Predicting Atomicity Violations in Concurrent Programs via Planning

Niloofar Razavi Azadeh Farzan Sheila A. McIlraith
Department of Computer Science,
University of Toronto,
Toronto, Ontario, Canada
{razavi, azadeh, sheila}@cs.toronto.edu

Abstract

Testing concurrent programs is more difficult than testing sequential programs due to the *interleaving explosion problem*: even for a fixed program input, there are numerous different runs that need to be tested to account for scheduler behaviour. Testing all such interleavings is not practical. Consequently, most effective testing algorithms attempt to generate runs that are likely to manifest bugs. *Atomicity violating* runs have been proposed as good candidates since a large fraction of the existing concurrency bugs result from such violations.In this paper we present a general approach to predicting atomicity violations that is based on techniques from Artificial Intelligence (AI) automated planning. We encode the dynamics of our program abstractly in terms of the properties of observed events from a successful program run. We characterize the generation of a run as a sequential planning problem with the temporally extended goal of achieving a particular pattern of atomicity violation. We have integrated our approach into the PENELOPE concurrency testing tool (Sorrentino, Farzan, & Madhusudan 2010). Initial experiments comparing the run prediction time for an implementation of our approach showed it to be comparable to PENELOPE's static analysis approach. However, there are indications that the planning approach may scale better on longer runs. Further, unlike PENELOPE'S current approach, runs predicted by our approach all correspond to concrete runs of the system. Finally, our planning-based approach has the merit that it can easily accommodate complex atomicity violation patterns by simply modifying the planning goal. For all these reasons, the planning-based approach presented here appears to be a fruitful area for further investigation.

1. Introduction and Background

Testing concurrent programs is an important and timely problem, particularly with the advent of multicore hardware. The problem is challenging because of the potentially large number of possible thread interleavings that exist, even for a fixed input. Programmers often find it difficult to account for all such interleavings when writing or updating code, and it is computationally infeasible to test for all such interleavings even with automated testing tools.

Much state-of-the-art technology for concurrent program testing involves stress testing (i.e., running the program repeatedly with various inputs, and as many threads as possible) with random sleep (or yield) statements inserted in the code to force the concurrent program to take a different path every time (e.g., IBM's ConTest tool[1]). Unfortunately, such methods are limited in their effectiveness at exposing bugs.

A popular approach to testing concurrent programs is to use a set of selection criteria to choose and test a small subset of interleavings (runs) that are likely to lead to bugs. Tools that adopt this approach differ with respect to the selection criteria they employ and with respect to the quality and fidelity of the runs that they generate for subsequent testing. One such tool is the CHESS system developed by Microsoft (Musuvathi & Qadeer 2006). CHESS tests all interleavings that use a bounded number of preemptions (unforced context-switches) with a small bound, based on the belief that most errors will be found in this set.

Another powerful interleaving selection criterion is to select runs that may lead to *atomicity violations* (e.g., (Sorrentino, Farzan, & Madhusudan 2010; Wang *et al.* 2010; Wang & Stoller 2006b; 2006a; Park, Lu, & Zhou 2009; Park & Sen 2008; Yi, Sadowski, & Flanagan 2009)). Atomicity, or *serializability*, is a semantic correctness condition for concurrent programs. Intuitively, a thread interleaving is serializable if it is equivalent to a serial execution, i.e., a thread interleaving that executes an execution block without other threads interleaved in between (Wang *et al.* 2010). A recent study that classified concurrency errors showed that just over two-thirds of bugs in concurrent programs could be attributed to atomicity violations (Lu *et al.* 2008). As such, they define an important class of errors for which interleaving selection criteria have been developed.

For simplicity, many approaches that attempt to predict atomicity-violating runs focus on three-access atomicity violations – patterns of atomicity violations that involve two threads and three global accesses to shared variables. Intuitively, two of these accesses belong to a code block of one thread, and the third access is the interfering access that if executed in the middle of the first two, could cause unwanted results. PENELOPE, a testing tool for concurrent software (Sorrentino, Farzan, & Madhusudan 2010) predict runs that may contain such three-access atomicity violations.

The PENELOPE system follows a three-phase approach.

[1] `http://www.alphaworks.ibm.com/tech/contest/`

In the first phase (monitoring phase), the program is executed on a test input and a sequence of critical program events in the run is recorded. These events form an abstraction of the run which consists of reads and writes to shared variables, lock acquisitions, and lock releases. In the second phase (run prediction phase), PENELOPE first performs a simple and comparably fast static analysis on the (abstract) run to generate the set of all *lock-valid* access patterns possible for this run. The patterns are selected such that for each of them there exists at least one lock-valid predicted run in which atomicity is violated. Note that this *lock-valid* predicted run does not necessarily correspond to an actual run of the concurrent program as data is ignored in the analysis. Then, PENELOPE predicts an atomicity-violating run according to each access pattern. This prediction is performed via static analysis using some heuristics. The predicted runs have the following properties: (1) each run is a permutation of the events of the original run, (2) each run has an atomicity violation, and (3) each run is valid with respect to synchronization operations but may not correspond to a real program execution. In the last phase (rescheduling phase), PENELOPE executes the program according to the predicted runs and the same fixed input. If a predicted run is not feasible it is discarded. Otherwise, the program output is checked to see if any error has occurred. The advantage is that PENELOPE scales very well compared to approaches such as FUSION, discussed below. It is important to note that PENELOPE only observes accesses to shared variables, and synchronization operations from the original run. It is blind to local accesses, and to any computation performed inside a thread. The disadvantage is that PENELOPE may predict some infeasible runs.

The FUSION system (Wang *et al.* 2010) also tries to predict atomicity-violating runs based on an observed run. Like PENELOPE, predicted runs by FUSION maintain properties (1) and (2), however unlike PENELOPE each FUSION run corresponds to a real execution in the program. This is achieved by observing all the operations that happen in the original run (local, global, synchronization, etc), and generating a set of constraints that enforce properties (1) and (2) in addition to ensuring each run is a concrete program execution. These constraints are encoded in a decidable logic, and SMT solvers are used to find a solution (a run) to the set of constraints. The approach is elegant and theoretically interesting, but currently does not scale beyond runs containing a few thousand events. This is while in practice, runs of even small concurrent programs contain millions of events.

In this paper we propose an approach to atomicity-violating run prediction based on techniques from AI automated planning. The novelty of our approach lies in the conceptualization and encoding of the problem as an AI automated planning problem, and in the addition of a constraint that enables us to ignore local computation while ensuring that predicted runs progress through unobserved parts of the run just as they did in the original observed run. Following the general approach of PENELOPE, we assume the existence of a sequence of observed events from a successful program run and use these to abstractly characterize the behaviour of a successful run. Observed events are treated as plan operators with temporal ordering properties. In this context, the prediction of an atomicity-violating run is characterized as a sequential planning problem with the temporally extended goal of achieving a particular pattern of atomicity violation. Our formal characterization of the task is general, and our plan-based problem encoding can be used with a diversity of planners (e.g., heuristic-search based, SAT based). Here we integrate our work into the PENELOPE infrastructure using the Fast Foward (FF) heuristic search planner to generate runs (Hoffmann 2001).

We compared our approach to the current static analysis approach employed by PENELOPE. Initial experimental evaluation indicates that the time for run generation is comparable to that of PENELOPE. However, unlike PENELOPE'S current approach where runs are sound with respect to synchronization violations but can be infeasible, our runs all correspond to concrete runs of the system. Further, as with PENELOPE'S current approach, and unlike other proposed approaches that are guaranteed to generate concrete runs, our approach appears to scale well, at least to the degree necessary to handle the suite of current test programs investigated by the concurrency testing community. FF's run prediction time was more than an order of magnitude faster than PENELOPE'S for the one test suite of significant length. While no conclusions can be drawn from 40 runs on one test suite, it suggests that the planning-based approach may scale better than PENELOPE current static analysis approach and calls for further experimental investigation.

In Section 2, we provide a motivating example that further elaborates upon the problem being addressed. In Section 3, we elaborate on some mathematical foundations underlying the prediction of atomicity-violating runs. In Section 4, we discuss our planning approach to run prediction, while in Section 5, we present our initial experimental results. We summarize and conclude in Section 7.

2. Motivating Example

Given a concurrent program, we want to find bugs, but due to the interleaving explosion problem it is infeasible to try all possible interleavings of threads even for a single test input. Therefore, we prefer to try a subset of runs that are more likely to expose concurrency bugs than others. The interleavings in which atomicity is violated are strong candidates to contain concurrency bugs.

A three-access atomicity violation pattern consists of three access points (e_1, e_2, f), such that e_1 and e_2 are in the same execution block of a thread that is intended to be executed without interference and f is in a different thread. Therefore, if e_1 and e_2 are interfered in between by f then a problem may occur. Hence, given an execution of the program and an atomicity violation access pattern (e_1, e_2, f) existing in that execution, we want to predict a run consisting of events in the given execution such that f occurs between e_1 and e_2. We will then execute the program according to the predicted run, hoping to find bugs.

Figure 2 provides the implementation of the `synchronized` method `addAll` from the built-in Java library class `vector`. The method gets a `collection`, `c` (can be a vector), as its input parameter and then adds

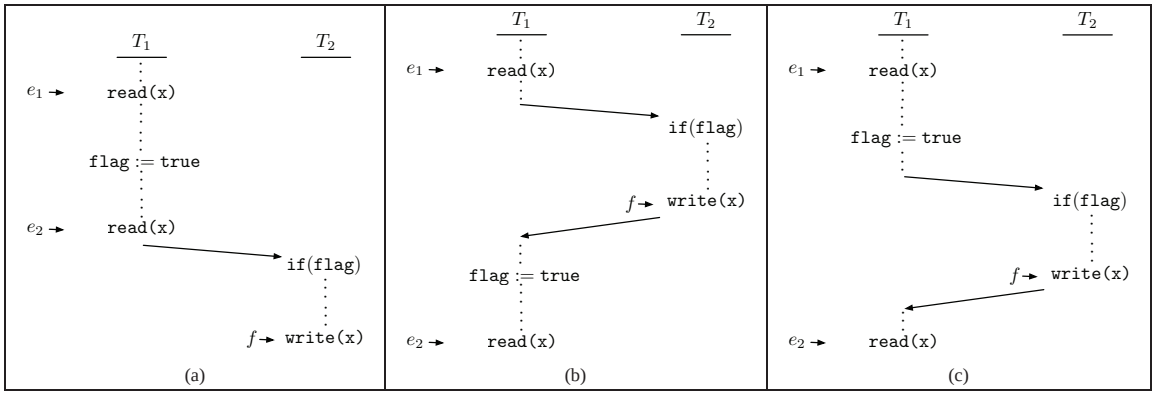


Figure 1: Example. A feasible vs. an infeasible schedule for atomicity violation.

```
public synchronized boolean addAll(Collection c) {
    modCount++;
    int numNew = c.size();
    // ....... possible interference ....
    ensureCapacityHelper(elementCount + numNew);
    Iterator e = c.iterator();
    for (int i=0; i<numNew; i++)
        elementData[elementCount++] = e.next();
    return numNew != 0;
}
```

Figure 2: Method `addAll` of concurrent Java class `vector`.

all elements of it to the end of the current vector. In the method, first method `size` is called on the input collection that returns the total number of elements in it. This number is written on a local variable `numNew`. Then, it iterates over the elements in the input collection for `numNew` times and adds one element to the current vector in each iteration. The problem with this method is that a concurrent thread can just access the collection after the current thread retrieves the number of elements in the collection, and modify it before the current thread finishes copying elements. For example, the concurrent thread can remove some elements from the collection and as a result variable `numNew` does not represent the total number of the variables that should be copied anymore. Therefore, an exception will be raised when the method tries to access elements that are not there anymore. This bug can be capture if we try a run in which atomicity of the method `addAll` is violated by an interfering access to the collection in another thread that reduces the number of elements in it.

In real programs, for each execution block, one can predict many runs that violate atomicity, even with a fixed interfering access. However, not all of the predicted runs may be feasible in the program and it is not effective to try all of them to find a feasible one. Therefore, our goal is to predict runs that are guaranteed to be feasible. Let us clarify this by an example. Consider run R, corresponding to an execution of a program, in Figure 1(a). Variables x and $flag$ are shared between the threads. We assume that the value of $flag$ is initially $false$ and is no write to $flag$ in R other than the one which is performed by thread T_1 in the figure.

If data is ignored during run prediction (which is true for PENELOPE (Sorrentino, Farzan, & Madhusudan 2010)), we would come up with the execution demonstrated in Figure 1(b). This execution, however, is *not* $feasible$ since before the control reaches `write(x)` event in T_2, T_2 will diverge from the above path since it finds the value of $flag$ to be $false$. Consequently, the `write(x)` event will never get executed, and the atomicity violation will never get hit.

In our approach, we ensure that in the predicted run thread T_2 will read the same value for $flag$ as it did in the original run. Therefore, `write(x)` will be executed. Our approach predicts a run represented in Figure 1(c). Of course, the violation occurs in this run. The feasibility of predicted runs are guaranteed if every read from a shared variable in the predicted run reads the same value as it did in the original run. In this case, we can guarantee that the local computation that follows the read (dots in Figure 1), will remain the same as the original run.

3. Foundations

3.1 Programs and Traces

Given a run of a program, each thread executes a series of execution blocks that are logical units of code intended by the programmer to be executed without interference. An execution block is a sequence of computations and synchronization. Every read or write to a shared variable is considered as a global computation. The sequence of reads and writes to local variables between two consecutive global computations of a thread is considered as a single local computation.

We assume an infinite set of thread identifiers $T = \{T_1, T_2, ...\}$ and define an infinite set of shared variables $SV = \{sv_1, sv_2, ...\}$ that the threads can access. Without loss of generality, we assume that each thread T_i can also access a single local variable lv_i. Let $V = SV \bigcup_i lv_i$ represent the set of all variables. We define $Val(x)$ representing the set of possible values that variable $x \in V$ can get, and $Init(x)$ representing the initial value of x. We also fix a set of global locks L.

A set of actions that a thread T_i can perform on a set of variables $X_i = SV \cup \{lv_i\}$ and global locks L is defined as

$\sum_{T_i} = \{T_i : \triangleright, T_i : \triangleleft\} \cup \{T_i : read_{x,val},\ T_i : write_{x,val} \mid x \in SV \text{ and } val \in Val(x)\} \cup \{T_i : lc_i\} \cup \{T_i : acquire(l), T_i : release(l) \mid l \in L\}$. Actions $T_i : \triangleright$ and $T_i : \triangleleft$ correspond to the beginning and the end of execution blocks in thread T_i, respectively. Actions $T_i : read_{x,val}$ and $T_i : write_{x,val}$ correspond to reading value *val* from and writing value *val* to shared variable *x*, respectively. Action $T_i : lc_i$ corresponds to a local computation that accesses only lv_i. Action $T_i : acquire(l)$ represents acquiring lock l and action $T_i : release(l)$ represents releasing lock l by thread T_i. Define $\sum = \bigcup_{T_i \in T} \sum_{T_i}$ as the set of all actions.

A word $\sigma \in \sum^*$ is lock-valid if it respects the semantics of the locking mechanism. Formally, let $\sum_l = \{T_i : acquire(l), T_i : release(l) \mid T_i \in T\}$ denote the set of locking actions on lock l. Then σ is lock-valid if for every $l \in L$, $\sigma|_{\sum_l}$ is a prefix of $[\bigcup_{T_i \in T}(T_i : acquire(l)\ T_i : relese(l))]^*$.

A word $\sigma \in \sum^*$ is data-valid if for each read action $r = read_{x,val}$ in σ, either the last write action to variable *x* writes value *val* (i.e. write action is $w = write_{x,val}$) or there is no write action to variable *x* before the read action *r* and *val* is the initial value of *x*. Formally, for each *i* such that $\sigma[i] = read_{x,val}$ one of the following hold:

- There is a $j < i$ such that $\sigma[j] = write_{x,val}$ and there is no $j < j' < i$ such that $\sigma[j] = write_{x,val'}$ for any val', or
- There is no $j < i$ such that $\sigma[j] = write_{x,val'}$, for some val', and $val = Init(x)$.

In the former case, action *r* is said to be coupled with write action *w* (represented by $Coupled_{r,w}$) and in the latter case action *r* is reading the initial value of *x* (represented by $Coupled_{r,init}$).

Let $ExecBlk_{T_i} = (T_i : \triangleright).[(lc_i\ GlobComp_i)^* lc_i].(T_i : \triangleleft)$ where $GlobComp_i = \{T_i : read_{x,val}, T_i : write_{x,val} \mid x \in SV \text{ and } val \in Val(x)\} \cup \{T_i : acquire(l), T_i : release(l) \mid l \in L\}$. An execution block of a thread T_i is a word in $ExecBlk_{T_i}$. Intuitively, an execution block can perform a number of global computations while there is a local computation between two consecutive global computations. $ExecBlks_{T_i} = (ExecBlk_{T_i})^*$ denotes the set of all possible sequences of execution blocks for a thread T_i.

Definition 1 (Run) *A run over a set of threads T, variables V, and locks L, is a word $\sigma \in \sum^*$ such that for each $T_i \in T$, $\sigma|_{T_i}$ belongs to $ExecBlks_{T_i}$, and σ is both lock-valid and data-valid. Let $Run_{T,V}$ denote the set of all runs over threads T and variables V.*

In other words, a run is a lock-valid, data-valid sequence of actions such that its projection to any thread T_i is a word divided into a sequence of execution blocks, where each execution block begins with $T_i : \triangleright$, is followed by a sequence of alternating local and global computations, and ends with $T_i : \triangleleft$.

Let $rn = e_1 ... e_n$ be a run of a program. The *occurrence* of actions in runs are referred to as *events* in this paper. Formally, the set of events of the run is $E = \{1, \ldots, n\}$, and there is a labeling function λ that maps every event to an action, given by $\lambda(u) = e_u$.

While the run *rn* defines a total order on the set of events in it $(E, \leq)$, there is an induced total order between the events of each thread. We formally define this as $\sqsubseteq_i$: for any $i, j \in E$, if e_i and e_j belong to thread T_i and $i \leq j$ then $i \sqsubseteq_i j$.

We define $W_{x,val}$ as the set of write events that write value *val* to *x*.

Definition 2 (Precisely Predictable Runs) *Given a run, rn, over a set of threads T, variables V, and locks L, run rn' is precisely predictable from rn if for each $T_i \in T$:*

1. *$rn'|_{T_i}$ is a prefix of $rn|_{T_i}$,*
2. *rn' is lock-valid,*
3. *for each read event $r = read_{x,val}$ in rn' we have either $Coupled_{r,w}$ such that $w = write_{x,val} \in W_{x,val}$ or $Coupled_{r,init}$.*

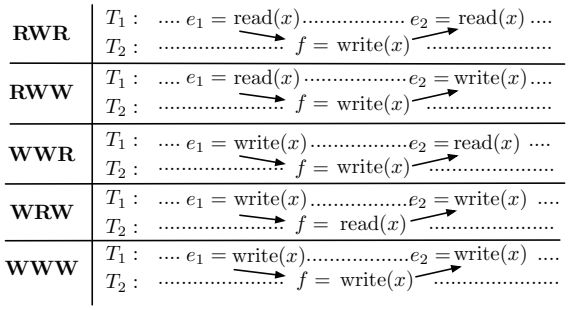
$Pred(rn)$ denotes the set of precisely predictable runs from rn.

Intuitively, precisely predictable runs of *rn* are lock-valid permutations of the events in *rn*, respecting the program order, such that each read *r* from a shared variable can be coupled only with a write event writing the value being read by *r* in *rn*.

Theorem 1 *Let P be a program and rn be a run corresponding to an execution of P. Then every precisely predictable run $rn' \in Pred(rn)$ is feasible in P.*

3.2 Access Patterns

There are *five* different types of access patterns that correspond to simple three-access atomicity violations of *two threads* and *one variable*. An access pattern *p* consists of three events (e_1, e_2, f), such that e_1 and e_2 are in the same block of a thread that is intended to be executed without interference and f is in a different thread where all accesses are to the same shared variable and f conflicts with both e_1 and e_2. Two accesses have conflict with each other if at least one of them is a writes access. As a result, a pattern should be of one of the following formats:



4. Predicting Atomicity-Violating Runs via Planning

In this section we describe our conceptualization and encoding of atomicity-violating run prediction using planning. We describe the basic correspondence between the run prediction task and that of planning. This is followed by a more

detailed description of the encoding for a particular three-access atomicity violation pattern. We begin with a brief review of planning.

4.1 Planning

A classical planning problem (Nau, Ghallab, & Traverso 2004) is defined as a tuple $P = (S_0,\ F,\ A,\ G)$ where F is a finite set of atomic facts, $S_0 \subseteq F$ is the *initial state*, and A is a finite set of deterministic *actions*. Each action $a \in A$ is described by a tuple $pre(a)$, $add(a)$, $del(a)$ where $pre(a)$ is a pair $(pre^+(a),\ pre^-(a))$ of disjoint subsets of F representing positive and negative preconditions of action a, respectively. Each action has a positive and a negative set of effects represented by $add(a)$ and $del(a)$, respectively, which are disjoint subsets of F.

A *planning state* is a subset of elements in F. In classical planning, complete information about the planning state is assumed. Therefore, every $f \in F$ that is not explicitly mentioned in a planning state, including the initial state, is assumed to be false in that state. Action a is applicable in a planning state $s \subseteq F$ iff $pre^+(a) \subseteq s$ and $pre^-(a) \cap s = \emptyset$. Applying action a in state s would result in a new, successor state, $succ(a,\ s) = (s \backslash del(a)) \cup add(a)$. The goal G corresponds to a set of planning states and a *plan*, $\vec{a}$, consists of a finite sequence of actions $a_0, ..., a_n$ which, when applied to the initial state, will produce a state in G.

A temporally extended planning problem, P (e.g., (Baier & McIlraith 2006)), in this setting is a classical planning problem $P = (S_0,\ F,\ A,\ G)$ where the goal G is not restricted to a final-state goal, but rather is a set of facts together with some ordering constraints. Such *temporally extended goals* are often specified in linear temporal logic (LTL) (Pnueli 1977). A plan for a temporally extended goal G is simply a sequence of actions, $\vec{a}$, which when applied to the inital state results in a sequence of actions that entails G.

Automated planning problems are typically encoded in terms of a *planning domain description* that describes the dynamics of the planning problem – the actions, their preconditions and effects, and by a *problem instance* that includes a description of the intial state and the goal. The de facto standard for specifying planning domains and planning instances is PDDL, the Plan Domain Definition Language (McDermott 1998). PDDL has evolved over the years to address increasing needs for expressiveness, and is firmly established as the input language for most automated planning systems. Automated planning systems themselves vary in their approaches to plan generation. Two popular approaches are those based on heuristic search, as exemplified by the very successful FF planner used here (Hoffmann 2001), and those based on SAT. While these systems take PDDL as input, most transform the PDDL into an internal representation that is tailored to the needs of their search algorithm.

4.2 Approach

We characterize the problem of predicting a feasible run that has the potential to violate atomicity as a sequential planning problem with the goal of achieving a particular pattern of atomicity violation. Using the observed abstract run as an (approximated) specification of the behaviour of our program, we encode the dynamics of our program as an initial state S_0, a set of facts F, and a set of actions A. Each action corresponds to an event (read, write, ...) within the observed abstract run. The facts record which actions (program events) have been executed and some specific properties relating to read-write synchronization and lock availability. The preconditions and effects for individual actions are written so as to enforce the necessary ordering of events imposed by the threads and the abstract run, and also to enforce the read-write constraints that ensure that any plan generated from this planning instance corresponds to a feasible run of the concurrent program. We illustrate this encoding in the section that follows.

An important contribution of this paper, and one that is not limited to the planning approach presented here, is the insight into how to enforce the prediction of *feasible* runs – runs that correspond to concrete executions of the program – while ignoring local computation. In the observed run (which is trivially feasible), each read event is coupled with a write event that determines the value being read by the read event. This write event is the most recent write to the corresponding shared variable that occurs before the read event (if there is no write to the variable before the read event then the initial value of the variable is being read). In our approach, we consider a class of feasible runs in which each read event reads in the same value as it did in the observed run. This forces the paths that are taken by each thread to remain the same as the paths in the observed run. The feasibility of predicted runs is guaranteed as long as each thread is guaranteed to take the same path as it took in the observed run. Note that the read events in the predicted run are allowed to be coupled with write events other than the ones they were coupled with in the observed run as long as data-validity is preserved.

An access pattern for an atomicity violation is either a partial or full ordering of the subset of events that define the particular access pattern for which the run is being generated. For example, the simple three-access we have been discussing is $e_1 < f < e_2$, where $<$ here is the order of occurrence. In PENELOPE access patterns are identified via static analysis of the abstract run in the prediction phase. In our planning approach to run prediction, such access patterns are treated as *temporally extended goal*. In the most general case, they can be specified as an LTL formula where the occurrence of an event e_i is encoded as the fact $Happened\_e_i$. As such the task of generating a run that achieves a pattern of atomicity violation is viewed as the automated generation of a plan with a temporally extended goal. In so doing, a plan that achieves this temporally extended goal corresponds to a feasible predicted run for an initial portion (a prefix) of an atomicity-violating run. I.e., it is the prefix of a feasible run of the concurrent program that realizes that access pattern and that has the potential to violate atomicity. Exploiting results proposed in (Baier & McIlraith 2006) such problems can be transformed into classical planning problems, by exploiting an established correspondence between LTL and Büchi automata. In more restrictive cases, such as the three-access pattern described here, there is an even sim-

pler transformation of the temporally extended goals into a final-state goals (Haslum & Grastien 2011) via what is effectively precondition control on the actions. We illustrate the use of this in the encoding section that follows.

4.3 Encoding

In this section we present schemas or templates for the general PDDL encoding we employ for run predication. For ease of explanation, syntax does not strictly conform to PDDL syntax but is expressively equivalent. We illustrate the encoding with respect to the set of event types observed by PENELOPE and the three-access pattern (e_1, e_2, f).

The concrete problem that we wish to address is *"Given a run R and an access pattern (e_1, e_2, f) is there any precisely predictable run from run R such that f happens after e_1 and before e_2?"*. Notice that if f is a read event from x, then we would like f to read a value other than the value it read in the observed run. Indeed, we would like to diverge from the observed run at f by letting f read a different value. Note that once f reads a different value, we cannot schedule any event in R that occurs after f and is dependant on the value read by f because such events might not happen in the predicted run when f is reading a different value; and in particular we cannot ensure that e_2 will occur after f. The prediction problem is hence formulated as follows: *"Given a run R and an access pattern (e_1, e_2, f) find a precisely predictable run of R that executes e_1 followed by f and e_2 has not occurred."*

Encoding Events and Program Order A run consist of a set of events which are reads and writes to shared variables, lock acquires, and lock releases that appeared in the run. Each event is encoded as an action in the planning domain. Therefore, we may have four different types of actions: read, write, lock acquire, and lock release actions.

Suppose σ is the given run. Let $\sigma|T_i = \{t_1, t_2, ..., t_m\}$ be the projection of the run on thread T_i, i.e. the set of events in the run that belong to thread T_i. Also, suppose that we have $t_1 \sqsubseteq_i t_2, t_2 \sqsubseteq_i t_3, ..., t_{m-1} \sqsubseteq_i t_m$. According to the program order, event t_{j+1} in thread T_i cannot be executed unless event t_j in thread T_i is executed. To encode the program order of events, a predicate is considered according to each action that represents the application of the action.

Let action $Ac_{i,j}$ represent event $t_{i,j}$, i.e. the j^{th} event in thread T_i. Predicate $(Done_{i,j})$ is used to show that $Ac_{i,j}$ is applied. Therefore, $(Done_{i,j})$ should be initially $false$ and become $true$ after the application of $Ac_{i,j}$.

In the planning domain, an action may be applied several times in order to find a plan. Note that in our case, each action is representing an event in the run. Therefore, each action cannot be applied more than once. To encode this fact, a predicate $(Not\_Executed_{i,j})$ is considered according to each action $Ac_{i,j}$ representing that the action has not been applied yet. These predicates should be initially $true$ and become $false$ after the application of the corresponding actions.

Putting these constraints together, the following is the template for event $t_{i,j}$. Each action might have other preconditions and effects according to the type of the event they represent.

$$
\begin{array}{l}
(:action\ Ac_{i,j} \\
\quad :precondition(and(Done_{i,(j-1)})\ (Not\_Executed_{i,j})...) \\
\quad :effect(and(Done_{i,j})\ (not\ (Not\_Executed_{i,j}))...) \\
\quad)
\end{array}
$$

Note that if an actions is encoding the first event in a thread T_i, then the precondition set consists of only $(Not\_Executed_{i,1})$.

Encoding Write Events There might be several write events in the run to a single variable. Suppose that W_x represent the set of all write events to variable x in the given run. To keep the track of the most recent write event to variable x, we consider a set of predicates, represented by $writes(x)$:

$$writes(x) = \{(x_{m,n}) |\ t_{m,n} \in W_x\} \bigcup \{(x_{init})\}$$

Predicate (x_{init}) represents the initial value of x. It is initially $true$ indicating that no write event has been performed to x. Predicate $(x_{m,n})$ denotes that $t_{m,n}$ has performed the most recent write to variable x. Predicates of this type are all initially $false$. Predicate $(x_{m,n})$ becomes $true$ when $Ac_{m,n}$ is applied. Semantically, at each point of time only one of the predicates in $writes(x)$ can be $true$.

Suppose that $Ac_{i,j}$ corresponds to a write event to variable x. The following shows how this event is encoded:

$$
\begin{array}{l}
(:action\ Ac_{i,j} \\
\quad :precondition(and(Done_{i,(j-1)})\ (Not\_Executed_{i,j})) \\
\quad :effect(and(Done_{i,j})\ (not\ (Not\_Executed_{i,j}))\ (x_{i,j}) \\
\qquad\qquad \forall p \in [writes(x) - \{(x_{i,j})\}] : (not\ (p))) \\
\quad)
\end{array}
$$

In the effect set, event $t_{i,j}$ is set to be the most recent write event to x by $(x_{i,j})$. In addition, any write event to x other than event $t_{i,j}$ is set not to be the most recent write event to x by $\forall p \in [writes(x) - \{(x_{i,j})\}] : (not\ (p))$.

Encoding Read Events To obtain a precisely predicted schedule, which guarantees feasibility, each read event from variable x is allowed to be coupled with only a write event to x that writes the same value as being read by the read event in the observed run.

However, we are not encoding the real values in the planning domain and it is just enough to know the set of write events that a read event can be coupled with. Suppose that event $t_{i,j} = read_{x,val}$ is reading value val from variable x and $Write_{x,val}$ denotes the set of events that write value val to variable x. The read event $t_{i,j}$ can be coupled with any write in $Write_{x,val}$. Therefore, for each write event $t_{m,n} \in Write_{x,val}$ an action is considered as follows:

$$
\begin{array}{l}
(:action\ Ac_{i,j}\_coupled_{m,n} \\
\quad :precondition(and(Done_{i,(j-1)})\ (Not\_Executed_{i,j}) \\
\qquad\qquad\qquad (x_{m,n})) \\
\quad :effect(and(Done_{i,j})\ (not\ (Not\_Executed_{i,j})) \\
\quad)
\end{array}
$$

Having $(x_{m,n})$ in the precondition of the action would force the read event to be coupled with the write event $t_{m,n}$. Here, we consider several actions according to a read event that allow the read event to be able to couple with each write event in $Write_{x,val}$.

Encoding Lock Acquiring and Lock Releasing Events Each lock can be obtained by at most one thread at each point in time. Therefore, if a lock is obtained by thread T then other threads cannot acquire it unless T releases the lock. Assume that $L = \{l_1, ..., l_m\}$ is the set of locks used in the run. To guarantee lock-validity, a predicate $(Available_{l_i})$ is considered according to each lock l_i, representing lock l_i is not obtained by any thread and is free. These predicates are initially *true* since all of the locks are available at the beginning of the run.

The actions corresponding to lock acquiring events on lock l have $(Available_l)$ in their precondition set and $(not\ (Available_l))$ in their effect set. Having $(Available_l)$ in the precondition set requires lock l to be available before the application of the action. Note that after performing the action, lock l is not available any more and cannot be acquired by any other thread. On the other hand, the actions corresponding to lock releasing events of lock l have $(Available_l)$ in their effect set, making the lock available again.

Encoding Atomicity Violation and Goal Atomicity is violated with respect to pattern (e_1, e_2, f) when e_1 is executed followed by f and e_2 never occurs. Two predicates $(Happened\_e_1)$ and $(Happened\_f)$ are considered to encode the occurrence of events e_1 and f, respectively. These predicates are both initially $false$, representing that none of these events has happened at the beginning of the run.

Suppose that $Ac_{i,m}$, $Ac_{i,n}$, and $Ac_{j,k}$ are the actions corresponding to events e_1, e_2, and f, respectively. Action $Ac_{i,m}$ has $(Happened\_e_1)$ in its effect set. Action $Ac_{j,k}$ has $(Happened\_e_1)$ in its preconditions set, requiring e_1 to happen before, and $(Happened\_f)$ in its effect set. Action $Ac_{i,n}$ has $(Happened\_f)$ in its precondition set forcing it to not occur before $Ac_{j,k}$.

From this, the classical final-state goal is defined as $(:goal(Happened\_f))$. Since action $Ac_{j,k}$ (event f) cannot be applied before action $Ac_{i,m}$ (event e_1) or after action $Ac_{i,n}$ (event e_2), atomicity is violated whenever it is applied. We see in this instance that the temporally extended goal has been easily compiled into constraints on the evolution of the domain. With more complex access patterns, alternative encodings may be necessary.

Initial State In the initial state, all shared variables have their initial values and also all locks are available. Therefore we have $\bigwedge_{\forall x \in SV}(x_{init}) \bigwedge_{\forall l \in L}(Available_l)$, where SV and L represent the set of all shared variables and the set of all locks, respectively. Predicate (x_{init}) becomes $false$ when a write action to variable x is applied and also $(Available_l)$ becomes $false$/$true$ after the application of each lock acquire/release action on lock l.

Proposition 1 *Suppose that given run R and an access pattern (e_1, e_2, f), P is a classical planning problem generated according to the above encoding. Every plan π (if it exists), represents a precisely predictable run from R in which atomicity is violated according to the access pattern (e_1, e_2, f).*

Lemma 1 *Suppose that given run R and an access pattern (e_1, e_2, f), P is a classical planning problem generated according to the above encoding. Every plan π (if it exists), represents a feasible run.*

This follows trivially from Proposition 1. and Theorem 1.

4.4 Example

Consider the simple run in Figure 1(a). The run consists of the sequence of events $t_{1,1} : read_{x,val}$, $t_{1,2} : write_{flag,true}$, $t_{1,3} : read_{x,val}$, $t_{2,1} : read_{flag,true}$, $t_{2,2} : write_{x,val'}$. Both read events from variable x are reading the initial value of x and the read of $flag$ by $t_{2,1}$ is coupled with the write event $t_{1,2}$. Considering $(e_1, e_2, f) = (t_{1,1}, t_{1,3}, t_{2,2})$ to be a three-access atomicity violation pattern, the corresponding planning problem would consists of the following actions:

$(:action\ Ac_{1,1}$
$:precondition(and\ (Not\_Executed_{1,1})\ (x_{init}))$
$:effect(and(Done_{1,1})\ (not\ (Not\_Executed_{1,1}))$
$(Happened\_e_1))$
$)$

$(:action\ Ac_{1,2}$
$:precondition(and(Done_{1,1})\ (Not\_Executed_{1,2}))$
$:effect(and(Done_{1,2})\ (not\ (Not\_Executed_{1,2}))$
$(not(flag_{init}))\ (flag_{1,2}))$
$)$

$(:action\ Ac_{1,3}$
$:precondition(and(Done_{1,2})\ (Not\_Executed_{1,3})$
$(x_{init})\ (Happened\_f))$
$:effect(and(Done_{1,3})\ (not\ (Not\_Executed_{1,3})))$
$)$

$(:action\ Ac_{2,1}$
$:precondition(and\ (Not\_Executed_{2,1})\ (flag_{1,2}))$
$:effect(and(Done_{2,1})\ (not\ (Not\_Executed_{2,1})))$
$)$

$(:action\ Ac_{2,2}$
$:precondition(and(Done_{2,1})\ (Not\_Executed_{2,2})$
$(Happened\_e_1))$
$:effect(and(Done_{2,2})\ (not\ (Not\_Executed_{2,2}))$
$(not(x_{init}))\ (x_{2,2})\ (Happened\_f))$
$)$

where only predicates $(Not\_Executed_{1,1})$, $(Not\_Executed_{1,2})$, $(Not\_Executed_{1,3})$, $(Not\_Executed_{2,1})$, $(Not\_Executed_{2,2})$, x_{init}, and $flag_{init}$ are initially *true* and the goal is $(:goal(Happened\_f))$. The plan for this goal is the sequence of actions $Ac_{1,1}, Ac_{1,2}, Ac_{2,1}, Ac_{2,2}$.

		Run Information					Run Prediction						
Program (Line of Code)	Input	Threads	Vars	Locks	Run Length	No. of Access Patterns	No. of Feasible Runs by PENELOPE	No. of (Feasible) Runs by FF	No. of Feasible Runs FF plus PENELOPE	Percentage of Feasible Runs PENELOPE - FF	Avg Time per Run by PENELOPE	Avg Time per Run by FF	Bugs
Pool 1.2 (5.8K)	Test1	4	18	2	356	374	275	280	285	96% - 98%	0.01s	0.01s	1
Pool 1.3 (7K)	Test1	4	18	2	422	417	211	195	300	70% - 65%	0.02s	0.15s	1
Elevator (566)	Data1	3	503	50	63K	43	41	42	42	97% - 100%	3.76s	0.20s	0
Vector (1.3K)	Test1 Test2	4 4	24 24	2 2	353 350	3 23	2 23	3 23	3 23	66% - 100% 100% - 100%	0.02s 0.01s	0.015s 0.01s	1 1
Apache FTPserver (22K)	lgn_script	5	112	4	578	33	7	29	33	21% - 87%	0.02s	0.01s	3

Table 1: Experimental Results: Percentage of feasible runs predicted by PENELOPE and FF (4^{th} column in Run Prediction) is obtained by dividing the number of feasible runs predicted by PENELOPE and FF, respectively, by the total number of feasible runs predicted considering both of them (i.e. 3^{rd} column in Run Prediction).

5. Implementation and Experimental Results

In the previous section, we proposed a means of encoding the task of feasible run prediction as a planning problem. The encoding supports run predication using a variety of plan generation algorithms. Here we describe experimental results using the FF heuristic search algorithm. We compare our experimental results to those generated by PENELOPE's static analysis approach. While the planning approach extends beyond three-access atomicity violation patterns, we were limited by PENELOPE to consider only this class of patterns in our experiments.

The PENELOPE tool executes each program on a provided test input. Then, the execution of the program is monitored and a set of three-access patterns are extracted from the observed run. Access patterns are extracted in such a way that at least one lock-valid run can be predicted in which atomicity is violated according to the pattern. Therefore, PENELOPE can always come up with a lock-valid predicted run that violates atomicity according to each access pattern. Since runs predicted by PENELOPE are not guaranteed to be feasible (data is ignored), the program is re-executed according to a predicted run to see whether it is feasible or not. In the case of feasibility, the outputs of the program are just examined to see whether they are as expected.

Although the planning approach we are advocating can be used to generate a run of the program from the beginning until atomicity is violated according to an access pattern $\alpha = (e_1, e_2, f)$, in the implementation within PENELOPE, we isolate a segment of the observed run σ_α that is *relevant* to α and use the planning approach to reorder the events in σ_α consistently to find an atomicity violation. Therefore, the observed run is cut at some point before e_1 and f. The algorithm for finding an appropriate and efficient cut-point is out of the scope of this paper. The prefix of the run before the cut-point, called σ_{pre}, is considered as the prefix of the predicted run and the planning approach is used to generate only a fragment of the run just after the cut-point till f is executed after e_1. Note that this is an optimization step that allows us to deal with shorter run segments.

Theorem 2 *Given an observed run σ, and an access pattern $\alpha = (e_1, e_2, f)$, suppose that we cut σ at some cut-point before e_1 and f. Let σ_{pre} represent the prefix of the run before the cut-point and σ_α be the suffix of the run after the cut-point. Our proposed encoding provided with σ_α and α would generate a fragment of the entire predicted run, called σ', having f as the last event. Then, the predicted run $\sigma'' = \sigma_{pre}.\sigma'$ is feasible.*

Proof Sketch: The proof follows from Theorem 1 and the fact that σ_{pre} is feasible.

We implemented a translation tool that takes a given run and a three-access atomicity violation pattern as input and automatically generates a PDDL planning problem according to the proposed encoding. We augmented the run prediction phase of PENELOPE with this tool. Given the observed run, for each three-access pattern, we automatically generate a planning problem such that any plan for it would represent a feasible run in which atomicity is violated according to the access pattern. Here, we use Fast Forward (FF) (Hoffmann 2001) to find such runs. It is obvious that FF is used as a box and it is possible to use any other planner as well. For every run predicted by FF, we re-execute the program based on the prediction to see whether a bug can be found. In cases where FF does not predict a feasible run, we try the corresponding run predicted by PENELOPE. This is done in order to assess the value of PENELOPE augmented with the planning approach.

Our experimental analysis was driven, in part, by the following questions: (1) How effective are the approaches in predicting feasible runs?, (2) How time efficient are the approaches in predicting feasible runs?, and (3) How effective are the approaches in finding bugs? To answer the first question, we compared the number of (trivially feasible) predicted runs by FF with the number of feasible runs predicted by PENELOPE. To answer the second question, we compare the average time taken by FF in predicting a run with that

of PENELOPE. To answer the last question, we report the number of the bugs found by our approach.

Our benchmark suite consists of 5 concurrent Java programs that use `synchronized` blocks and methods as their means of synchronization. They include `elevator` from (von Praun & Gross 2001), `Vector` from Java libraries, `Pool` (two different releases) from the Apache Commons Project, and `Apache FtpServer`.

Table 1 summarizes information regarding the i) observed runs, ii) the runs that were predicted, and iii) the number of bugs found in each program. With respect to the observed runs, we record the number of threads, variables, and locks; the length of the run; and the number of access patterns. Under run prediction, we report the number of feasible runs predicted by PENELOPE, by FF, and by the combination of the two. I.e., in cases where FF alone found no feasible run, the corresponding run predicted by PENELOPE was used. We also report the average time taken by each of PENELOPE and FF to predict a run. From this table, we make the following observations.

- *The number of feasible runs predicted by each approach is comparable (with one exception where the planning approach was notably superior).* Recall that the planning approach finds a set of precisely predictable runs which are guaranteed to be feasible while PENELOPE only guarantees lock-validity and the predicted runs are not guaranteed to be feasible. The planning approach predicted considerably more feasible runs in the `Apache FtpServer` test suite.
- *Time efficiency is comparable (with one exception where the planning approach was notably superior).* The average time required for each of PENELOPE and the planning approach to predict a run was comparable with one exception. In the `elevator` test suite, the planning approach was more than an order of magnitude faster than PENELOPE. Note that the execution length of the observed run for this program was also an order of magnitude greater than in the other programs. This leads us to question whether the planning approach might scale better for longer runs and suggests the need for further experimental evaluation.
- *Both approaches are effective at finding bugs (with one exception where* PENELOPE *was superior).* We ran the programs under the test inputs several times, randomly generating runs. These random runs found (almost) none of the reported bugs. The number of bugs found in each run prediction approach (the planning approach, PENELOPE, and the combination of the two) was the same in all cases with the exception of `Apache FtpServer`. There, none of the runs predicted by the planning approach found the bugs. In contrast, PENELOPE found 3 bugs, as did the approach that combined PENELOPE with the planning approach in cases where the latter did not find a feasible run.
- *The combination of both approaches has merit.* All runs predicted by the planning approach are feasible. However, in cases where the approach was unable to find a feasible run, the combination approach tried a run predicted by PENELOPE. Some of these lock-valid runs could be feasible, but are not guaranteed to be so. In the case of `Pool 1.3`, the combination approach outperformed each of the planning approach and PENELOPE individually, demonstrating the merit of this combined approach and indicating that some of PENELOPE's runs were feasible.
- *Despite the lack of guarantee* PENELOPE *predicted feasible runs quite frequently in some cases.* In some cases, the number of runs predicted by PENELOPE that turned out to be feasible was comparable to or slightly exceeded the number of feasible runs predicted with the planning approach (e.g. `Pool 1.3`). Further experiments are warranted to evaluate the frequency of this phenomenon.

Finally, there is an important issue in comparing the planning approach with PENELOPE, which is not reflected in Table 1. Since the runs predicted by PENELOPE are not guaranteed to be feasible, one has to re-execute the program according to the predictions to find out whether the PENELOPE runs are feasible or not. In cases where PENELOPE predicts a large number of infeasible runs, this overhead can significantly impact overall performance.

6. Related Work

In Section 1, we provided significant discussion of various approaches to testing concurrent programs. Nevertheless, we only discussed a subset of the previous work on detecting atomicity-violating runs. In this section we complete this discussion by referencing other related work. None of it exploits techniques related to AI planning and as such, underlies some of the novelty of our approach.

In two papers (Wang & Stoller 2006b; 2006a), Wang and Stoller study the prediction of runs that violate atomicity from a single run. However, they keep track of a large graph, which doesn't scale as the size of executions increases.

A recent related work is the tool CTRIGGER (Park, Lu, & Zhou 2009), that has similar motivation as PENELOPE (Sorrentino, Farzan, & Madhusudan 2010) in predicting and scheduling atomicity violations. CTRIGGER uses techniques that are entirely heuristics in nature and the predicted runs are not guaranteed to be feasible.

There is a recent work on *active randomized testing* for atomicity in the tool ATOMFUZZER (Park & Sen 2008). It uses randomization techniques that executed and holds threads at strategic points to try to find atomicity. However, it can interrupt the threads at wrong positions and therefore it may not be able to create all atomicity violations that could happen with a given input.

There has also been work on finding atomicity violations by using a *generalized* dynamic analysis of an execution. SIDETRACK is a new tool (Yi, Sadowski, & Flanagan 2009) that finds atomicity violations by a generalized analysis of the observed run. Note that this technique does not examine runs that are causally different from the original run, and hence does not do any rescheduling.

Apart from the related work discussed above, *atomicity violations based on serializability* have been suggested to be effective in finding concurrency bugs in many works (Flanagan & Freund 2004; Wang & Stoller 2006b). Lipton trans-

actions have been used to find atomicity violations in programs (Lipton 1975; Flanagan & Qadeer 2003).

7. Summary and Conclusion

Atomicity-violating runs are some of the best candidates runs to use when searching for bugs in concurrent programs. Indeed, a recent study claimed that over two-thirds of bugs in concurrent programs could be attributed to atomicity-violating runs. As outlined in this paper, a number of approaches have focused on predicting runs that violate three-atomicity access patterns. These approaches suffer from either an inability to scale beyond runs containing a few thousand events, or they do not guarantee the feasibility of the predicted runs. In this paper, we conceptualized and encoded the problem of predicting atomicity-violating runs as an AI automated planning task. We implemented a translator to automatically generate our encoding from an observed abstract run, combining it with FF, a well-known heuristic-search based planner, to perform the actual run generation. An important property of our approach is that, unlike previous related work, it guarantees the *feasibility* of predicted runs. We undertook a preliminary evaluation of the effectiveness of our approach. Unfortunately, a more extensive evaluation was not possible because of a lack of large test suites, and because of PENELOPE's inability to consider anything more complicated that an three-access atomicity violation. Nevertheless, in the experiments that were performed, our approach scaled well to large runs. While our approach was generally consistent with PENELOPE's, in several instances (with long runs) it significantly outperformed PENELOPE. This observation underlines the need for further experimental evaluation.

Perhaps one of the most compelling aspects of the proposed approach is its generality and extensibility. While the encoding was used here with one particular heuristic-search planner, it could equally well be used with other planners, with a general SAT solver or with a SAT-based planner that is tailored to the encoding. Further, the characterization of atomicity patterns as temporally extended goals, supports the simple extension of the encoding to more complex atomicity patterns. These are all fruitful avenues for future work.

Acknowledgements The authors gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

Baier, J., and McIlraith, S. 2006. Planning with first-order temporally extended goals using heuristic search. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI'06)*, 788–795.

Flanagan, C., and Freund, S. N. 2004. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of 31st Symposium on Principles of Programming Languages (POPL)*, 256–267.

Flanagan, C., and Qadeer, S. 2003. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 338–349.

Haslum, P., and Grastien, A. 2011. Diagnosis as planning: Two case studies. In *Proceedings of the International Scheduling and Planning Applications Workshop SPARK*.

Hoffmann, J. 2001. Ff: The fast-forward planning system. *AI magazine* 22:57–62.

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

Lu, S.; Park, S.; Seo, E.; and Zhou, Y. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of 13th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 329–339.

McDermott, D. V. 1998. PDDL — The Planning Domain Definition Language. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Musuvathi, M., and Qadeer, S. 2006. Chess: Systematic stress testing of concurrent software. In *Proceedings of the 2006 International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR)*, 15–16.

Nau, D.; Ghallab, M.; and Traverso, P. 2004. *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann.

Park, C.-S., and Sen, K. 2008. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 135–145. New York, NY, USA: ACM.

Park, S.; Lu, S.; and Zhou, Y. 2009. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 25–36.

Pnueli, A. 1977. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS'77)*, 46–57.

Sorrentino, F.; Farzan, A.; and Madhusudan, P. 2010. Penelope: weaving threads to expose atomicity violations. In *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, 37–46. New York, NY, USA: ACM.

von Praun, C., and Gross, T. R. 2001. Object race detection. *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)* 36(11):70–82.

Wang, L., and Stoller, S. D. 2006a. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 137–146.

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

Wang, C.; Limaye, R.; Ganai, M.; and Gupta, A. 2010. Trace-based symbolic analysis for atomicity violations. In *TOOLS AND ALGORITHMS FOR THE CONSTRUCTION AND ANALYSIS OF SYSTEMS*, 328–342.

Yi, J.; Sadowski, C.; and Flanagan, C. 2009. Sidetrack: generalizing dynamic atomicity analysis. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 1–10.