equivalent to a *serial* run where the transaction blocks are scheduled one after another, without any interleaving. The idea then is that non-atomic runs violate programmer intentions and hence are likely to be unintended interactions that may be concurrency errors. This notion of atomicity stems from the concept of *serializability* studied in *database concurrency control*, and the idea of using the notion of atomicity as a generic specification for concurrent programs (running on non-TM platforms) was first proposed by Flanagan and Qadeer in 2003 [9].

The most well-studied, accepted and tractable notion of serializability is *conflict serializability* [20, 4]. Intuitively, we declare events to be *dependent* if they cannot be commuted— so, two accesses to the same variable are dependent if one of them is a write. Two runs r and r' are equivalent if we can obtain one from the other by commuting independent events in the run. A run is conflict-serializable if it has an equivalent serial run. We study only conflict-serializability in this paper, and will henceforth refer to it as simply serializability. The terms serializability and atomicity are synonymous in this paper.

Notice that our methodology of finding errors in programs is parameterized with annotations of blocks of code intended to be atomic. While we can choose natural syntactic blocks of code (such as methods in a class) to be blocks intended to be atomic (as many papers in the literature have done), it is also possible to learn the intended atomic blocks from positive test runs [17] (see also [27]). Here, we are interested in building algorithms for identifying non-atomic runs, and hence we will assume that the annotations of transactional blocks as given.

The objective of this paper is to (a) study the algorithmics of monitoring atomicity of individual runs of concurrent programs, and (b) leverage the monitoring algorithm to solve the model checking problem for checking atomicity in concurrent Boolean programs.

A simple monitoring algorithm for serializability works by maintaining a *conflict graph*, which is a graph depicting the precedence order imposed by the run on the transactions (blocks of code). A run is serializable if, and only if, this graph remains acyclic. A tempting idea to minimize the conflict graph while monitoring a run is to remove *completed transactions* from the conflict-graph, replacing it with *transitive edges* that summarize its effect, with the intuition being that the completed transactions cannot play any role in the future. However, this intuition is *wrong*. In a paper by Alur, McMillan and Peled [2], automata-based algorithms were designed for checking serializability that overlooked this subtlety and deleted transactions, resulting in an *erroneous algorithm* (confirmed by one of its authors [1]).

Unearthing techniques from designs of database schedulers, we obtain simple space-efficient algorithms for monitoring runs for serializability. The main idea is not to remove completed transactions, but rather summarize their effects by throwing in transitive edges and absorbing their event content into active transactions. Such algorithms are present in the database literature (see [5]),

and we have adapted these to the multithreaded software realm (especially to the setting of threads executing multiple transactions) to provide space-efficient monitoring. We refer the interested reader to two textbooks on database theory [20, 4], a paper on the combinatorics of removing completed transactions and related deletion policies on the conflict-graph [12], and a volume on concurrency control by Casanova [5], from which our intuitions have been gained.

The monitoring algorithm we obtain is a streaming algorithm that reads runs of a program, and after reading a run r uses space $O(k^2 + k.n)$, where k is the number of active threads after r and n is the number of entities accessed by r. Furthermore, the time taken to update the information on reading an event is only linear in this graph. Note that there is no dependence on the *number* of events or transactions executed in r, and hence our algorithm scales well when working on long executions of programs.

The monitoring algorithm, surprisingly, paves the way to decision procedures to *model check* programs for atomicity violations. When there are a finite number of threads and entities, our monitoring algorithm uses a bounded amount of space, and hence can be seen as a *deterministic finite automaton*. Using this, we prove that concurrent Boolean programs without recursion (where each thread runs a program with a regular control structure and where all variables are interpreted over finite domains), the model checking problem for atomicity is decidable and is . PSPACE-complete. As far as we know, this is the first time that the decidability of model checking atomicity for finite-state programs has been established. Note that a similar claim appears in [2], but is incorrect due to the grave error we mentioned.

Turning to the experiments, we have implemented both the conflict-graph based monitoring algorithm and the new monitoring algorithm based on summarized conflict graphs. We evaluate these on a suite of benchmarks, and illustrate the significant space (and hence time) gains our algorithm provides in monitoring long runs of realistic concurrent benchmark programs.

Related Work: Atomicity is a new notion of correctness; the more classical notion is *race checking*: a race is a pair of accesses to the same variable by different threads, where one of the accesses is a write [19, 22, 6]. Data races also signal improper synchronization in code, and are routinely used to find errors in concurrent programs (see [22, 6] for testing and runtime checking for data races and [18] for static data race analysis). It has been suggested [10, 8, 26, 25] that *atomicity notions based on serializability* are more appropriate and yield fewer false positives; some practical tools built for serializability demonstrate this [27].

Most work in software verification for atomicity errors are based on approximations of the concept, including Lipton transactions (a *sufficient* but not necessary condition) that ensures serializability [15]. Type systems [10] and model checkers [13] for atomicity based on Lipton transactions have been developed. The work in [8] reports ways of exploring a run and a possible serialization of it simultaneously and checks whether they result in the same effect. In [7], we had proposed a slightly different notion of atomicity called *causal atomicity* which can be checked using partial-order methods. The work in [17] defines *access in-*

terleaving invariants that are certain patterns of access interactions on variables, learns the intended specifications using tests, and monitors runs to find errors.

The closest work to ours is a series of papers by Wang and Stoller on runtime verification of atomicity [25, 26]. In these papers, the authors consider a different and harder problem than what we tackle: they consider a run, project the run onto each thread, and ask whether they can be recombined in some way to produce a non-serializable run This is significantly harder: first, recombinations of runs may not be feasible in the original program in general, and though the authors handle locks accurately, the runs may still be infeasible (say due to data checks) and hence raise false alarms. Also, even abstracting the program to a set of reads and writes as they do, the problem of checking if a non-serializable run exists can be shown to be NP-hard. The authors provide approximate algorithms, that are neither sound nor complete, for the settings where locks are nested and transactions have no potential for deadlocks. The experimental results are reported for a small number of threads (3 in most cases). Our problem however is to simply check if the current run is itself non-serializable for which we show a scalable algorithm and where we have tested the benchmarks for a large number of threads (going up to 50; see Section 5 for more details).

A variant of dynamic two-phase locking algorithm [20] for detection of serializability violations is used in the atomicity tool developed in [27]. As discussed in [20], the set of runs that are detected as atomic by a two-phase locking algorithm are a strict subset of the set of conflict serializable runs.

2 Preliminaries

Modeling Runs of Concurrent Programs: We consider programs that run threads concurrently, with accesses to local and global data. We also assume that blocks of program code are marked as *transactions*, with each thread running a sequence of transactions on any run. We will check runs of programs for atomicity violations with respect to these blocks. We first define a general notion of a run of a concurrent program, where we assume the global accesses, the thread creations and termination, and the beginning and ending of transactions are observable.

Let us assume an infinite but countable set of *thread identifiers* $\mathcal{T} = \{T_1, T_2, \ldots\}$. Let us also assume a countable set of (global) *entity names* (or just entities) $\mathcal{X} = \{x_1, x_2, \ldots, \}$. The set of actions $\mathcal{A}$ over X, is defined as: $\mathcal{A} = \{rd(x), wr(x) \mid x \in \mathcal{X}\}$. The alphabet of events of a thread $T \in \mathcal{T}$ is

$$\Sigma_T = \{T{:}a \mid a \in \mathcal{A}\} \cup \{T{:}\triangleright, T{:}\triangleleft\} \cup \{Beg_T, End_T\}.$$

The events $T{:}rd(x)$ and $T{:}wr(x)$ correspond to thread T reading and writing to entity x, respectively, and $T{:}\triangleright$ and $T{:}\triangleleft$ correspond to transaction boundaries that begin and end blocks of code in thread T, while Beg_T and End_T denote the creation and termination of the thread T itself. Let $\Sigma = \bigcup_{T \in \mathcal{T}} \Sigma_T$ denote the set of all events.

Note that the above can model dynamic memory allocation as well, provided we observe the memory allocation/release actions. The only difference is that the set of actions $\mathcal{A}$ changes during the monitoring. We can assign a fresh name

to every new piece of memory allocated (based on the desired granularity), and maintain aliasing information to observe accesses to the location.

For any alphabet A, $w \in A^*$, let $w[i]$ denote the i'th element of w, and $w[i,j]$ denote the substring from position i to position j in w. For $w \in A^*$ and $B \subseteq A$, let $w|_B$ denote the word w projected to the letters in B. For a word $\sigma \subseteq \Sigma^*$, let $\sigma|_T$ be a an abbreviation of $\sigma|_{\Sigma_T}$, which includes only actions of thread T.

The runs of concurrent programs, which we call *schedules*, are executions of the program where actions of threads are interleaved.

Definition 1. *A schedule is a word $\sigma \in \Sigma^*$ such that for each $T \in \mathcal{T}$, $\sigma|_T$ is a prefix of the word $Beg_T \cdot \left[(T{:}\triangleright) \cdot \{T{:}a \mid a \in \mathcal{A}\}^* \cdot (T{:}\triangleleft)\right]^* \cdot End_T$.*

In other words, the actions of thread T start with Beg_T and end with End_T, and the actions within are divided into a sequence of transactions, where each transaction begins with $T{:}\triangleright$, is followed by a set of reads and writes, and ends with $T{:}\triangleleft$. Let *Sched* denote the set of all schedules.

Notice that schedules do not observe synchronization mechanisms such as mutual exclusion using locks, semaphores, etc. as serializability of *one schedule* is independent of the synchronization mechanism.

A transaction tr of thread T is a word of the form $T{:}\triangleright\ w\ T{:}\triangleleft$, where $w \in \{T{:}a \mid a \in A\}^*$. Let $Tran_T$ denote the set of all transactions of thread T, and let *Tran* denote the set of all transactions.

When we refer to two particular events $\sigma[i]$ and $\sigma[j]$ in σ, we say they *belong* to the same transaction if they belong to the same transaction block: i.e. if there is some T such that $\sigma[i] = T{:}a$, $\sigma[j] = T{:}a'$, where $a, a' \in A$, and there is no i', $i < i' < j$ such that $\sigma[i'] = T{:}\triangleleft$. We refer to the transaction blocks freely and associate (arbitrary) names to them, using notations such as tr, tr_1, tr', etc.

Defining atomicity: We define atomicity through the notion of *conflict serializability*. The *dependency* relation D is a symmetric relation defined over the events in Σ, and captures the dependency between (a) two events accessing the same entity, one of them being a write, and (b) any two events of the same thread, i.e.,

$$\begin{aligned} D = \{(T_1{:}a_1, T_2{:}a_2) \mid (T_1 = T_2 \wedge a_1, a_2 \in A \cup \{\triangleright, \triangleleft\}) \vee \\ [\exists x \in \mathcal{X} \text{ such that } (a_1 = rd(x) \wedge a_2 = wr(x)) \vee \\ (a_1 = wr(x) \wedge a_2 = rd(x)) \vee (a_1 = wr(x) \wedge a_2 = wr(x))]\} \end{aligned}$$

Definition 2 (Equivalence of schedules). *The equivalence of schedules is defined as the* smallest *equivalence relation $\sim \subseteq Sched \times Sched$ such that the following condition holds:*

$$\text{if } \sigma = \rho e e' \rho', \sigma' = \rho e' e \rho' \in Sched \text{ with } (e, e') \notin D, \text{ then } \sigma \sim \sigma'.$$

It is easy to see that the above notion is a well-defined equivalence relation. Intuitively, two schedules are considered equivalent if we can derive one schedule from the other by iteratively swapping consecutive independent actions in

the schedule. It is also clear (given that two actions accessing an entity are independent only if both are reads), that equivalent schedules produce the same valuation of all entities. Formally, assuming each thread has its view limited to the entities it has read and written, we can show that no matter what the domain of the entries are, and what functions the individual threads may compute, and no matter how many *local* variables a thread may have, the final values of both local variables and global entities remains unchanged when executing two equivalent schedules.

We call a schedule σ *serial* if all the transactions in it occur atomically: formally, for every i, if $\sigma[i] = T{:}a$ where $T \in \mathcal{T}$ and $a \in A$, then there is some $j < i$ such that $T[i] = T{:}\triangleright$ and every $j < j' < i$ is such that $\sigma[j'] \in \Sigma_T$. In other words, the schedule is made up of a sequence of complete transactions from different threads, interleaved at boundaries only (the final transactions of a thread may be incomplete, but even then the actions in each incomplete transaction must occur together).

Definition 3. *A schedule is* serializable *if it has an equivalent serial schedule. That is, σ is a serializable schedule if there a* serial *schedule σ' such that $\sigma \sim \sigma'$.*

The conflict-graph characterization: A simple characterization of atomic (or conflict-serializable) schedules uses the notion of a conflict-graph, and is a classic characterization from the database literature (this notion is so common that many papers in database theory *define* conflict serializability using it).

If a transaction tr of thread T reads x and is followed by a transaction tr' of thread T' that writes x, then we must schedule the (entire) transaction T before the (entire) transaction T'. The conflict graph [11, 20, 4] is a graph that captures these constraints, and is made up of transactions as nodes, and edges capturing ordering constraints imposed by a schedule. A schedule is serializable iff its conflict graph represents a partial order (i.e. is acyclic).

Formally, for any schedule σ, let us give names to transactions in σ, say $tr_1, \ldots, tr_n$. The *conflict-graph* of σ is the graph $CG(\sigma) = (V, E, S)$ where $V = \{tr_1, \ldots, tr_n\}$, $S : V \longrightarrow 2^{\Sigma}$ is a labeling of vertices such that $S(tr_i)$ is precisely the set of events that have been scheduled in tr_i, and E contains an edge from tr to tr' iff there is some event e in transaction tr and some event e' in transaction tr' such that (1) the e-event occurs before e' in σ, and (2) eDe'.

Note that transactions of the same thread are always ordered in the order they occur (since all actions of a thread are dependent on each other to preserve sequential consistency).

Lemma 1. *A schedule σ is atomic iff the conflict graph of σ is acyclic.* □

The above lemma essentially follows from [11] (also [4, 20]). The classic model is however a database model where transactions are independent; we require an extension of this lemma to our model where there are additional constraints imposed by the fact that some transactions are executed by the same thread. The above characterization actually happens to hold when the underlying alphabet

has *any arbitrary* dependence relation, and hence holds in our threaded model since we have ensured that any two events of a thread are dependent.

If the conflict graph is acyclic, then it can be viewed as a partial order, and it is clear that *any* linearization of the partial order will define a serial schedule equivalent to σ. If the conflict-graph has a cycle, then it is easy to show that the cyclic dependency rules out the existence of any equivalent serial schedule.

The above characterization yields a simple algorithm for serializability that simply constructs the conflict-graph and checks for cycles. The algorithm can process the schedule event by event, updating the graph, and finally call a cycle-detection routine. Hence [20, 4]:

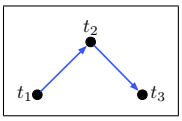
Proposition 1. *The problem of checking whether a singe schedule σ is atomic is decidable in polynomial time.* □

3 Monitoring Atomicity

The goal of this section is to build a space-efficient monitoring algorithm for checking serializability violations. Our algorithm, after reading a run r, will take a space at most polynomial in the (maximum) number of *active threads* (at every moment) in r and the number of entities accessed in r; most importantly, this will have no dependence on the length of r itself.

Note that there is a simple algorithm that monitors serializability violations by keeping track of the conflict graph and checking it for cycles. However, since the conflict graph contains one node for every transaction that has ever happened in the system, it can grow arbitrarily large, and does not result in the monitoring algorithm we seek. We want to keep a reduced conflict graph when monitoring.

Let us look at an example to see how to reduce the conflict graph. Consider the following non-serializable schedule:



$$T_1{:}\triangleright\ T_1{:}rd(x)\ T_2{:}\triangleright T_2{:}wr(x)\ T_2{:}rd(z)$$
$$T_3{:}\triangleright\ T_3{:}wr(z)\ T_2{:}\triangleleft\ T_1{:}wr(x)$$

The figure on the right demonstrates the conflict graph for the above schedule without the last event $T_1{:}wr(x)$ (t_1, t_2, and t_3 are transactions of threads T_1, T_2, and T_3).

Now, in this graph, since the transaction t_2 of T_2 has finished, it is very tempting to remove the node t_2, and summarize its effect by replacing it by an edge from t_1 to t_3. However, this is a *serious error:* for example, if we deleted t_2, then the next event $T_1{:}wr(x)$ *does not cause* a cycle. The reason is that though t_2 is a completed transaction, it may still participate in later cycles as *outgoing* edges from t_2 can always be introduced even after t_2 has completed.

The work by Alur, McMillan and Peled [2] considers precise algorithms for monitoring serializability violations (using automata that keep track of reduced conflict-graphs on the schedule it has read). Their paper has the grave error mentioned above, and the algorithms establishing upper bounds of checking serializability in the paper are *wrong*.

In fact, the problem of when completed transactions can be deleted is a well studied problem in database concurrency control, and there have been several approaches to finding safe deletion policies [4, 5, 12]. Next, we present a way to *summarize* the essential information of the conflict-graph using just a graph of nodes formed by active threads[3].

Summarized Conflict Graph for Serializability

The following notion of summarized conflict graphs is adapted from a conflict-serializable scheduling algorithm in [5]. Intuitively, we keep the conflict graph restricted to edges between the active transactions only (paths between active transactions summarized as edges), and also maintain for each active transaction/thread T a set C which denotes *the set of events that occurred in transactions that must be scheduled later than this transaction.* Moreover, when keeping track of events of completed threads, we erase the thread id from its description.

Recall that $\mathcal{A}$ is the set of actions of reads and writes to global entities, and Σ includes the set $\{T{:}a \mid T \in \mathcal{T}, a \in \mathcal{A}\}$ as well as begin and end events for transactions and threads. In the summarized graph, each node corresponding to an active thread T will be associated with two sets, S and C, where $S \subseteq \Sigma_T$ is the set of *scheduled events* of the current active transaction of thread T (as in the conflict graph), and $C \subseteq \Sigma \cup \mathcal{A}$ is the set of *conflicting events*, events that occurred in completed transactions that must be scheduled later than the current transaction of T.

Definition 4. *Let σ be a schedule and let $CG(\sigma)$ be its conflict graph. The* summarized conflict graph *of a schedule σ is a tuple $SCG_\sigma = (V, E, S, C)$, where (V, E) is a graph and S and C are two vertex-labeling functions $S : V \longrightarrow 2^{\Sigma}$, $C : V \longrightarrow (2^{\Sigma \cup \mathcal{A}})$, where*

- *V contains a node v_i for each active thread T_i;*
- *E contains an edge from v to v' (respectively associated to transactions tr and tr') if and only if there exists a path from the node corresponding to tr to that corresponding to tr' in $CG(\sigma)$ which does not contain any nodes corresponding to active transactions;*
- *For any $v_i \in V$, if v_i corresponds to active transaction tr, then $S(v_i)$ consists of precisely the label of tr in the conflict-graph of σ, and $C(v_i)$ contains:*
  - *the set of events $T{:}a \in \Sigma$ such that T is an active thread and there is some* completed *transaction tr', reachable from tr in $CG(\sigma)$, whose label contains $T{:}a$, and*
  - *the set of actions $a \in \mathcal{A}$ such that there is some* completed *transaction tr', reachable from tr in $CG(\sigma)$, whose label contains $T{:}a$, and where T is a thread that has already ended in σ.*

The above is a static definition of the summarized conflict graph, and it is easy to see that cycles in the conflict graph manifest themselves in the summarized conflict graph:

[3] An active thread T is a thread for where the $begin_T$ action has appeared in the schedule, and end_T has not appeared yet .

Lemma 2. *There is a cycle in the conflict graph of σ iff there is a cycle in the summarized cycle graph of σ' for some σ' that is a prefix of σ.* □

Notice that when a cycle gets formed in the summarized conflict graph, it may get removed later (for example if all the threads that form the transactions of the cycle end) However, since whenever a node is removed, we combine incoming edges and outgoing edges from this node with a transitive edge, it follows that unless all nodes in the cycle are removed, the cycle is preserved. When the cycle is finally removed, it will be in the form of a self-loop on a transaction node that is being deleted. Hence, when monitoring, it is sufficient to check for self-loops.

Maintaining the summarized conflict graph: Let us now turn to an algorithm for maintaining the summarized conflict graph. The following set of rules show how the summarized conflict graph can be constructed dynamically as the schedule σ progresses. The dynamic algorithm updates the graph based on these rules until a self-loop is created, and at which point reports a serializability violation. The algorithm maintains a set AT of the currently active threads.

- **(Rule 1):** If the next event in σ is $T_i{:}\triangleright$, then create a new node v_i and set $S(v_i) = \emptyset$ and $C(v_i) = \emptyset$.
- **(Rule 2):** If the next event in σ is $T_i{:}\triangleleft$, then remove v_i by connecting all its (immediate) predecessors to all its (immediate) successors. Also for every (immediate) predecessor v_k of v_i, set $C(v_k) = C(v_k) \cup S(v_i) \cup C(v_i)$.
- **(Rule 3):** If the next event in σ is $T_i{:}a$, then set $S(v_i) := S(v_i) \cup \{T_i{:}a\}$. For all $v_k \neq v_i$, if there is an event $T_j{:}b \in S(v_k) \cup C(v_k)$ such that $(T_j{:}b, T_i{:}a) \in \mathcal{D}_\Sigma$, add an edge (v_k, v_i) to E. Also, for any action $b \in C(v_k)$ such that $(T_k{:}b, T_i{:}a) \in \mathcal{D}_\Sigma$, add an edge (v_k, v_i) to E.
- **(Rule 4):** If the next event in σ is Beg_{T_i} , then set $AT = AT \cup \{T_i\}$.
- **(Rule 5):** If the next event in σ is End_{T_i}, then set $AT = AT \setminus \{T_i\}$, and replace every $T_i{:}a$ in every conflict s et C by a.

The summarized conflict graph given here is derived from a similar one presented in [4, 5], but adapted to handle threads. Furthermore, for terminated threads, we have adapted the algorithm to remove the thread id information, thereby bounding the information kept at each node to the product of the number of *active* threads and entities accessed only. In the case of dynamic memory allocation, when a location is freed, it can be removed from all the label sets.

The following theorem captures the correctness of the algorithm in maintaining the summarized conflict graph, and hence, by Lemma 2 and Lemma 1, is a streaming algorithm that detects serializability violations.

Theorem 1. *The algorithm presented streams events of a schedule σ, and maintains the summarized conflict graph. Hence, for schedules in which all started threads end, the algorithm detects a self-loop on some node at some point in time iff the schedule is non-serializable.* □

Note that if one is interested in checking a schedule that does not conform to the above, it is always easy to add transaction end and thread end actions to the end of the schedule to make it so.

Complexity of the algorithm: Our monitoring algorithm using the summarized conflict graph simply reads events of a schedule, maintains the summarized conflict graph, and checks if at any point a self-loop is introduced. Its space complexity is as follows:

Proposition 2. *For any schedule σ, the number of nodes of the summarized conflict graph is bounded by k and the combined sizes of the sets associated with the nodes is bounded by $k.n$, where k is the maximum number of active threads during σ and n is the number of entities accessed by it. The size of the summarized conflict graph is therefore of $O(k^2 + k \cdot n)$.* □

While scanning the schedule σ, the algorithm spends time $O(k \cdot \log n)$ when the next event is a read or a write action by some thread. If the next event is an end of a transaction, then the monitoring algorithm spends time $O(n \cdot k)$. The updates for other action are performed in constant time.

4 Model Checking Atomicity for Boolean Programs

In this section, we present the second main result of the paper: a solution for the problem of checking atomicity of concurrent Boolean programs and establishing its complexity to be precisely PSPACE-complete. The result we prove here was claimed in [2], but as we mentioned, the proof there was wrong.

We consider succinct representations of programs with Boolean variables with logical encodings of initial positions and the transition relation (very much akin to how systems are described in model checking based on Boolean decision diagrams such as NuSMV).

Let us fix a finite set of threads $\mathcal{T} = \{T_1, \ldots, T_k\}$ and a finite set of entities $\mathcal{X} = \{x_1, \ldots, x_n\}$. Recall the set of actions associated with these threads and entities:

$$\Sigma_T = \{T{:}a \mid a \in \mathcal{A}\} \cup \{T{:}\triangleright, T{:}\triangleleft\} \cup \{Beg_T, End_T\}$$

and $\Sigma = \bigcup_{T \in \mathcal{T}} \Sigma_T$. Note that Σ is a finite set.

A Boolean program over threads $\mathcal{T}$ and $\mathcal{X}$ is defined over a finite set of Boolean variables $\mathcal{V}$, where the initial set of states is described using a Boolean formula $Init(V)$, and for each $e \in \Sigma$, we have a Boolean formula $Trans_e(\mathcal{V}, \mathcal{V}')$, where $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$, which describes the transitions on the event e. The size of the program is defined as $|\mathcal{V}| + |Init(\mathcal{V})| + \sum_{e \in \Sigma} |Trans_e(\mathcal{V}, \mathcal{V}')|$. The semantics of the program is the natural one.

Imperative concurrent programs with common synchronization constructs can easily be described as a Boolean program provided there are only a finite number of threads and entities, and the data manipulated by the program has been abstracted into a finite domain. This model is particularly interesting in an *abstract-interpretation* framework where data domains are abstracted, say using predicate abstraction [3].

The key to model checking for atomicity violations of Boolean programs is to realize that the monitoring algorithm presented in this paper maintains a *bounded* graph to check atomicity, when the number of threads and entities are

bounded. We can turn this monitoring algorithm into a deterministic automaton that checks for atomicity violations, and using the automata-theoretic method, reduce model checking to a decidable emptiness problem on automata. We build an automaton Ser that accepts the set of all serializable runs (of k threads and n entities). It is easy to see that size of Ser is exponential in n and k. By building an automaton B that accepts all the runs of the program (which will also be exponential in the size of the program), we can model check for atomicity by checking if $L(B) \subseteq L(Ser)$, which can be achieved in PSPACE by generating these automata on-the-fly.

Theorem 2. *The problem of checking if a Boolean program is serializable is* PSPACE*-complete.* □

5 Experimental Evaluation

We implemented two algorithms to monitor serializability of program runs: one was the classic algorithm based on conflict graphs, and the other of our new algorithm based on summarized conflict graphs. Comparing with the conflict graph algorithm is useful since existing methods [25] use structures that are similar to the conflict graph.

We evaluated the algorithms on a benchmark suite of 5 programs. These benchmarks include `sor` (successive over-relaxation), `lufact` (LU factorization), and `raytracer` from the Java Grande multithreaded benchmarks [14], and `elevator` and `tsp` from [21]. `sor` and `lufact` are (data-intensive) scientific computation programs which perform numerical computation on matrices, `elevator` simulates multiple lifts in a building, `tsp` solves the traveling sales man problem for a given input map, and `raytracer` renders a frame of an arrangement of spheres from a given view point.

We extracted runs by manually instrumenting programs to output the accesses to entities while executing. We have a simple automatic *escape analysis* unit that excludes from the run all accesses to thread-local entities. We then run the monitoring algorithms on these output files offline. We use the `glib` library to efficiently implement set and graph operations. In the case of summarized conflict graphs, we check for cycles by checking for self loops in the graph, and for conflict graphs, we check the graph for cycles *once* at the end.

Table 1 presents the results of our evaluation. We ran each benchmark with different input parameters such as number of threads, and input files. For each program, we report in the table the number of lines of code (LOC) (appears below the program names), number of threads used in the run and number of *truly shared* entities between threads, the length of the run (number of events). The table presents the results of running the two monitoring algorithms, conflict graph (CG) and summarized conflict graph (SCG) on these runs, and we report the size of each graph (in number of nodes and edges), and the time (in seconds) consumed to monitor each run (an entry of 0 means the time was less than 0.01 seconds). Note that the times are for processing the run only, and not

Application (LOC)	Spec	Threads; Entities	Length of the Run	CG (n, e)	CG (time)	SCG (n, e)	SCG (time)	ser-viol/ bug
`sor` (470)	100×100	3; 400	97M	600, 80K	118s	3, 0	0.5s	no/–
	100×100	10; 1800	97M	2K, 300K	8872s	6, 0	3.5s	no/–
	100×100	50; 10000	101M	—	> 8h	17, 0	30.4s	no/–
`lufact` (1234)	100×100	3; 10K	17M	894, 168K	1824s	3, 2	37s	no/–
	100×100	10; 10K	18M	3K, 617K	3940s	10, 9	55s	no/–
	100×100	50; 10K	22M	15K, 3M	11640s	50, 49	84s	no/–
`elevator` (566)	data	2; 32	7K	137, 6K	0.06s	3, 2	0	yes/no
	data2	4; 32	220K	416, 26K	0.97s	5, 2	0.01s	yes/no
	data3	4; 200	571K	258, 10K	6.3s	5, 3	0.1s	yes/no
`tsp` (794)	map4	3; 15	80	6, 5	0	1, 0	0	no/–
	map14	5; 40	1.4M	18, 109	2.2s	4, 3	1.0s	yes/no
`raytracer` (1537)	150×150	10; 1	66	10, 90	0.02s	10,9	0.02s	yes/yes
	200×200	10; 1	66	10, 79	0.02s	10,9	0.02s	yes/yes

Table 1. Monitoring Results (K=1000; M=1000000)

generating the run (as that part is common to both algorithms). Finally, we report whether we found a serializability violation (yes/no), and if yes, whether it pointed to a real bug in the program (yes/no), the latter determined manually. All experiments were performed on Linux PC with two 4GHz processors and 4GB of memory.

Our results clearly illustrate the tremendous impact of using our summarized conflict-graph algorithm, giving orders of magnitude speedup when compared to the classic conflict-graph algorithm. For example, for `sor` with 50 threads, the CG-algorithm did not finish even after 8 hours while SCG finishes in 30s. This is primarily due to space savings; for example, `lufact` with 50 threads gives a conflict graph of $15K$ nodes and $3M$ edges, while the SCG graph never uses more than 50 vertices (one for each thread) and 49 edges.

Note that the algorithm reported in [25] solves a harder problem, as it tries to find serializability violations in the current run as well as all other runs that can be inferred from this run. In fact, their technique does not scale well as they use a graph similar to the conflict graph; they reported to us that they get timed-out after 25 minutes while checking the `sor` benchmark for 50 threads while we check the run in 30 seconds [24].

All the runs of `sor` and `lufact` that we monitored were serializable. `tsp` and `elevator` generated non-serializable runs. But a closer investigation of the source of non-serializability of these runs shows that they do not correspond to real errors in the program. They both refer to interesting cases of non-trivial thread interactions which are intended by the programmer, and we believe a programmer would find such reports of interactions useful. The non-serializable runs of `raytracer`, however, are related to a real bug in that program. The benchmarks, the monitoring programs, and the precise runs monitored are available at `http://www.cs.uiuc.edu/~madhu/cav08/`.

Acknowledgement. We thank Liqiang Wang and Scott Stoller for useful discussions and clarifications both theoretical and experimental.

References

1. R. Alur. Personal communication.
2. R. Alur, K. L. McMillan, and D .Peled. Model-checking of correctness conditions for concurrent objects. *Information and Computation*, 160(1-2):167–188, 2000.
3. T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.
4. P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.
5. M.A. Casanova. *Concurrency Control Problem for Database Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1981.
6. D. R. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.
7. A. Farzan and P. Madhusudan. Causal atomicity. In *CAV*, LNCS 4144, pages 315–328, 2006.
8. C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multi-threaded programs. In *POPL '04*, pages 256–267, 2004.
9. C. Flanagan and S. Qadeer. Types for atomicity. In *TLDI*, pages 1–12, 2003.
10. C. Flanagan and S .Qadeer. A type and effect system for atomicity. In *ACM SIGPPLAN PLDI '03*, pages 338–349, 2003. ACM Press.
11. M. P. Fle and G. Roucairol. On serializability of iterated transactions. In *PODC '82: Proc. of ACM SIGACT-SIGOPS PODC*, pages 194–200, 1982. ACM Press.
12. T. Hadzilacos and M. Yannakakis. Deleting completed transactions. In *ACM SIGACT-SIGMOD PODS*, pages 43–46, 1986.
13. J. Hatcliff, Robby, and M. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model checking. In *VMCAI*, pages 175–190, 2004.
14. Java Grand Benchmark Suite: `http://http://www.javagrande.org/`
15. R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
16. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.
17. S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access-interleaving invariants. In *ASPLOS*, 2006.
18. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In M. I. Schwartzbach and T. Ball, editors, *PLDI*, pages 308–319. ACM, 2006.
19. R. H. B. Netzer and B. P. Miller. What are race conditions? some issues and formalizations. *LOPLAS*, 1(1):74–88, 1992.
20. C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, Inc., USA, 1986.
21. von Praun, C., Gross, T.R.: Object race detection. SIGPLAN Not. **36**(11) (2001) 70–82
22. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *SOSP*, 1997.
23. N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.
24. L. Wang and S. D. Stoller. Personal communication.
25. L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP*, pages 137–146, 2006.
26. L. Wang and S D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, 2006.
27. M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Not.*, 40(6):1–14, 2005.