Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs

Vineet Kahlon and Chao Wang

NEC Laboratories America, Princeton, NJ 08540, USA

Abstract. Triggering errors in concurrent programs is a notoriously difficult task. A key reason for this is the behavioral complexity resulting from the large number of interleavings of operations of different threads. Efficient static techniques, therefore, play a critical role in restricting the set of interleavings that need be explored in greater depth. The goal here is to exploit scheduling constraints imposed by synchronization primitives to determine whether the property at hand can be violated and report schedules that may lead to such a violation. Towards that end, we propose the new notion of a *Universal Causality Graph (UCG)* that given a correctness property P, encodes the set of all (statically) feasible interleavings that may violate P. UCGs provide a unified happens-before model by capturing causality constraints imposed by the property at hand as well as scheduling constraints imposed by synchronization primitives as causality constraints. Embedding all these constraints into one common framework allows us to exploit the synergy between the constraints imposed by different synchronization primitives, as well as between the constraints imposed by the property and the primitives. This often leads to the removal of significantly more redundant interleavings than would otherwise be possible. Importantly, it also guarantees both soundness and completeness of our technique for identifying statically feasible interleavings. As an application, we demonstrate the use of UCGs in enhancing the precision and scalability of predictive analysis in the context of runtime verification of concurrent programs.

1 Introduction

Detecting errors in concurrent programs is a notoriously difficult task. A key reason for this is the behavioral complexity resulting from the large number of interleavings of different threads. This leads to the state-explosion problem which renders a full-fledged state space exploration of concurrent programs infeasible. As a result, in recent years runtime error detection techniques have been gaining in popularity. These come in many variants. Runtime monitoring aims at identifying violations exposed by a given execution trace [12, 18, 5, 9]. However, due to the large number of interleavings of the program, triggering a concurrency bug by exploring just one interleaving is unlikely. In contrast, runtime prediction aims at detecting violations in all feasible interleavings of events of the given trace. In other words, even if no violation exists in that trace, but an alternative interleaving is erroneous, a predictive method [8, 15, 2, 7, 6, 14] may be able to catch it without actually re-running the test.

Predictive analysis seems to offer a good compromise between runtime monitoring and full-fledged model checking in that it guarantees better coverage than runtime monitoring but mitigates the state explosion inherent in model checking. In its most general form, predictive analysis has three steps (1) Run a test of the concurrent program to obtain an execution trace. (2) Run a sound (over-approximate) static analysis of the given trace to detect *potential* violations, e.g., data races, atomicity violations, etc. If no violation is found, return. (3) Build a precise predictive model, and for each potential violation, check whether it is feasible. If it is feasible, find a concrete and replayable witness trace. Many variants of this basic framework have been proposed in the literature to explore the various tradeoffs between scalability and precision. Clearly, the main bottleneck in scalability is the feasibility check in step 3 (essentially model checking).

In the interest of scalability some techniques avoid step 3 altogether. For instance, Farzan et. al. [6] have proposed a static analysis for predicting atomicity violations in which they focus on the control paths and model only nested locks. For threads synchronizing via nested locks only and assuming no data variables, their analysis is sound and complete, in that step 3 can be avoided. However, this technique is not applicable to programs using non-nested locks or synchronization primitives other than locks, including wait/notify, barriers, etc. As a result, the reported violations may be spurious. Although such warnings can serve as hints for subsequent analysis, they are not immediately useful to programmers because deciding whether they are real errors remains a challenging task. Other techniques try to address the scalability problem by exploring only a small subset of the feasible interleavings via trace-based under-approximations [15, 2, 14] thereby suffering from a very limited coverage of interleavings.

If precision is of paramount concern then static analysis (step 2) is augmented with model checking in step 3 wherein the feasibility of the set of statically generated warnings can be verified. Since model checking is computationally expensive, it is imperative that the static analysis be made as precise as possible. First, if static analysis can deduce that a set of warning locations is simply unreachable then the expensive step 3 can be avoided altogether. Second, if static analysis can deduce invariants with respect to the trace and the property at hand, we can use them to weed out many interleavings that need be explored via step 3 thereby enhancing its scalability. Therefore, irrespective of the predictive analysis method being used, step 2, i.e., statically detecting potentially erroneous interleavings of events of the given trace, occupies a key role in determining the scalability and precision of the overall framework.

However, existing static analysis techniques suffer from several drawbacks. **1. Comprehensive Handling of Synchronizations:** State-of-the-art static analysis techniques, e.g. [7], apply only to programs with nested locks and are therefore not applicable to programs using non-nested locks or wait/notify-style primitives in conjunction with locks (nested or non-nested) which are very common in Java programs. The mover-based atomicity checkers, such as *atomizer* [8], are conservative even for control path reachability (no data); they typically can robustly handle locks but not the other synchronization primitives. In contrast, our new static analysis technique can handle multiple synchronization primitives used in real-life programs (e.g. Java) in a unified manner, and is both sound and complete for control path reachability for two threads, i.e., when there is no data, or data does not affect the control flow of the program. **2. Causal Constraints from Properties:** The existence of standard concurrency errors like data races and atomicity violations can be expressed uniformly as a set of happens-before constraints between events of different threads. These property-induced constraints can be used in conjunction with scheduling constraints imposed by synchronization primitives to infer yet more happens-before constraints. To our knowledge, the interaction of these two types of constraints has not been exploited by existing techniques which therefore end up retaining more (spurious) interleavings than are necessary.

The main contribution of this paper is the new notion of a *Universal Causality Graph (UCG)*, which is a unified happens-before model for the given (trace) program as well as the property at hand, that addresses the above challenges. UCGs allow us to capture, as happens-before constraints, the set of all possible interleavings that are feasible under the scheduling constraints imposed by synchronization primitives that may potentially lead to violations of the property at hand. With a given execution trace of a program specified as a set of local computations $x^1, ...x^n$ of n threads and a property P, we associate a UCG, denoted by $U_{(x^1,...,x^n)}(P)$, which is a directed graph whose vertices are a subset of the set of synchronization events occurring along $x^1, ..., x^n$ and each of whose edges $e_1 \rightsquigarrow e_2$, represents a happens-before constraint, i.e., e_1 must be executed before e_2. Thus the UCG implicitly captures the set of all interleavings of $x^1, ..., x^n$ that satisfy all the happens-before constraints represented by its edges.

UCGs have the following desirable properties **(a) Precision:** If data is not tracked, for two threads the UCG captures precisely the set of interleavings of $x^1, \ldots, x^2$ satisfying the property P, e.g., the existence of a data race or atomicity violation. For an arbitrary number of threads the set of interleavings captured is a super-set of the set of interleavings satisfying P. In other words, the analysis is sound in general and complete for two threads interacting via synchronization primitives only. **(b) Unified View:** The UCG encodes both property-induced causality constraints and scheduling constraints imposed by synchronization primitives in terms of happens-before constraints. Unlike existing techniques, it can handle multiple synchronization primitives in a unified manner. This enables us to leverage the synergy between causality constraints induced by both the property as well as the program, and allows us to deduce more causal constraints than would otherwise be possible. More importantly, these constraints are necessary to guarantee both soundness and completeness of our method for two threads. **(c) Scalability:** Since the given trace could be arbitrarily long, incorporating all synchronization events of the trace as vertices and all the deduced causal constraints as edges into the UCG would impact the scalability of the analysis. However, we show that, for predictive analysis of a given property, the UCG need not keep track of causality edges induced by the entire traces $x^1, ..., x^n$ but only short suffixes thereof. The novelty of this *chopping result* lies in the existence of a set of special *lock-free control states*, from which the UCG based analysis can still guarantee both soundness and completeness.

UCGs are a generalization of lock causality graphs (LCGs) [10] which were formulated to reason about pairwise reachability for threads communicating purely via locks. However, LCGs could only be used to reason about programs using locks. Furthermore, LCGs were formulated to reason only about pairwise reachability and therefore could not exploit causality constraints induced by properties such as atomicity violations. Also, our UCG based analysis is a backward inference process starting from the property at hand—it does not enumerate interleavings. This differs from the forward analysis used in [15, 2, 14] which explicitly enumerate thread interleavings.

Happens-before constraints have been exploited before for predictive analysis for detecting races and atomicity violations [4, 9]. However, the causal models considered were restrictive in that the set of interleavings explored had to preserve the global ordering of lock/unlock statements in the original global computation x. Since the number of such interleavings is a small fraction of the total number of feasible interleavings of local computations of different threads along x, these techniques could miss detection of errors and were therefore not guaranteed sound (though they were sound with respect to the global ordering of lock/unlock statements along x). UCGs, on the other hand, allow the lock/unlock statements from different threads to be re-ordered relative to each other and will therefore explore all possible statically feasible interleavings of local computations of different threads along x. This guarantees soundness of our technique.

The proofs of all the results can be found in the full version available on-line [1].

2 Preliminaries

A *concurrent program* has a set $\{T_1, \ldots, T_n\}$ of threads and a set SV of *shared variables*. Each thread T_i, where $1 \leq i \leq n$, has a set of *local variables* LV_i. Let $Tid = \{1, \ldots, n\}$ be the set of thread indices. Let $V_i = SV \cup LV_i$, where $1 \leq i \leq n$, be the set of variables accessible in T_i. An execution trace of the program is a sequence $x = t_1 \ldots t_K$ of events. An *event* $t \in x$ is a tuple $\langle tid, action \rangle$, where $tid \in Tid$ and $action$ is of one of the form (let $tid = i$)

- guarded assignment $(\mathsf{assume}(g), asgn)$, where g is a condition over V_i and $asgn$ is a set of assignments, each of the form $v := exp$, where $v \in V_i$ and exp is an expression over V_i. Intuitively, g must be true for the assignments to proceed.
- $fork(j)$ and $join(j)$. The former models the creation of child thread T_j by thread T_i. The latter models that thread T_i waits for thread T_j to join back.
- $lock(l)$ and $unlock(l)$. The former models the acquire of lock l. The latter models the release of lock l.
- $wait_{pre}(c)$, $wait_{post}(c)$ and $notify(c)$. The first two, when combined, model the wait of condition variable c. The last event models the notification of c.

Each event t in x is a unique execution instance of a statement in the program. If a statement in the program is executed multiple times, e.g., in a loop or a recursive function, each execution instance is modeled as a separate event. If we project x back to the local threads, each current thread $x^1, \ldots, x^n$ is a purely straight-line program.

Synchronization Primitives. In both Java and PThreads, the primitive `wait(c,l)`, where l is a lock and c is a condition variable, is a composite statement. Before calling it, thread T_i is expected to hold lock l. Upon calling it, thread T_i releases lock l, and then blocks—waiting for another thread T_j to call `notify(c)`. After that, and only when lock l is available again, thread T_i wakes up and immediately re-acquires lock l. Therefore, for verification purposes, we model `wait(c,l)` using the semantically equivalent event sequence $wait_{pre}(c)$; $unlock(l)$; $lock(l)$; $wait_{post}(c)$. Also note that the suggested way of using condition variables, in both Java and PThreads, is to wrap both `wait(c,l)` and `notify(c)` with a pair of `lock(l)` and `unlock(l)`.

By defining the expression syntax suitably, the guarded assignment event itself is expressive enough to model the execution of all kinds of statements including synchronization primitives. In fact, this is what we have implemented in the model checking procedure at the final step. The reason why we represent lock-unlock events and wait-notify events separately is for convenience in understanding the UCG-based static analysis; in static analysis, our focus is on these concurrency/synchronization events only (data is ignored). To understand the expressiveness of guarded assignment, consider the following variants: (1) when the guard is `true`, the set $asgn$ models normal assignment statements; (2) when the set $asgn$ is empty, $\mathsf{assume}(g)$ models a branching statement `if(cond)`; and (3) with both the guard and the assignment set, it can model the atomic *check-and-set* operation, which is the foundation of all synchronization primitives. For example, acquire of lock l in thread T_i, where $i \in Tid$, is modeled as event

$\langle i, (\mathsf{assume}(l = 0), \{l := i\})\rangle$; here 0 means the lock is available and thread index i indicates the lock owner. Release of lock l is modeled as $\langle i, (\mathsf{assume}(l = i), \{l := 0\})\rangle$.

Concurrent Trace Programs The semantics of an execution trace $x = e_1 \dots e_K$ is defined using a state transition system. Let V be the set of all program variables and Val be a set of values of variables in V. A *state* is a map $s : V \to Val$ assigning a value to each variable. We also use $s[v]$ and $s[exp]$ to denote the values of $v \in V$ and expression exp in state s. We say that a *state transition* $s \xrightarrow{t} s'$ exists, where s, s' are states and e is a guarded assignment event in thread T_i, if the action is $(\mathsf{assume}(g), asgn)$, $s[g]$ is true, and for each assignment $v := exp$ in $asgn$, $s'[v] = s[exp]$ holds; states s and s' agree on all other variables. The execution trace $x = t_1 \dots t_K$ can be viewed as a total order of the events along x. From x one can derive a partial order called the concurrent trace program (CTP) [17].

Definition 1. *The* concurrent trace program *with respect to x, denoted CTP_x, is a partially ordered set $(T, \sqsubseteq)$ such that, $T = \{t \mid t \in x\}$ is the set of events, and for any $t_i, t_j \in T$, $t_i \sqsubseteq t_j$ iff*

- *$tid(t_i) = tid(t_j)$ and $i < j$; or*
- *t_i has action $fork(tid(t_j))$; or*
- *t_j has action $join(tid(t_i))$; or*
- *t_i has action $wait_{pre}(c)$ and t_j has the matching $notify(c)$; or*
- *t_j has action $wait_{post}(c)$ and t_i has the matching $notify(c)$.*

Intuitively, CTP_x orders events from the same thread by their execution order along x, and orders events from different threads by the causal relations of fork-join and wait-notify. Otherwise, events from different threads are not *explicitly* ordered with respect to each other.

We now define *feasible linearizations* of CTP_x. Let $x' = t'_1 \dots t'_n$ be a linearization of CTP_x, i.e. an interleaving of events of x. We say that x' is *feasible* iff there exist states $s_0, \dots, s_n$ such that, s_0 is the initial state of the program and for all $i = 1, \dots, n$, there exists a transition $s_{i-1} \xrightarrow{t'_i} s_i$. This definition captures the standard sequential consistency semantics for concurrent programs, where we modeled concurrency primitives such as locks by using auxiliary shared variables.

Causal Models for Feasible Linearizations. We recall that in predictive analysis the given concurrent program is first executed to obtain an execution trace x. By projecting x onto the local states of individual threads one can obtain CTP_x. Then given a property P, e.g., existence of data races or atomicity violations, the goal of predictive analysis is to find a feasible linearization of CTP_x that satisfies P.

Since the total number of linearizations of CTP_x may be too large, *static* analysis is often employed to isolate a (small) set of linearizations of CTP_x whose feasibility can then be checked via model checking. Here static feasibility implies that data is typically ignored and the linearizations generated are required to be feasible only under the scheduling constraints imposed by synchronization and fork-join primitives. We propose the notion of a *Universal Causality Graph* that captures precisely the set of feasible interleavings of CTP_x that may lead to violations while guaranteeing (i) soundness in general and completeness for two threads, (ii) scalability, (iii) handling of different synchronization primitives in a unified manner, and (iv) exploiting the synergy between causal constraints imposed by the property as well as the program. To the best of our knowledge, none of the existing techniques satisfies all four of these requirements.

3 Universal Causality Graph

Given a set of local computations $x^1, ..., x^n$ of threads T_1,..., T_n, respectively, and a standard property P such as the presence of a data race or an atomicity violation, we construct a causality graph, denoted $U_{(x^1,...,x^n)}(P)$, such that there exists an interleaving of $x^1, ..., x^n$ satisfying P if and only if $U_{(x^1,...,x^n)}(P)$ is acyclic. We express both the property as well as scheduling constraints imposed by synchronization primitives in terms of happens-before constraints. To start with, we show how to express the occurrence of a property violation as a set of happens-before constraints.

3.1 Properties As Causality Constraints

We consider two standard concurrency violations: data races and atomicity violations.

Data Races. A data race occurs if there exist events t_a and t_b of two different threads such that (a) a common shared variable is accessed by t_a and t_b with at least one of the accesses being a write operation, and (b) there exists a reachable (global) state of the concurrent program in which both t_a and t_b are enabled. In order to express the occurrence of a data race involving t_a and t_b, we introduce the two happens-before constraints $t_{a'} \leadsto t_b$ and $t_{b'} \leadsto t_a$ in the universal causality graph, where $t_{a'}$ and $t_{b'}$ are the events immediately preceding t_a and t_b in their respective threads. Note that given an execution trace, $t_{a'}$ and $t_{b'}$ are defined uniquely.

Atomicity Violations. A three-access atomicity violation [12, 6, 17] involves an event sequence $t_c \ldots t_r \ldots t_{c'}$ such that (a) t_c and $t_{c'}$ are in a user transaction of one thread, and t_r is in another thread, and (b) t_c and t_r are data dependent; and t_r and $t_{c'}$ are data dependent. Depending on whether each event is a *read* or *write*, there are eight possible combinations of the triplet $t_c, t_r, t_{c'}$. While R-R-R, R-R-W, and W-R-R are serializable, the remaining five may indicate atomicity violations. Given the CTP_x and a transaction $trans = t_i \ldots t_j$, where $t_i \ldots t_j$ are events from a thread in x, the set of potential atomicity violations can be computed by scanning the trace x once, and for each remote event $t_r \in CTP_x$, finding the two local events $t_c, t_{c'} \in trans$ such that $\langle t_c, t_r, t_{c'} \rangle$ forms a non-serializable pattern. Such an atomicity violation can be captured via the two happens-before constraints $t_c \leadsto t_r$ and $t_r \leadsto t_{c'}$.

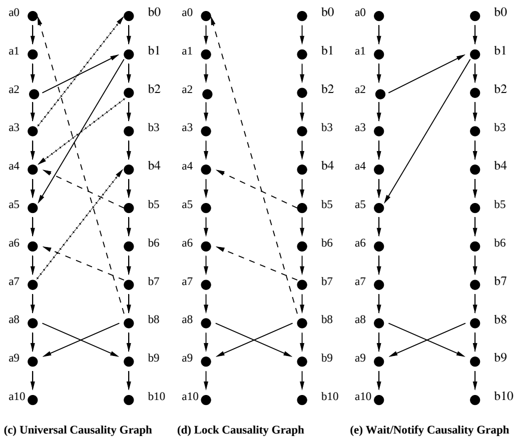
3.2 Universal Causality Graph Construction

We motivate the concept of a *Universal Causality Graph (UCG)* via an example comprised of local traces x^1 and x^2 of threads T_1 and T_2, respectively, shown in fig 1. Suppose that we are interested in deciding whether *a9* and *b9* constitute a data race. Since the set of locks held at *a9* and *b9* are disjoint, these pair of locations constitute a potential data race. Furthermore, since the traces use wait/notify statements as well as non-nested locks, we cannot use existing techniques [7, 6, 11] to decide simultaneous reachability of *a9* and *b9*. As discussed in Sec. 3.1, for the race to occur there must exist an interleaving of x^1 and x^2 that satisfies the constraints $a8 \leadsto b9$ and $b8 \leadsto a9$. Furthermore, the locks along x^1, x^2 must be acquired in a consistent fashion and causality constraints imposed by wait/notify events must be respected.

We now show that the causality constraints generated in the UCG by the property as well as scheduling constraints imposed by locks, fork/join, and wait/notify events that are *relevant* in exposing the data race, can be captured in a unified manner. For two

```
a0:  lock($l_3$);              b0:  lock($l_1$);
a1:  lock($l_1$);              b1:  notify($c$);
a2:  wait$_{pre}$(c)           b2:  unlock($l_1$);
a3:  unlock($l_1$)             b3:  lock($l_1$);
a4:  lock($l_1$)               b4:  lock($l_3$);
a5:  wait$_{post}$(c);         b5:  unlock($l_1$);
a6:  lock($l_2$);              b6:  lock($l_2$);
a7:  unlock($l_3$);            b7:  unlock($l_2$);
a8:  unlock($l_1$);            b8:  unlock($l_3$);
a9:  $sh = sh + 1$;            b9:  $sh = sh + 2$;
a10:  unlock($l_2$);           b10:  ...

$T_1$                          $T_2$
```



(c) Universal Causality Graph (d) Lock Causality Graph (e) Wait/Notify Causality Graph

Fig. 1. An Example Universal Causality Graph

arbitrary events c_1 and c_2 of $U_{(x^1,x^2)}(P)$, there exists an edge $c_1 \rightsquigarrow c_2$ if c_1 must be executed before c_2 in order for P to hold.

A UCG has two types of edges (i) *Seed* edges and (ii) *Induced* edges. Seed edges, shown as bold edges in the UCG in Fig. 1(c), can be further classified as *Property* and *Synchronization* seed edges.

Property Seed Edges are introduced by properties as in Sec. 3.1. In our example, the potential data race at *a9* and *b9* introduces the edges $a8 \rightsquigarrow b9$ and $b8 \rightsquigarrow a9$.

Synchronization Seed Edges are induced by fork/join and the various synchronization primitives. For simplicity, we only discuss wait/notify. Locks do not add seed edges.

– **Wait-Notify:** Recall that the primitive $wait(c, l_1)$ is modeled as the event sequence $a1 : lock(l_1)$, $a2 : wait_{pre}(c)$, $a3 : unlock(l_1)$, $a4 : lock(l_1)$, $a5 : wait_{post}(c)$. Let $b1$ be the matching $notify(c)$. The seed edges are $a2 \rightsquigarrow b1$ and $b1 \rightsquigarrow a5$. Note that since x^1 and x^2 are generated from concrete traces for each notify statement there exists a unique matching wait statement, if any, and vice versa. Strictly speaking, we should consider all scenarios wherein notify statements are executed without the matching wait statements. However, that will block the wait statements causing us to miss potential violations. In order to maximize the number of violations, we assume that all pairs of matching wait/notify statements are executed in unison.

– **Fork-Join:** The first edge is from *fork* to the first event of the child thread. The second edge is from the last event of the child thread to *join*.

The interaction between these seed edges and locks can be used to deduce more constraints that are captured as *induced* edges. They are the dashed edges in Fig. 1(c). These induced edges are key in guaranteeing soundness and completeness.

Induced Edges: Consider the seed causality constraint *b8* $\rightsquigarrow$ *a9*. From this we can deduce the new causality constraint $b7 \rightsquigarrow a6$. Towards that end, we observe that at location *a9*, lock l_2 is held which was acquired at *a6*. Also, once l_2 is acquired at *a6* it is not released until after T_1 exits *a10*. Furthermore, we observe that *b6* is the last

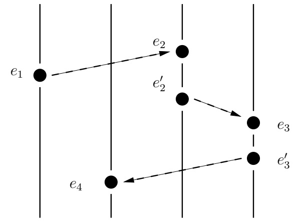


Fig. 2. A Causality Chain from e_1 to e_4

statement to acquire l_2 before *b8* and *b7* is its matching release. Due to constraint *b8* $\rightsquigarrow$ *a9* and the local constraint *b7* $\rightsquigarrow$ *b8*, one can deduce, via transitivity, that *b7* $\rightsquigarrow$ *a9*. Moreover, from the mutual exclusion semantics of lock l_2, we have that since l_2 is held at *a9* it must first be released by T_2 before T_1 can acquire it via *a6* without which $a9$ cannot be executed. Thus *a6* must also be executed after *b7*.

From $b7 \rightsquigarrow a6$ one can, in turn, deduce that $b8 \rightsquigarrow a0$. This is because the last statement to acquire l_3 before $b7$ is *b4* and its matching release is *b8*. Using the same argument as above, from the causality constraint *b7* $\rightsquigarrow$ *a6* and mutual exclusion semantics of lock l_3, we can deduce that l_3, which is held at b7, must first be released by T_2 before T_1 can acquire it via *a0* which it needs to in order to execute *a6*, i.e., $b8 \rightsquigarrow a0$. The process is continued till a fixpoint is reached. Fig. 1(b) shows all the induced edges added by starting at the seed edges *b8* $\rightsquigarrow$ *a9* and *a8* $\rightsquigarrow$ *b9*.

Similarly it can be seen that the wait/notify seed edges $a2 \rightsquigarrow b1$ and $b1 \rightsquigarrow a5$ add further induced edges which are not shown for reasons of clarity.

3.3 Computing the Universal Causality Graph

The procedure to compute the UCG is formulated as alg. 1. It adds causality constraints one-by-one (seed edges via steps 3-7, and induced edges via steps 8-19) till it reaches a fixpoint. Note that steps 20-22, preserve the local causality constraints along $x^1, ..., x^n$.

Since each edge in $U_{(x^1,x^2)}(P)$ is a *happens-before* constraint, we see that in order for P to hold $U_{(x^1,x^2)}(P)$ has to be acyclic. In fact, it turns out that for two threads acyclicity is also a sufficient condition leading to the following *Acyclicity Result.*

Theorem 1. *Property P is violated via a (statically) feasible interleaving of local paths x^1 and x^2 of T_1 and T_2, respectively, if and only if $U_{(x^1,x^2)}(P)$ is acyclic.*

3.4 Generalization to n Threads

For the case of n threads, the only modification that is required to alg. 1 is in step 10. Here a causality relation between events d_i and d_j can be induced not only via a single edge of the form $d_j \rightsquigarrow d_i$ but also via a *causality chain* from d_j to d_i (see fig. 2), i.e., a sequence of pre-existing causality edges of the form $e_1 \rightsquigarrow e_2$, $e_2' \rightsquigarrow e_3$, $e_3' \rightsquigarrow e_4$,..., $e_{k-1}' \rightsquigarrow e_k$, where (i) $e_1 = d_j$ and $e_k = d_i$, and (ii) for each m, e_m' occurs after e_m along $x^{m'}$ for some $m' \in [1..n]$. Thus the condition at line 10 of alg. 1 is modified as follows:

for each pair of events d_i and d_j belonging to different threads T_i and T_j, respectively, such that there is a causality chain from d_j to d_i **do**

Algorithm 1 Computing the Universal Causality Graph

```
Input: Property $P$ and local paths $x^1$ and $x^2$ of $T_1$ and $T_2$, respectively.
Initialize the vertices and edges of $U_{(x^1,x^2)}(P)$ to $\emptyset$
Add causality edges for $P$ as defined in sec. 3.1    (Property Seed Edge)
Add fork-join induced causality edges    (Fork-Join Seed Edge)
for each pair of locations $w$ and $n$ corresponding to matching wait/notify events do
    Add edges $w_{pre} \rightsquigarrow n$ and $n \rightsquigarrow w_{post}$,    (Wait/Notify Seed Edge)
end for
repeat
    for each lock $l$ do
        for each edge $d_j \rightsquigarrow d_i$ between events $d_j$ and $d_i$ of $T_j$ and $T_i$, respectively do
            If $l$ is held at $d_j$ and not released after $d_j$ along $x^j$ then add an edge $r_i \rightsquigarrow a_j$, where $a_j$ is the last statement to acquire $l$ before $d_j$ and $r_i$ is the last statement to release $l$ before $d_i$
            If $l$ is held at $d_j$ and not released after $d_j$ along $x^j$ and $l$ is held at $d_i$ then output $P$ does not hold and Quit
            Let $a_j$ be the last statement to acquire $l$ before $d_j$ along $x^j$ and $r_j$ the matching release for $a_j$; and let $r_i$ be the first statement to release $l$ after $d_i$ along $x^i$ and $a_i$ the matching acquire for $r_i$
            if $l$ is held at either $d_i$ or $d_j$ then
                add edge $r_j \rightsquigarrow a_i$            (Induced Edge)
            end if
        end for
    end for
until no new edges can be added
for $i \in [1..2]$ do
    Add edges among all events of $x^i$ occurring in $U_{(x^1,x^2)}(P)$ to preserve their relative ordering along $x^i$
end for
```

Complexity of the UCG Construction. The total time taken for building the UCG is $O(|E||L|)$, where $|E|$ denotes the number of edges that can be added to the UCG and $|L|$ is the number of different locks acquired/released along $x^1, ..., x^n$. In the worst case $|E|$ is $O((n|N|)^2)$, where $|N|$ is the maximum number of synchronization events occurring along any of the local sequences $x^1, ..., x^n$.

Exploiting the Synergy between Synchronization Primitives Existing static techniques for reasoning about programs with multiple synchronization primitives like locks and wait/notify consider the scheduling constraints imposed by them separately. Thus a violation is said to exist if it can occur either under scheduling constraints imposed by locks or under those imposed by wait/notifies. However, UCGs capture constraints imposed by all the primitives in a unified manner thereby allowing us to exploit the synergy between them. In our example, by considering constraints imposed by locks and wait/notifies separately we cannot deduce that *a9* and *b9* do not race. Indeed, the scheduling constraints imposed only by locks results in the *acyclic* lock causality graph Fig. 1(d). Similarly, the scheduling constraints imposed only by wait/notify results in the *acyclic* wait/notify causality graph in Fig. 1 (e). In order generate a cycle that proves infeasibility of the data race we have to consider both the primitives in unison. Towards that end, we construct the UCG shown in Fig. 1(c) which has the cycle

$a0 \rightsquigarrow a2 \rightsquigarrow b1 \rightsquigarrow b8 \rightsquigarrow a0$ thereby allowing us to deduce that *a9* and *b9* do not constitute a data race.

Exploiting the Synergy between Program and Property Consider the cycle $a0 \rightsquigarrow a2 \rightsquigarrow b1 \rightsquigarrow b8 \rightsquigarrow a0$ in Fig. 1(c). It is comprised of the induced edge $b8 \rightsquigarrow a0$ and the wait/notify seed edge $a2 \rightsquigarrow b1$. The induced edge $b8 \rightsquigarrow a0$ was added via an induction sequence (via steps 8-18) starting at the property seed edge $b8 \rightsquigarrow a9$. Thus in order to rule out the data race we have to consider the causality constraints induced by the property as well as the synchronization primitives in unison. In contrast, existing techniques do not exploit the synergy between these constraints and are therefore not guaranteed complete for two threads.

3.5 Handling Multiple Properties

For clarity, the UCG construction above was formulated for a single property. However, a given trace might generate many potential warnings for concurrency bugs and building the UCG from scratch for each warning would be infeasible in practice. In order to build a single UCG for all the warnings, we start by adding the seed edges for all the warnings. The seed edges for a given property are now labeled with an id that is unique to that property. If the same seed edge needs to be added for multiple properties then it is labeled with the set of ids of all these properties. During the UCG construction via alg. 1 when an edge e induces another edge f, then the property-ids are propagated from e to f by relabeling f with the union of ids of e and f's pre-existing ids. In this way each edge in the UCG may be labeled with multiple property-ids. It is easy to see that an edge will occur in the UCG of a property P if and only if it is labeled with P's id (and possibly other ids).

The resulting UCG U contains edges that are induced by all the properties. If we are interested in checking whether a given property P holds, then we can extract from U the UCG induced by P by simply projecting on to the edges that are labeled with P's id. Then P is satisfied if and only if the resulting sub-graph is acyclic.

The above technique of propagating the ids of properties starting from the seed edges allows us to build the UCG only once while the validity of the different properties can be checked separately by projecting onto the appropriate sub-graphs. Note that since the wait/notify seed edges occur in the UCGs for all the properties, they are labeled with ids of all the properties.

4 Chopping Result: Scaling UCG Construction

In order to leverage the UCG for a practically feasible analysis we have to address the key issue that the number of constraints added to the UCG may be too large. This is because (1) the traces x^1 and x^2 could be arbitrarily long, and (2) wait/notify events could be many and could span the entire lengths of these traces. Thus a very large number of wait/notify seed edges, and, as a result, induced edges, could be added along the entire lengths of x^1 and x^2. It contrast (see fig. 3), when constructing the lock causality graph (LCG) as in [10] for reasoning about threads interacting only via locks, causality edges are added only between lock/unlock statements occurring along the suffixes of x^1 and x^2 starting at their last lock-free states. In practice, these suffixes of x^1 and x^2 are short, as for performance reasons programmers tend to keep the lengths of critical sections small. This ensures that the LCG size is small thereby ensuring scalability.

Decomposition Result. In order to guarantee scalability of the UCG construction in the presence of both wait/notifies and locks, our goal, analogous to LCGs in [10], is to restrict the UCG construction to only small suffixes of $x^1, ..., x^n$. Towards that end, we start with the following key *decomposition result* which provides useful insight into the structure of UCGs. Intuitively, the decomposition result states that the given paths $x^1, ..., x^n$ can be broken down into an equal number, say m, of segments, with $x^j = x^{j1}...x^{jm}$ such that in order to check the acyclicity of $U_{(x^1,...,x^n)}(P)$ it suffices to check the acyclicity of each of the m smaller UCGs $U_{(x^{1i},...,x^{ni})}(P)$.

Theorem 2. (Partitioning Result). *Given finite local computations $x^1, ..., x^n$ of $T_1, ..., T_n$ respectively, for each j, let $x^j = x^{j1}x^{j2}$ be a partition of x^j such that*

- *the last state occurring along x^{j1} is lock-free, and*
- *for $j \neq k$, there does not exist a wait/notify seed edge, a fork-join seed edge or a property seed edge with endpoints along x^{j1} and x^{k2} or along x^{k1} and x^{j2}.*

Then $U_{(x^1,...,x^n)}(P)$ is acyclic if and only if $U_{(x^{11},...,x^{n1})}(P)$ and $U_{(x^{12},...,x^{n2})}(P)$ are acyclic.

Proof.

For ease of notation, we consider the two thread case. Also, we use $barrier^i$ to denote the last local state of T_i occurring along x^{i1}. By condition 1 of the hypothesis $barrier^i$ is lock-free.

($\Rightarrow$) Since the set of edges of $U_{(x^{11},x^{21})}(P) \cup U_{(x^{12},x^{22})}(P)$ is a subset of the set of edges of $U_{(x^1,x^2)}(P)$, this direction is trivial.

($\Leftarrow$) We start with two claims:

Claim 1: Any edge of $U_{(x^1,x^2)}(P)$ that is not in $U_{(x^{11},x^{21})}(P) \cup U_{(x^{12},x^{22})}(P)$ is of the form $a \rightsquigarrow b$, where, for some $j, k \in [1..2]$, a lies along x^{j1} and b lies along x^{k2}. In other words, edges of $U_{(x^1,x^2)}(P)$ that are not in $U_{(x^{11},x^{21})}(P) \cup U_{(x^{12},x^{22})}(P)$ can only go 'forward'.

To prove this, we start by noting that if $a \rightsquigarrow b$ is an edge where a lies along x^{j2} and b along x^{k1} for some $j \neq k$, then $a \rightsquigarrow b$ cannot be a seed edge (condition 2 in the hypothesis). Without loss of generality, we can assume that $a \rightsquigarrow b$ is the first edge added in the iterative computation of alg 1 that satisfies the above condition. Suppose that $a \rightsquigarrow b$ is induced by the edge $c \rightsquigarrow d$. Note that each induced edge of a UCG goes from a lock release to a lock acquisition statement for the same lock. Let a and b be statements releasing and acquiring lock l, respectively.

There are two cases to consider. First assume that $a \rightsquigarrow b$ was added via step 11 of alg. 1. In that case b would be an acquisition statement for a lock l that is never released after b along x^k. Since, by our assumption, b lies along x^{k1}, i.e., before $barrier^k$, we have that lock l would be held at $barrier^k$, thereby contradicting hypothesis 1.

Thus $a \rightsquigarrow b$ has to be induced via step 16 of alg. 1. There are two sub-cases two consider. Since $a \rightsquigarrow b$ is the first edge added in the iterative computation of alg 1 such that a lies along x^{j2} and b along x^{k1} for some $j \neq k$ the edge $c \rightsquigarrow d$ inducing $a \rightsquigarrow b$ is such that either c and d lie along x^{j1} and x^{k1}, respectively, or c and d lie along x^{j2} and x^{k2}, respectively

First assume that c and d lie along x^{j1} and x^{k1}, respectively. This implies that c lies before $barrier^j$ along x^j. Since $a \rightsquigarrow b$ is added via step 16 of alg. 1 we see that a has to be the matching release of the last acquisition of l before c. Thus l is acquired before

c and hence before $barrier^j$ and released at a, i.e., after $barrier^j$. Thus $barrier^j$ is not lock-free contradicting condition 1 of the hypothesis.

The other case where c and d lie along x^{j2} and x^{k2}, respectively, can be handled in a similar fashion. This proves claim 1.

Claim 2: No 'local' edge is induced by a 'cross' edge, i.e., no edge of the form $a \rightsquigarrow b$, where a and b lie along x^{jm} and x^{km}, respectively, is induced by an edge of the form $c \rightsquigarrow d$, where c and d lie along x^{j1} and x^{k2}, respectively.

To see that first consider the case when $a \rightsquigarrow b$ is induced by step 16. If $a \rightsquigarrow b$ is such that a and b lie along x^{j2} and x^{k2}, respectively, then we can see that a would be a lock release statement matching a lock acquisition for a lock held at c. Thus l would be held at all states occurring between c and a. Thus l would also be held at $barrier^j$. If $a \rightsquigarrow b$ is such that a and b lie along x^{j1} and x^{k1}, respectively, then we can similarly show that l would be held at $barrier^k$.

The case where $a \rightsquigarrow b$ in induced by step 11, is not possible. Indeed, then there would exist a lock l held at c that is never released after c along x^j, i.e., l would be held at $barrier^j$, contradicting condition 1 of the hypothesis.

Finally, to prove the main result, let $cyc : a_1 \rightsquigarrow b_2 \rightsquigarrow c_2 \rightsquigarrow d_1 \rightsquigarrow a_1$ be a cycle in U that does not exist in $U_{(x^{11},x^{21})}(P) \cup U_{(x^{12},x^{22})}(P)$. Since cyc does not exist in $U_{(x^{11},x^{21})}(P) \cup U_{(x^{12},x^{22})}(P)$, at least one of the edges along cyc, say $a_1 \rightsquigarrow b_2$, must not belong to $U_{(x^{11},x^{21})}(P) \cup U_{(x^{12},x^{22})}(P)$. By claims 1 and 2, $a_1 \rightsquigarrow b_2$ has to be a forward cross edge, i.e., a_1 and b_2 lie along x^{j1} and x^{k2}, respectively, where $j \neq k$. Thus a_1 lies before $barrier^j$ along x^j and b_2 lies after $barrier^k$ along x^k. However, since c_2 lies after b_2 along x^k it also lies after $barrier^k$ along x^k. Similarly, since d_1 lies before a_1 along x^j it also lies before $barrier^j$ along x^j. However, in that case $c_2 \rightsquigarrow d_1$ would be a *backward* edge thereby violating claim 1 above. This concludes the proof. □

Repeated application of the above result leads to the following partitioning result.

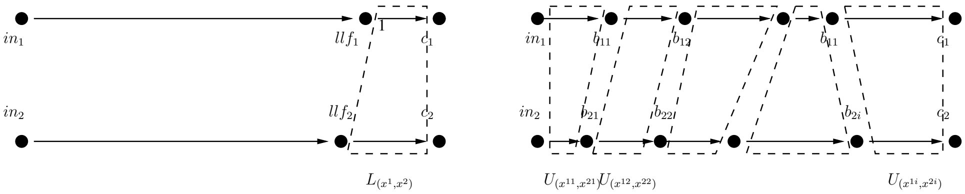
Corollary 3. (Decomposition Result). *Given finite local computations $x^1, ..., x^n$ of threads $T_1, ..., T_n$, respectively, for each j, let $x^j = x^{j1}...x^{jm}$ be a partition of x^j such that*

- *the last states occurring along x^{ji}, where $i \in [1..m]$, are lock-free, and*
- *there does not exist a wait/notify seed edge, a property seed edge or a fork-join seed edge with endpoints in x^{ji} and $x^{ki'}$, where $j \neq k$ and $i \neq i'$.*

Then $U_{(x^1,...,x^n)}(P)$ is acyclic if and only if for each $i \in [1..m]$, $U_{(x^{1i},...,x^{ni})(P)}$ is acyclic.

The situation is illustrated in fig. 3. The lock causality graph, shown in 3(a), is generated only by the suffixes of x^1 and x^2 starting with the last lock free states llf_1 and llf_2 along x^1 and x^2, respectively. However the UCG for x^1 and x^2 is comprised of the UCGs $U_{(x^{1i},x^{2i})}(P)$ (and some more edges which don't impact its acyclicity) generated by the pairs of segments x^{1i} and x^{2i} delineated, respectively, by the *causality barriers* b_{1i} and $b_{1(i+1)}$, and b_{2i} and $b_{2(i+1)}$, where a causality barrier is as defined below:

Definition (Causality Barrier). *Given finite local computations $x^1, ..., x^n$ of threads $T_1, ..., T_n$, respectively, where $x^i = x^i_0...x^i_{n_i}$, we say that the n-tuple $(x^1_{b_1}, ..., x^n_{b_n})$, with $x^i_{b_i}$ being a local state of T_i, forms a causality barrier if (1) for each i, $x^i_{b_i}$ is lock-free,*



(a) Lock Causality Graph

(b) Universal Causality Graph

Fig. 3. Universal Causality Graph Decomposition

i.e., no lock is held by T_i at $x^i_{b_i}$, and (2) there does not exist a seed edge $(x^j_m, x^k_{m'})$, where $j \neq k$, $m \in [0..b_j]$ and $m' \in [b_k + 1, n_k]$ or $m \in [0..b_k]$ and $m' \in [b_j + 1, n_j]$.

Intuitively, seed edges along the traces x^1 and x^2 gives rise to *localized* universal causality graphs that are separated by causality barriers.

Chopping Result for Predictive Analysis. In predictive analysis, we start from a global execution trace x of the program. Our goal is to decide whether there exists a different valid interleaving of the local computations x^1 and x^2 of T_1 and T_2 along x, that may uncover an error. If we were given two arbitrary local computations y^1 and y^2 of threads T_1 and T_2 then in order to decide whether there exists an interleaving of y^1 and y^2 leading to an error state, we would have to build the complete UCG along the entire lengths of y^1 and y^2. However, by exploiting the fact in predictive analysis, x^is are projections of a valid global computation x onto the local states of individual threads, we now show that we need not build the entire UCG $U_{(x^1,x^2)}(P)$ but only the one generated by suffixes x^{1b} and x^{2b} of x^1 and x^2, respectively, starting at a last barrier pair along x^1 and x^2. This ensures scalability of our analysis as we can, in practice, ignore most synchronization primitives except for the last few. We say that the n-tuple $(x^1_{b_1}, ..., x^n_{b_n})$ of local states of threads $T_1, ..., T_n$ is a *last causality barrier* along $x^1, ..., x^n$ if there does not exist another causality barrier $(x^1_{b'_1}, ..., x^n_{b'_n})$ such that for each i, $x^i_{b'_i}$ occurs after $x^i_{b_i}$ along x^i and all property seed edges are of the form $a \rightsquigarrow b$, where, for some i, j, a and b occur after $x^i_{b_i}$ and $x^j_{b_j}$ along x^i and x^j, respectively. Then

Theorem 4. (Chopping Result). *Let $x^1, ..., x^n$ be local computations of threads $T_1, ...,$ T_n, respectively, along a valid global computation x of the given concurrent program. Let $U_{(x^{1b},...,x^{nb})}(P)$ be the UCG generated by the suffixes $x^{1b}, ..., x^{nb}$ of $x^1, ..., x^n$, respectively, beginning with a last causality barrier $(x^1_{b_1}, ..., x^n_{b_n})$ along $x^1, ..., x^n$. Then property P is violated via a statically feasible interleaving of $x^1, ..., x^n$ if and only if $U_{(x^{1b},...,x^{nb})}$ is acyclic.*

Proof.

Let $x^{1p}, ..., x^{np}$ be the prefixes of $x^1, ..., x^n$ starting from the initial local states of $T_1, ..., T_n$ along $x^1, ..., x^n$ and leading to the last causality barrier states $x^1_{b_1}, ..., x^n_{b_n}$, respectively. Let U_{pre} be the UCG generated by $x^{1p}, ..., x^{np}$. Since a last causality barrier is also a causality barrier, by theorem 2 we have that $U_{(x^1,...,x^n)}(P)$ is acyclic if and only if both U_{pre} and $U_{(x^{1b},...,x^{nb})}(P)$ are acyclic.

($\Rightarrow$) Let y be a feasible interleavings of $x^1, ..., x^n$ violating P. Then we have that $U_{(x^1,...,x^n)}(P)$ is acyclic which by the above observation implies that $U_{(x^{1b},...,x^{nb})}(P)$ is acyclic,

($\Leftarrow$) Let $U_{(x^{1b},...,x^{nb})}$ be acyclic.

In order to show that these exists a feasible interleaving y of $x^1, ..., x^n$ violating P we have to show that $U_{(x^1,...,x^n)}(P)$ is acyclic which by the above observation reduces to showing that U_{pre} is acyclic.

Towards that end, we exploit the fact that x is a valid computation, i.e., a valid interleaving of $x^1, ..., x^n$ leading to the final states occurring along $x^1, ..., x^n$. Note that x need not satisfy P, i.e., the causality constraints induced by the property seed edges need not be satisfied. However, from the fact that x is a feasible interleaving of $x^1, ..., x^n$,, we have that all scheduling constraints imposed by the wait/notify seed edges as well as the causality constraints induced from the wait/notify seed edges are satisfied. Thus if we consider the UCG U' computed by considering the wait/notify seed edges as the only seed edges, then the fact that x is a feasible interleaving implies that U' is acyclic.

Note that if we remove the property seed edges then the tuple $(x^1_{b_1}, ..., x^n_{b_n})$ remains a causality barrier for $x^1, ..., x^n$. Let U'_{pre} be the UCG generated by the prefixes $x^{1p}, ..., x^{np}$ by ignoring the property seed edges. Similarly, let U'_{suf} be the UCG generated by the suffixes $x^{1b}, ..., x^{nb}$ of $x^1, ..., x^n$ by ignoring the property seed edges. Then, by theorem 2, U' is acyclic if and only if U'_{pre} and U'_{suf} are acyclic. Thus U'_{pre} and U'_{suf} are both acyclic. However, from the fact that $(x^1_{b_1}, ..., x^n_{b_n})$ is a last causality barrier, we have that the vertices of all property seed edges lie along the suffixes $x^{1b}, ..., x^{nb}$. Thus U'_{pre} had no property seed edges to start with and so the set of seed edges of U'_{pre} and U_{pre} are the same. This implies that the set of induced edges of U_{pre} and U'_{pre} are also the same. Thus $U'_{pre} = U_{pre}$. Since U'_{pre} is acyclic, we have that U_{pre} is also acyclic. The concludes our proof. □

Computing a Last Casualty Barrier. As the final step, we present a procedure (alg. 2) to identify a last causality barrier. For simplicity, we handle only the two thread case. Let c_1 and c_2 be the last local states along x^1 and x^2, respectively. From c_1 we traverse backwards along x^1 till we reach the last lock free state lf_1 along x^1 before c_1. Note that all the wait/notify events occurring between lf_1 and c_1 along x^1, denoted by WN_1, must be matched along the suffix beginning with $x^2_{b_2}$. Therefore from c_2, we have to traverse backward till we encounter the first lock-free state lf_2 such that all events in WN_1 are matched along the suffix of x^2 starting at lf_2. However, in traversing backward from c_2 to lf_2 we may encounter wait/notify events, denoted by the set WN_2, that are not matched along the suffix of x^1 starting at lf_1. In that case, we need to traverse further backwards starting at lf_1 till we encounter a lock-free state lf'_1 such that all events in WN_2 are matched along the suffix of x^1 starting at lf'_1. If we do not encounter any new wait/notify event that is unmatched along the suffix of x^2 starting at lf_2 then we have reached a fixpoint. Else if there exist wait/notify events occurring along the suffix of x^1 starting at lf'_1 that are unmatched along the suffix of x^2 starting at lf_2 then the whole procedure is repeated till a fixpoint is reached.

5 Experiments

We have implemented the proposed algorithm in a tool called *Fusion* [17]. Our tool is capable of analyzing execution traces generated by both Java programs and multi-threaded C programs using *PThreads*. For C programs, we use CIL [13] to instrument the source code to create executables that can log execution traces at runtime. For Java

Algorithm 2 Computing a Last Causality Barrier

```
Input: A pair of local paths $x^1$ and $x^2$ leading to local states $c_1$ and $c_2$ of threads $T_1$ and $T_2$, respectively.
Let $lf_1$ be the last lock-free state before $c_1$ along $x^1$ such all (start or end) vertices of property edges occur after $lf_1$ along $x^1$ and let $WN_1$ be the set of wait/notify events encountered along the segment $x^1_{[lf_1,c_1]}$, i.e., between local states $lf_1$ and $c_1$ along $x^1$
Set terminate to false and $lf_2$ to $c_2$
while terminate equals false do
    Let $lf'_2$ be the last lock-free state before $lf_2$ along $x^2$ such that each wait/notify event in $WN_1$ is matched by an event along the segment $x^2_{[lf'_2,c_2]}$ and all (start or end) vertices of property edges occur after $lf_2$ along $x^2$. Let $WN_2$ be the set of wait/notify events encountered along $x^2_{[lf'_2,lf_2]}$
    Let $lf'_1$ be the last lock-free state at or before $lf_1$ along $x^1$ such that each wait/notify event in $WN_2$ is matched by an event along the segment $x^1_{[lf'_1,c_1]}$. Let $WN'_1$ be the set of wait/notify events encountered along $x^1_{[lf'_1,lf_1]}$
    if $lf'_1$ equals $lf_1$ then
        Set $terminated = true$ and output ( $lf_1$, $lf'_2$) as a last causality barrier
    else
        Set $WN_1 = WN'_1$, $lf_1 = lf'_1$ and $lf_2 = lf'_2$
    end if
end while
```

programs, we use execution traces logged at runtime by a modified version of the Java PathFinder. The Java traces used in our experiments were kindly provided to us by Mahmoud Said. Our experiments were conducted on a PC with 1.6 GHz Intel processor and 2GB memory running Fedora 8.

The overall predictive analysis is as follows. We first find the potential errors (warnings) by a simple static analysis; these are event pairs for data races and event triplets for atomicity violations. Then we apply the UCG analysis to prune away as many spurious warnings as we can. Finally, we use a SMT-based procedure (as in [17]) to check the remaining UCG warnings. This final step uses the *Yices* SMT solver [3]. For each reported error, the SMT-based procedure also returns a witness trace. The UCG warnings are checked one by one in the SMT-based procedure, but we use the incremental feature of the SMT solver to share the cost of checking different warnings. We also add the induced constraints of the UCG to the SAT solver to help speed up the search.

Table 1 shows the results of predicting data races and three-access atomicity violations [17] in traces of Java and C programs. All benchmarks are public domain[1]. The first two columns show the name and the number of threads. The next five columns show the statistics of the trace, including the number of events, the number of lock events, the number of wait-notify events, the number of lock variables, and the number of condition variables. The next six columns show the results of predicting data races using both static analysis and model checking. The first four columns are the total number of warnings, the number of warnings after a lock based analysis alone (lsa), the number of warnings after a fork/join/wait/notify analysis alone (mhb), and the number of warnings after the combined UCG analysis (ucg). The next two columns shows the results of

[1] The traces are available at http://www.nec-labs.com/∼chaowang/pubDOC/LnW.tar.gz

Test Program		Given Trace					Predicting Data Races						Predicting Atomicity Violations					
		# events			# vars		static ana. (warnings)				witness gen		static ana. (warnings)				witness gen	
name	thrd	total	lk	wn	lk	wn	total	lsa	mhb	ucg	wits	time(s)	total	lsa	mhb	ucg	wits	time(s)
ex.race	3	29	4	0	2	0	8	8	2	**2**	1	0.1	2	2	0	**0**	0	0.0
ex.norace	3	37	8	0	2	0	8	6	2	**0**	0	0.0	2	2	0	**0**	0	0.0
ra.Main	3	55	12	5	3	4	13	13	4	**4**	1	0.1	2	2	0	**0**	0	0.0
connectionpool	4	97	16	5	1	3	89	21	28	**0**	0	0.0	30	6	4	**0**	0	0.0
liveness.BugG	7	285	39	6	9	6	408	138	280	**10**	0	0.4	280	60	220	**0**	0	0.0
s1.JGFBarrier	10	649	62	21	2	7	1831	488	1214	**30**	0	1.4	852	102	612	**3**	0	1.6
s1.JGFBarrier	13	799	77	28	2	7	2952	656	2077	**49**	0	3.6	950	87	709	**9**	0	3.7
account.Main	11	902	146	12	21	10	372	342	186	**162**	20	4.0	140	140	60	**60**	2	1.3
philo.Philo	6	1141	126	41	6	22	1719	566	576	**0**	0	0.0	413	81	177	**0**	0	0.0
s1.JGFSyncB	16	1510	237	0	2	0	21230	1142	17415	**117**	0	800	13578	186	11532	**0**	0	0.0
account.Main	21	1747	282	20	41	20	740	680	420	**360**	80	54.5	280	280	120	**120**	3	5.9
elevator.E	4	3000	368	0	11	0	1293	1276	17	**0**	0	0.0	6	4	2	**0**	0	0.0
elevator.E	4	4998	587	0	11	0	3178	3128	50	**0**	0	0.0	12	8	4	**0**	0	0.0
elevator.E	4	8000	1126	0	11	0	3553	3458	95	**0**	0	0.0	18	12	6	**0**	0	0.0
tsp.Tsp	4	45653	20	5	5	3	113	113	4	**4**	3	0.1	0	0	0	**0**	0	0.0
atom001	3	88	6	0	1	0	8	5	3	**0**	0	0.0	4	4	2	**2**	1	0.1
atom001a	3	100	8	1	1	1	12	8	4	**0**	0	0.0	4	4	2	**2**	0	0.1
atom002	3	462	124	0	2	0	96	45	51	**0**	0	0.0	68	68	34	**34**	33	36.7
atom002a	3	462	126	3	2	1	101	49	52	**0**	0	0.0	68	68	34	**34**	0	32.0
banking-av	3	748	20	0	3	0	284	284	72	**72**	72	3.1	64	64	32	**32**	32	1.5
banking-sav	3	852	28	2	3	2	333	325	80	**72**	24	5.2	64	64	32	**32**	16	3.4
banking-noav2	3	856	32	2	3	2	337	305	80	**48**	0	1.3	64	48	32	**16**	0	1.2

Table 1. Predicting data races and atomicity violations in traces of Java/C programs.

model checking the UCG warnings, including the number of witnesses generated and the model checking time in seconds.

The last six columns of Table 1 show the results of predicting three-access atomicity violations. The data format is the same as predicting data races, except that the warnings are now potential atomicity violations. Note that in order to predict atomicity violations, the user transactions (which are intended to be atomic) need to be marked explicitly in the traces. For Java traces, we have assumed that all the synchronized blocks are intended to be atomic, unless the synchronized block has a *wait* (in which case it is clearly intended to be non-atomic). For the C programs used in this experiment, we have manually annotated certain blocks in the program source code as intended-to-be-atomic. Note that in all the examples the runtime of the UCG-based analysis is negligible in comparison to the model checking time.

The results in Table 1 show that, if one relies on either the lock analysis alone or the fork-join-wait-notify based analysis alone, the number of (spurious) warnings (for data races or atomicity violations) can be large. In contrast, our UCG based analysis, by exploiting the interaction among these two types of happens-before constraints, is effective in pruning away spurious warnings. In our predictive analysis framework a precise SMT-based algorithm [16, 17] is used at the final step to check all the UCG warnings. The algorithm is precise in that it generates witness traces if and only if the UCG warnings are indeed real errors. In the end, all the witnesses generated can be fed to a special thread scheduler in the *Fusion* tool, to re-run the program and deterministically replay the actual violation.

6 Conclusions

We have proposed the notion of a *Universal Causality Graph (UCG)*, as a unified happens-before model for detecting bugs in concurrent programs. Given a concurrent (trace) program and a property, UCGs allow us to capture, as causality constraints, the set of all possible interleavings that are feasible under the scheduling constraints imposed by synchronization primitives and that may potentially lead to violations of the property while guaranteeing (i) soundness and completeness, (ii) scalability, (iii) handling of multiple synchronization primitives in a unified manner, and (iv) exploiting the synergy between causal constraints imposed by the property as well as the program. As an application, we demonstrated the use of UCGs in enhancing the precision and scalability of predictive analysis in the context of runtime verification of concurrent programs.

References

[1] www.cs.utexas.edu/users/kahlon/ .

[2] F. Chen and G. Rosu. Parametric and sliced causality. In *CAV*, 2007.

[3] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for dpll(t). In *CAV*, 2006.

[4] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI*, 2007.

[5] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *CAV*, 2008.

[6] A. Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *TACAS*, 2009.

[7] A. Farzan, P. Madhusudan, and F. Sorrentino. Meta-analysis for atomicity violations under nested locking. In *CAV*, 2009.

[8] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *IPDPS*, 2004.

[9] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, 2008.

[10] V. Kahlon. Boundedness vs. Unboundedness of Lock Chains: Characterizing Decidability of CFL-Reachability for Threads Communicating via Locks. In *LICS*, 2009.

[11] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, 2005.

[12] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[13] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *CCC*, 2002.

[14] C. Sadowski, S. N. Freund, and C. Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *ESOP*, 2009.

[15] K. Sen, G. Rosu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Formal Methods for Open Object-Based Distributed Systems*, pages 211–226. Springer, 2005. LNCS 3535.

[16] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *ISFM*, 2009.

[17] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *TACAS*, 2010.

[18] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Software Eng.*, 32(2):93–110, 2006.