GPredict: Generic Predictive Concurrency Analysis

Jeff Huang
Parasol Laboratory
Texas A&M University
Email: jeff@cse.tamu.edu

Qingzhou Luo and Grigore Rosu
Deparment of Computer Science
University of Illinois at Urbana-Champaign
Email: {luo2, grosu}@illinois.edu

***Abstract*—Predictive trace analysis (PTA) is an effective approach for detecting subtle bugs in concurrent programs. Existing PTA techniques, however, are typically based on adhoc algorithms tailored to low-level errors such as data races or atomicity violations, and are not applicable to high-level properties such as "a resource must be authenticated before use" and "a collection cannot be modified when being iterated over". In addition, most techniques assume as input a globally ordered trace of events, which is expensive to collect in practice as it requires synchronizing all threads. In this paper, we present GPredict: a new technique that realizes PTA for *generic* concurrency properties. Moreover, GPredict does not require a global trace but only the *local traces* of each thread, which incurs much less runtime overhead than existing techniques. Our key idea is to uniformly model violations of concurrency properties and the thread causality as constraints over events. With an existing SMT solver, GPredict is able to precisely predict property violations allowed by the causal model. Through our evaluation using both benchmarks and real world applications, we show that GPredict is effective in expressing and predicting generic property violations. Moreover, it reduces the runtime overhead of existing techniques by 54% on DaCapo benchmarks on average.**

I. Introduction

The difficulty of concurrent programming has inspired a wide range of fault detection and diagnosis tools. Among them, predictive trace analysis (PTA) has drawn a significant attention [18], [15], [16], [32], [13], [27], [29], [30], [10], [35], [36], [34], [20], [19]. Generally speaking, a PTA technique has two steps: it first records a trace of execution events at runtime, then, offline, it generates other (often exhaustive) permutations of these events under certain *causal model* of scheduling constraints, and predicts concurrency faults unseen in the recorded execution. PTA is powerful as, compared to dynamic analysis, it is capable of exposing bugs in unexercised executions and, compared to static analysis, it incurs much fewer false positives. Moreover, recent PTA techniques such as Penelope [32], PECAN [15], and [27] can not only predict faults, but also produce witnesses (*i.e.*, buggy schedules or even concrete executions) that manifest the faults, which can significantly speed up the debugging process.

We observe that existing PTA techniques are generally limited to detecting low-level memory access errors, such as data races [27], [18], atomicity violations [32], [36], atomic-set serialization violations [21], [33], or deadlocks [20], [19]. While these errors are common, they only capture a small portion of concurrency faults in real world programs. For example, consider a resource `authenticate-before-use` property, which requires a method `authenticate` to be always called before a method `use` that uses the resource. Any violation of this property is an indication of a serious security bug. However, it cannot be characterized by conventional data races, because in a violation of this property there may not even exist conflicting reads and writes to shared data. As another example, in Java, a collection is not allowed to be modified when an iterator is accessing its elements. This property, again, is neither a data race nor an atomicity violation, but a more generic contract on the use of Java Iterators. Existing techniques do not directly target these properties.

Moreover, while existing techniques are effective in detecting the targeted race or atomicity errors, their algorithms are usually adhoc and are not applicable to such more general properties. For instance, the cut-point based algorithm of Penelope [32] is specialized for predicting atomicity violations, and the pattern-directed graph search algorithm in PECAN [15] detects concurrency access anomalies only. Furthermore, for building the causal model, existing algorithms generally assume as input a linearized trace of events, which contains all the necessary causal ordering information (*e.g.*, happens-before) between critical events (*i.e.*, shared data accesses and synchronizations). However, this relies on the ability to track a globally ordered sequence of events by all threads at runtime, which often incurs hundreds or even thousands of times of program slowdown [15], [16], [32], [10], making these techniques less useful in practice.

In this paper, we present a new PTA technique, GPredict, that is able to predict violations of *high-level more generic properties*. Our central observation is that a vast category of concurrency faults, together with the causal model, can be modeled uniformly as first-order logical constraints between events. For example, for the `authenticate-before-use` property, suppose we model the calls of these two methods as two events, `auth` and `use`, and give each of them a corresponding order variable, $O_{\texttt{auth}}$ and $O_{\texttt{use}}$, respectively. A violation of this property can be simply modeled by the constraint $O_{\texttt{use}} < O_{\texttt{auth}}$, stating that the property is violated if there exists any feasible schedule in which the order of the `use` event is smaller than the order of the `auth` event. Similarly, violations of the collection iterator property can be modeled as $O_{\texttt{create}} < O_{\texttt{update}} < O_{\texttt{next}}$, specifying that the property is violated if a collection `update` event comes between the iterator `create` event and an iterator `next` event. Meanwhile, inspired by our prior work [18], we can

also soundly encode the causal model as constraints over the order of critical events. By solving a conjunction of these constraints, we can determine if a property can be violated in other feasible executions, hence to predict faults defined by the properties.

Based on the observation above, we first design a specification language for users to specify generic concurrency property violations. Our language is adapted from MOP [8], a runtime verification system for parametric properties. Similar to MOP, users of GPredict can declare the property events (which are parametric) with AspectJ pointcuts, and specify the property with a formalism over the declared events. Differently, in our formalism, we explicitly support concurrency properties by associating events with thread attributes and atomic regions, and allowing parallelism between events. To instantiate our design, we implemented an initial specification formalism for properties written in the form of regular expressions. We present our specification language and the constraint encoding algorithm for the property violations in Section II.

Another main contribution of this work is a new and sound causal model that is based on only the local traces of each individual thread, rather than a global trace. This new model not only ensures that GPredict never reports any false positive (*i.e.*, every property violation reported by GPredict is real), but also enables GPredict to be synchronization-free for collecting the execution traces at runtime, incurring much less runtime overhead than existing techniques. The main challenge we address is how to extract from the thread local traces the synchronization constraints (*e.g.*, causal orderings caused by the signal *wait/notify* events). We present a formal constraint modeling in Section III and prove its soundness.

We have implemented GPredict for Java programs and evaluated it on a set of real world applications with high level generic properties, as well as conventional data races, atomicity violations, and deadlocks written in our specification language. We show GPredict is both expressive and effective in predicting generic property violations. Moreover, comparing to the state of art techniques that log a global trace, GPredict has significantly better runtime performance due to the use of thread-local traces, incurring 10%-82% less overhead on DaCapo benchmarks [6]. We present the implementation and evaluation of GPredict in Sections IV and V, respectively.

In summary, we make the following contributions:

- We present a new predictive trace analysis (PTA) technique, GPredict, that is able to predict generic concurrency property violations based on constraint solving.
- We present a specification language for generic concurrency properties and the corresponding constraint encoding algorithm for the property violations.
- We present a sound constraint modeling of the predictive causal model with only the thread-local traces as input, which frees predictive analysis from expensive runtime synchronizations in order to obtain a global trace.
- We evaluate GPredict with real world applications and demonstrate its effectiveness and runtime performance for predicting generic property violations.

```
     Collection<Item> c;
   Item A, B;  Iterator i1, i2;
      T1                  T2
1: c.add(A);
2: start T2;           5: c.add(B);
3: i1=c.iterator();    6: i2=c.iterator();
4: i1.next()           7: i2.next()
```

Fig. 2. A collection iterating example

II. Generic Predictive Analysis

The idea behind PTA is that computations of a concurrent program may be scheduled in different orders, due to scheduling non-determinism, and that from one observed execution, a causal model can be constructed to infer a set of similar feasible executions of the same program, which can be used to predict behaviours not seen in the observed execution.

GPredict provides a general technique for the PTA of generic property violations based on constraint solving. The main idea is that both the causal model and the property violations can be modeled uniformly by first-order logical constraints. By solving the constraints, we can predict property violations in all the feasible executions captured by the causal model. There are two categories of events in our model:

- *property events*: declared in the property specification.
- *model events*: critical events that determine the causal model, *i.e.*, all the reads and writes to shared data and thread synchronizations.

We next present the specification and constraint modeling of property events. Model events are addressed in Section III.

A. Overview

We first give an overview of GPredict using a simple example. We then discuss the challenges and explain how we address them. In Fig. 2, there are two threads (`T1` and `T2`) accessing a shared collection. `T1` first initializes the collection with an item `A`, then it forks `T2` and iterates over the collection. In `T2`, it first adds item `B` to the collection, then iterates over the collection. This program, although intuitive, may throw a `ConcurrentModificationException` because when `T1` is iterating over the collection, `T2` might simultaneously update the collection, which breaks the contract of Java Iterators. Suppose our problem here is to detect this error. A classic solution is through runtime verification, such as MOP [8], that allows the users to specify the safe iterator property using specification formalisms, and automatically generates monitors to detect violations of the property at runtime. However, since the error depends on the thread schedule to manifest, which is non-deterministic, conventional runtime monitoring approaches may not detect it.

From a high level view, GPredict addresses this problem by analyzing the causal ordering relation between events observed at runtime with a constraint model. We give each event (including both property events and model events) an order variable representing its order in the schedule of a possible feasible execution and use these order variables to

<*GPredict Specification*>	::=	<*Property Name*> "("<*Parameters*>")" "{" <*Event*>* <*Pattern*> "}"
<*Event*>	::=	"event" <*Id*> <*AspectJ AdviceSpec*> ":" <*AspectJ Pointcut*>
<*Pattern*>	::=	"pattern :" (<*RegExp*> "||")*<*RegExp*>
<*Property Name*>	::=	<*Identifier*>
<*Parameters*>	::=	(<*Type*> <*Identifier*>)$^+$
<*Id*>	::=	<*Identifier*>
<*AspectJ AdviceSpec*>	::=	AspectJ AdviceSpec syntax
<*AspectJ Pointcut*>	::=	AspectJ Pointcut syntax
<*Thread*>	::=	<*Identifier*>
<*AtomRegion*>	::=	<*Identifier*>
<*Identifier*>	::=	Java Identifier syntax
<*Begin*>	::=	"<"<*AtomRegion*>
<*End*>	::=	">"<*AtomRegion*>
<*RegExp*>	::=	Regular expression over {<*Id*>, <*Id*>"("<*Thread*>,<*Begin*> | <*End*>")"}

Fig. 1. GPredict property specification language. The new syntax introduced for concurrency properties is highlighted in gray color.

formulate the property violation. For example, let O_i denote the order of the event at line i. A violation of the safe iterator property can be formulated as $O_3 < O_5 < O_4$. Similarly, the causal ordering constraints between events can be modeled as $O_1 < O_2 < O_3 < O_4 \wedge O_5 < O_6 < O_7$ (to respect the program order, *e.g.*, $O_1 < O_2$ means line 1 must happen before line 2) and $O_2 < O_5$ (to respect the synchronization semantics, *e.g.*, line 5 can only happen after line 2, because `T2` is forked at line 2). Conjuncting all these constraints, GPredict invokes an SMT solver (*e.g.*, Z3 [11]) to solve them. If the solver returns a solution, it means that there exists a schedule that violates the property. Moreover, such a schedule represents a witness to the property violation, and can be deterministically replayed to manifest the error. Back to our example, the solver may return O_1=1, O_2=2, O_3=3, O_4=5, O_5=4, O_6=6, O_7=7, which corresponds to the property violating schedule `1-2-3-5-4-6-7`.

Although our technique can be easily illustrated, there are several challenges we must tackle:

1) *Property specification.* How to specify the properties? What type of formalisms can we support? How to specify conventional concurrency errors as well, such as races, atomicity violations, deadlocks, etc?
2) *Property encoding.* How to encode the constraints for parametric properties? For example, in Figure 2 both line 4 and line 7 access an iterator of the collection, but on different instances; if we ignore this difference, we might formulate the property as $O_3 < O_5 < O_7$ instead of $O_3 < O_5 < O_4$, which would result in missing the real property violation.
3) *Soundness*[1] *(i.e. No-false-positive).* How to guarantee that every property violation we detect is real? For example, if there exists certain causal order not modeled by our constraints, the detected property violation might be false.

In the rest of this section, we focus on discussing the first two issues. We present a formal constraint modeling of our sound causal model to address the third issue in Section III.

[1]By *soundness* we mean *no-false-positive* in this paper.

```
UnsafeIterator (Collection c, Iterator i) {
  event create after(Collection c) returning(Iterator i) :
        call(Iterator Collection+.iterator()) && target(c);
  event update after(Collection c) :
        (call(* Collection+.remove*(..))
        || call(* Collection+.add*(..)) ) && target(c);
  event next before(Iterator i) :
        call(* Iterator.next()) && target(i);

  pattern: create next* update+ next
}
```

Fig. 3. UnsafeIterator property specification

B. Generic property specification

GPredict allows specifying properties using regular expressions (RegExp). We choose RegExps as they are natural and convenient to reflect the ordering relation between property events. Nevertheless, our technique should work with any formalism whose properties/formulae can be monitored using finite-state machine monitors (*e.g.*, linear temporal logic).

Fig. 1 shows the syntax of our property specification language. It is an extension of the MOP specification [8], consisting of the property declaration (name and parameters), a list of event definitions, and a formula specifying the property. The event syntax makes use of AspectJ, containing an identifier, an advice (with no body), and a pointcut. The property is then defined in terms of the event identifiers using RegExp. Fig. 3 shows an example of the `UnsafeIterator` property in our specification. The property is parameterized by a collection and an iterator. There are three types of events defined in the specification: `create` (creating an iterator i of the collection c), `update` (adding or removing an item to/from the collection), `next` (iterating over the collection via calling `next()` on the iterator). The formula of the property violation pattern is written as `create next* update+ next`, meaning that the property is violated if an `update` event can happen after `create` and before a `next`. Events in this pattern are parameterized by c and i as defined in the specification.

```
AtomicityViolation (Object o){
    event begin before(Object o): execution(m());
    event read before(Object o): get(* s) && target(o);
    event write before(Object o): set(* s) && target(o);
    event end after(Object o): execution(m());

    pattern: begin(t1,<r1) read(t1) write(t2) write(t1) end(t1,>r1)
}
```

Fig. 4. Atomicity violation property specification

```
        m()            T1          T2
1:   r1 = s;
2:   r2=r1+1;       4: m();     6: m();
3:   s= r2;         5: m();
```

Fig. 5. An example of atomicity violations

To explicitly support concurrency related properties, a major difference of our specification from MOP is that in the property formula, the event identifiers are also allowed to bind with thread attributes and begin/end of atomic regions, in the form of <*Id*>(<*Thread*>,<*Begin*>|<*End*>). The <*Thread*> attribute denotes a meta ID of the thread performing the corresponding event, such that events bound with different <*Thread*> attributes are by different threads. The <*Begin*> and <*End*> attributes are written as "<"<*AtomRegion*> and ">"<*AtomRegion*>, denoting the begin and end of an atomic region identified by <*AtomRegion*>.

Fig. 4 shows an example of the `read-write-write` atomicity violation written in our specification language that uses these attributes. The atomicity violation is concerned with three accesses to a shared variable s by two threads, which can be declared as `read` and `write` events using the `get` and `set` pointcuts. The `begin` and `end` events mark the beginning and ending of the execution of a method `m`, which is considered to be atomic. In the formula, to distinguish events by different threads, we bind each event with a thread attribute, *e.g.*, `read(t1)` and `write(t2)`. To match `begin` with `end`, they are written as `begin(t1,<r1)` and `end(t1,>r1)`, ensuring that these two events are marking the same atomic region (denoted by a meta ID `r1`). The whole formula is then written as `begin(t1,<r1) read(t1) write(t2) write(t1) end(t1,>r1)`, denoting that the violation occurs if the two `read` and `write` events inside an atomic region marked by the `begin` and `end` events of any thread `t1`, can be interleaved by a `write` event from a different thread `t2`. Fig. 5 shows a simple program with such atomicity violations. Note that the specification of atomic regions in our language is general and can be specified by arbitrary events and their orders, which is much more expressive than conventional atomic regions that are limited to synchronization methods or blocks.

In addition, we introduce a new notation "`||`" in our specification language, which is used to denote the parallelism between events. For example, <*Id1*> `||` <*Id2*> means that the two events <*Id1*> and <*Id2*> can be executed in parallel, with no causal ordering between each other. This notation is useful for specifying a range of interesting properties, *e.g.*, data

```
DataRace (Object o) {
    event read before(Object o): get(* s) && target(o);
    event write before(Object o): set(* s) && target(o);

    pattern: read(t1) || write(t2)
}
```

Fig. 6. Data race property specification

races. Fig. 6 shows the specification of a read-write data race property on a shared variable s. The property is parametrized by the object instance of s to distinguish different memory locations. The read event is declared as a `get` pointcut, and the write event as `set`. The formula is then written as `read(t1) || write(t2)`, meaning that the two events by two different threads can happen in parallel.

C. Property encoding

Recall Fig. 1 that properties are written as RegExp patterns over the alphabet of the declared event identifiers. Since the events are defined with pointcuts, which can be triggered multiple times in the execution, each declared event may correspond to multiple event instances in the execution. We shall refer to such event instances as property events.

Consider the order of each event identifier in the RegExp patterns. For the pattern to be satisfied, there must exist a corresponding ordered sequence of property events such that each event matches with the corresponding event identifier. In other words, the pattern actually specifies the ordering constraints between property events, which can be directly modeled by their corresponding order variables. To model the constraints specified in the pattern, however, we must address the following important problems:

1) Property events must be parametric; how to handle the parametricity?
2) An event identifier may have multiple matching property events; how to encode the constraints for all?
3) Our pattern allows the RegExp quantifiers ("`?`", "`*`", "`+`"), as well as negation "`!`", boolean logics "$\vee$" and "$\wedge$", and grouping parentheses "`()`", and supports the bindings of thread attributes, atomic regions, and the parallel notation "`||`"; how to handle all these features?

1) Parametricity: For parametric property events, the key is to bind the property parameters to concrete object instances. Each binding corresponds to a different property instance, and we construct a separate constraint. Constraints of all property instances can then be combined together by disjunction ($\vee$). To create the bindings for each instance, we enumerate the set of object instances corresponding to each parameter associated with the property events. For each object instance, we create a separate binding to the corresponding parameter. By joining the bindings for all parameters in the property, we can create the bindings for all property instances. The total number of bindings is a multiplication of the number of object instances for each parameter. Back to our example in Fig. 2, there is only one binding to the Collection parameter, *c*, and two to the

Iterator parameter, i_1 and i_2; hence, there are two parametric bindings in total: (c, i_1) and (c, i_2).

2) Multiple event instances: For each event identifier in the pattern, since the existence of one such property event is sufficient to witness the property violation, if there are multiple events corresponding to an identifier, it seems intuitive to pick any one to build the constraint. However, this naive approach may miss predictable property violations, because the constraint with respect to the chosen event might not be satisfied, while there might exist other events that are not chosen that can manifest the property violation. To address this issue, similar to parametric bindings, we enumerate the corresponding events per property instance that match with the event identifier in the pattern. For each property event, we create a separate ordering constraint and disjoin them. In this way, no property violation will be missed. Back to our example, for the property instance (c, i_1), there exist two `update` events (at lines 5 and 1, respectively), so we construct the disjunction $O_3 < O_1 < O_4 \vee O_3 < O_5 < O_4$. Although $O_3 < O_1 < O_4$ cannot be satisfied (because line 1 must happen before line 3), $O_3 < O_5 < O_4$ can, so the property violation can still be detected.

3) RegExp pattern constructs: Taking the RegExp pattern as input, we first preprocess it to handle quantifiers ("$\star$", "+", "?"). For "?", we replace it by "$|\epsilon$", meaning that one or zero of its preceding event identifier may appear in the pattern. For "$\star$" and "+", because both of them can denote an infinite number of events, to avoid exploding the constraints, we remove "$\star$" and its associated event identifier or identifiers from the pattern (because "$\star$" can denote zero event), and remove "+" from the pattern (because "+" can denote one event). For example, the `UnsafeIterator` pattern in Fig. 3 "`create next* update+ next`" is processed to "`create update next`". This treatment, however, may result in missing certain violations. We expect this is acceptable since GPredict is used for predicting if a property can be violated or not. The number of violations is less important. In fact, we may simply exclude "$\star$" and "+" from the specification. We choose not to, so that existing MOP properties can be supported without any change.

In constructing the constraints, we handle "`!`", "$\vee$", "$\wedge$", "`||`", and "`()`" as follows. For "`!`", we add a negation (*i.e.*, a logical *not*) to the corresponding constraint. Note that "`!`" may conflict with our treatment to "$\star$" and "+", which results in over-approximation when "`!`" and "$\star$" (or "+") are used together in the pattern. To avoid this issue, we disallow such patterns in our specification. For "$\vee$" and "$\wedge$", we take them as disjunction and conjunction, respectively, between the corresponding constraints. For "`||`", we create an equality constraint between the order variables of the two events[2]. For example, for the data race specification in Fig. 6, for any pair of such property events $e_i||e_j$, we add the equality constraint $O_i = O_j$. For parentheses "`()`" that embrace thread attributes and atomic regions, we handle them as follows.

[2]Currently, we only support "`||`" for two parallel events.

For thread attributes, similar to the treatment of parametric bindings, we first group the corresponding events by their thread ID, and then enumerate each group. During enumeration, the only condition is that events with different thread attributes must be bound to different groups of events. For example, consider the example in Fig. 5 with the data race pattern "`read(t1)||write(t2)`", we can bind both `t1` and `t2` to either `T1` or `T2`, but they cannot be simultaneously bound to the same thread. The constraints of different groups are then combined together by disjunction.

For atomic region attributes, note that we must match each *<Begin>* event with its corresponding *<End>* event; otherwise, it might lead to false alarms. Taking Fig. 5 as an example. Suppose we change the method `m` to be synchronized, then there is no atomicity violation. However, since `m` is called twice by thread `T1`, there are two `begin` and two `end` events by `T1`; if the first `begin` is matched with the second `end`, then the ordering of events can still be satisfied when `T2` calls `m` between the two calls of `m` by `T1`, which is not a real atomicity violation. Hence, to maintain a correct match, we preprocess all events with atomic region attributes. Because all such events are always nested, we can simply use a stack to keep track of the current active atomic region, and match each *<Begin>* with the correct *<End>*. We assume the specified RegExp pattern is consistent, and currently we do not perform any static checking for it. Otherwise, if the pattern is inconsistent, no violation will be predicted.

III. Constraint Modeling

Several previous work [36], [13], [27], [18] have used first-order logical SMT formulae to model the ordering constraints between events. Our constraint modeling of the causal model extends our prior work [18], with the main improvement that it is built upon the thread-local traces instead of a global trace.

We consider the following types of model events:

- *begin*(t)/*end*(t): the first/last event of thread t;
- *read*(t, x, v)/*write*(t, x, v): read/write a value v on a shared variable x;
- *lock*(t, l)/*unlock*(t, l): acquire/release a lock l;
- *fork*(t, t')/*join*(t, t'): fork a new thread t'/block until thread t' terminates;
- *wait*(t, l, g): a composition of three consecutive events[3] *unlock_w*(t, l)-*wait*(t, g)-*lock_w*(t, l): first release lock l, then block until receiving signal g, finally acquire l;
- *notify*(t, g)/*notifyAll*(t, g): send a signal g to wake up a waiting thread/all waiting threads.

From a high level view, taking the model events by each thread as input, we encode all the necessary ordering constraints between model events as a set of first-order logic formulae. The whole formula, Φ, is a conjunction of three sub-formulae over the order variables of the model events:

$$\Phi = \Phi_{mhb} \wedge \Phi_{sync} \wedge \Phi_{rw}$$

[3]In this work, we do not model spurious wakeups, which happen rarely in practice and are typically handled by enclosing wait in loops.

A. Must happens-before constraints (Φ_{mhb})

The must happens-before (MHB) constraints capture the causal order between events that any execution of the program must obey. Let $\prec$ denote the smallest transitively-closed relation over the events in the observed execution such that for any two events e_i and e_j, $e_i \prec e_j$ holds if one of the following holds:

- Program Order: e_i and e_j are by the same thread, and e_i occurs before e_j.
- Fork Order: e_i=*fork*(t, t') and e_j=*begin*(t').
- Join Order: e_i=*end*(t) and e_j=*join*(t', t).

For each MHB relation $e_i \prec e_j$, we add a conjunction of the constraint $O_i < O_j$ to Φ_{mhb}. The size of Φ_{mhb} is linear in the total number of model events.

B. Synchronization Constraints (Φ_{sync})

The synchronization constraints capture the locking and wait-notify semantics introduced by synchronization events: *lock*, *unlock*, *wait*, *notify*, and *notifyAll*. Recall the semantics that a *wait*(t, l, g) event can be split into three events: *unlock_w*(t, l)-*wait*(t, g)-*lock_w*(t, l). Hence, we divide each lock region enclosing *wait* into two smaller lock regions. Φ_{sync} is constructed as a conjunction of two constraints:

Locking constraints (Φ_{lock}) For each lock l, we first extract a set S of all the (*lock*,*unlock*) pairs on l (including *unlock_w* and *lock_w* from the *wait* events), following the program order locking semantics, *i.e.*, an *unlock* event is paired with the most recent *lock* event on the same lock by the same thread. We then add the following constraint to Φ_{lock}:

$$\bigwedge_{(a,b)\neq(a',b')\in S} (O_b < O_{a'} \vee O_{b'} < O_a)$$

The size of Φ_{lock} is quadratic in the number of lock regions.

Wait-notify/notifyAll constraints (Φ_{signal}) The core problem of constructing Φ_{signal} is to find, for each *wait* event, a matching *notify* or *notifyAll* event that can send the correct signal to wake it up. In previous predictive trace analysis work [15], [16], [18], [36], [27], this task is easy, because a global trace is available and each wait can be simply matched with the most recent *notify* or *notifyAll* event with the same signal. However, this problem becomes challenging when we have only the thread-local traces, where there is no causal ordering information between *wait*/*notify*/*notifyAll* events across different threads. For example, we cannot match a *wait* with an arbitrary *notify*, because the *notify* might happen after the *wait*, or it might have been matched with another *wait*.

We develop a sound constraint model that addresses this issue. Our key observation is that *wait* and *notify*/*notifyAll* events are always executed inside a lock region. For a *wait*$(t1, l, g)$ to match with a *notify*$(t2, g)$, suppose the enclosing lock regions of *wait*(t, l, g) and *notify*$(t2, g)$ are marked by *lock*$(t1, l)$/*unlock*$(t1, l)$ and *lock*$(t2, l)$/*unlock*$(t2, l)$, respectively, it must hold that the *unlock_w*$(t1, l)$ event must happen before *lock*$(t2, l)$. Otherwise, *notify*$(t2, g)$ would happen before *unlock_w*(t, l) and the signal would be lost. Meanwhile, for all the other *wait* events, they must be either before *lock*$(t2, l)$ or after *unlock_w*$(t1, l)$. Otherwise, *notify*$(t2, g)$ might be matched with more than one *wait* event.

Specifically, let X and Y denote the set of *wait* and *notify* events on the same signal. For each *wait*(t, l, g) event w, let O_w^{ul}, O_w, and O_w^l denote the corresponding order variables of *unlock_w*(t, l), *wait*(t, g) and *lock_w*(t, l), respectively, and let O_{l_e}/O_{ul_e} denote the order variables of the *lock*/*unlock* events of the enclosing lock region of a *wait* or *notify* event e. Φ_{signal} for w is written as follows:

$$\bigvee_{w\in X, n\in Y}(O_w^{ul} < O_{l_n} \wedge O_n < O_w \bigwedge_{w\neq w'\in X}(O_{ul_w} < O_{w'}^l \vee O_{ul_w'} < O_{l_n}))$$

The constraint model for *wait*-*notifyAll* is similar, except that the conjunction over the other *wait* events in X is not needed, because a *notifyAll* event can be matched with multiple *wait* events. The total size of Φ_{signal} is $2|X|^2|Y|$, which is cubic in the number of *wait*/*notify*/*notifyAll* events.

C. Read-write constraints (Φ_{rw})

The read-write constraints ensure the data-validity of events: a read must read the same value as that in the observed execution, though it may be written by a different write. Specifically, for each property event p, we add a constraint $\Phi_{rw}(p)$ to Φ_{rw}. $\Phi_{rw}(p)$ is constructed over a set, R, containing all the read events that must happen-before ($\prec$) it. For each *read*(t, x, v) in R, let W denote the set of *write*$(_, x, _)$ events in the trace (here '_' means any value), and W_v the set of *write*$(_, x, v)$ events. $\Phi_{rw}(p)$ is written as:

$$\bigwedge_{\forall w\in W_v} \Phi_{rw}(w) \wedge O_w < O_r \wedge \bigwedge_{\forall w'\in W} O_{w'} < O_w \vee O_{w'} > O_r$$

The constraints above state that the read event, r, may read the value v on x written by any write event, $w = $ *write*$(_, x, v)$, in W_v (which is a disjunction), with the constraint that the order of w must be smaller than the order of r, and there is no other *write*$(_, x, _)$ event that is between them. Moreover, this *write*$(_, x, v)$ event itself must be feasible, so we add a conjunction of the constraint $\Phi_{rw}(w)$.

The size of Φ_{rw} is cubic in the number of *read* and *write* events. Nevertheless, in practice, Φ_{rw} can be significantly reduced by considering the MHB relation $\prec$. For example, consider two *write* events $w1$ and $w2$ in W_v. If $w1 \prec w2 \prec r$, we can exclude $w1$ from W_v because it is impossible for r to read the value written by $w1$ due to Φ_{mhb}. Similarly, for any $w' \in W$, if $r \prec w'$, then w' can be excluded from W. Also, when constructing the constraints for matching an event $w \in W_v$ to r, if $w' \prec w$, then w' can be skipped.

D. Soundness

The next theorem states that our constraint modeling based on the thread-local traces is sound:

Theorem 1. *Φ captures a sound causal model, i.e., any solution to Φ represents a feasible schedule.*

Proof. (Sketch) It's clear that Φ_{mhb} and Φ_{rw} capture the data and control dependencies for every event in the trace. The

```
T1                      T2                      T3
x1: lock(l)             y1: lock(l)
    ...                     ...                 z1: lock(l)
x2: unlock_w(l)         y2: unlock_w(l)             ...
x3: wait(g);            y3: wait(g);            z2: notify(g);
x4: lock_w(l);          y4: lock_w(l);              ...
    ...                     ...                 z3: unlock(l);
x5: unlock(l);          y5: unlock(l);
```

MHB constraint: x1<x2<x3<x4<x5 & y1<y2<y3<y4<y5 & z1<z2<z3

Locking constraints: (x1>y2 | x2<y1) & (x1>y5 | x2<y4) & (x1>z3 |x2<z1) & (x3>y2 | x4<y1) & (x3>y5 | x4<y4) & (x3>z3 |x4<z1) & (y1>x5 | y2<x4) & (y1>z3 |y2<z1) & (y4>x5 | y5<x4) & (y4>z3 |y5<z1)

Wait-notify constraints: x2<z1 & z2<x3 & (x5<y4 | y5<z1) && y2<z1 & z2<y3 & (y5<x4 | x5<z1)

Fig. 7. Example of synchronization constraints

only less obvious part is the thread synchronization constraints captured by Φ_{sync}. Let's use an example in Fig. 7 to show the soundness of Φ_{sync}. For simplicity, we use the label to the left of each event to refer to both the event and its order variable. In the example program, both threads `T1` and `T2` perform a *wait* within a lock region (`x1`/`x5` and `y1`/`y5` respectively) on lock l, and `T3` performs a *notify* (`z2`) within a lock region (`z1`/`z3`) on the same lock. Following the semantics of *wait*, the *wait* events of `T1` and `T2` are split into six events (denoted by `x2`-`x3`-`x4` and `y2`-`y3`-`y4`, respectively). Hence, there are five *lock*/*unlock* pairs in the trace. Our locking constraints (shown in the figure) enforces that every two *lock*/*unlock* pairs by different threads cannot overlap. Clearly, mutual exclusion is ensured. For wait-notify, note that there is only one *notify* event but two *wait* events, either the *wait* event of `T1` or of `T2` can be matched with the *notify* event, but not both. Following our construction of Φ_{signal}, our constraints for the two *wait* events are written as $x2 < z1 \wedge z2 < x3 \wedge (x5 < y4 \vee y5 < z1)$ and $y2 < z1 \wedge z2 < y3 \wedge (y5 < x4 \vee x5 < z1)$. By analyzing the conjunction of these two constraints, we can see that it contradicts with the locking constraints. Hence, Φ_{signal} cannot be satisfied, which ensures the semantics that a *notify* cannot be matched with more than one *wait* event. □

IV. Implementation

We have implemented GPredict for Java programs based on JavaMOP [8] and Soot [22]. Fig. 8 shows an overview of the GPredict infrastructure. Taking the target program (Java bytecode) and the property specification as input, GPredict first adds necessary instrumentation into the program for logging the model events during execution, and uses JavaMOP's front-end parser to produce a corresponding AspectJ file for the property. During program execution, the AspectJ file is weaved into the instrumented program to emit the property events. All events are logged and grouped by their thread ID, and saved into database together with the property pattern. Then taking the logged information as input, the offline analyzer constructs the necessary constraints and invokes an SMT solver to solve them. A property violation is reported if the solver returns a solution. We next present each of the components in detail.

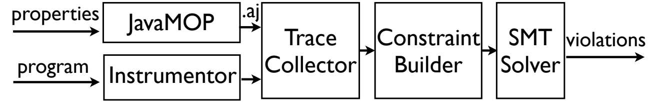

Fig. 8. Overview of GPredict architecture

Instrumentation This phase consists of two parts. The first part takes in the target program, and performs the instrumentation on Jimple, an intermediate representation of Java bytecode in the three-address form [22]. The instrumented events include read and write accesses to shared data such as instance fields and class variables, entry and exit to monitors and synchronized methods, wait and notify/notifyAll method calls, and thread fork and join. The second part is parsing the property specification. Since our specification language is adapted from MOP, we make use of JavaMOP parser to produce an AspectJ file with each declared event converted into a corresponding pointcut. The aspects are then weaved into the instrumented program dynamically to emit both the model events and property events at runtime.

Trace Collection For each model event, we log the runtime data as described in Section III, such as the thread ID, the memory address, the read or write value, etc. The logging of property events is slightly different. Recall Section II-C1 that for parametric properties, we need to group events into different property instances according to the runtime object of the event. Instead of performing this grouping task offline, we do it online by reusing the monitoring mechanism of JavaMOP. Specifically, JavaMOP internally creates a separate monitor for each property instance, and matches each event to all the related monitors. Hence, we simply insert a logging method call in each monitor function and save the property event associated with the monitor ID (which is equivalent to the property instance) into database. During constraint construction, we can directly use the monitor ID to identify each property instance without grouping the events again.

In order to reduce the runtime overhead, remember that our technique does not collect a global trace but the events for each thread separately. We maintain for each thread a thread-local buffer for storing the events performed by itself. Once a new event is created by a thread, we add it to the thread's local buffer. At the end of the logging phase, all events are saved into database indexed by the thread ID.

Constraint Construction and Solving The constraint construction follows the algorithms in Section II-C (for property constraints) and Section III (for model constraints). It is worth noting that our constraint model is very extensible. It is not limited to a single property, but multiple properties can be encoded simultaneously. For instance, the `UnsafeIterator` property can be encoded together with the data race patterns by a disjunction. For solving the constraint, we use Z3 [11] in our implementation and set the timeout to five minutes. Note that almost all of our constraints are ordering constraints (*i.e.*, comparing two integer variables by "$<$"), which can be effi-

```
public void addChangeListener(SeriesChangeListener listener) {
        this.listeners.add(listener);
}
public void fireSeriesChanged() {
        notifyListeners(new SeriesChangeEvent(this));
}
protected void notifyListeners(SeriesChangeEvent event) {
        Iterator iterator = listeners.iterator();
        while (iterator.hasNext()) {
            SeriesChangeListener listener =
                            (SeriesChangeListener) iterator.next();
          listener.seriesChanged(event);}
}
```

Fig. 9. JFreeChart bug#1051

ciently solved by the Integer Difference Logic (IDL). The only exception is the equality constraints (*i.e.*, $O_i = O_j$) encoded for "||" in the property specification. For such constraints, we simply filter them out by replacing all occurences of O_i in the constraints by O_j.

V. Evaluation

We have evaluated GPredict on a set of real concurrent programs with both generic properties and conventional concurrency errors written in our specification language. Moreover, to assess the improved runtime performance of GPredict over previous techniques by using thread-local traces, we have run GPredict with a set of DaCapo [6] benchmarks and compared its performance with the approach of logging global traces. This section presents our results. All experiments were conducted on a 8-core 3.50GHz Intel i7 Linux machine.

A. Effectiveness

We have applied GPredict to six programs: `Derby`, `H2`, `JFreeChart`, `Jigsaw`, and two JDK libraries and examined properties including `UnsafeIterator`, `NullPointer Dereference`, `UnsafeDatabaseAccess`, `Check-Then-Act`, as well as data races, atomicity violations, and deadlocks. The results (shown Table 1) demonstrate that GPredict is effective in expressing properties and predicting violations.

1) UnsafeIterator: Fig. 9 shows a real bug violating the `UnsafeIterator` property (as explained in Section II-B) in `JFreeChart` [1]. When the two methods `addChangeListener` and `fireSeriesChanged` are called concurrently by different threads, a `ConcurrentModificationException` may be thrown. The reason is that in `fireSeriesChanged` an arraylist of listeners are iterated to notify the `SeriesChangedEvent`, while new listeners can be added to the arraylist from `addChangeListener` concurrently. This error is common in concurrent programs, however, as it is neither a data race nor an atomicity violation, it cannot be detected by conventional race or atomicity violation detectors.

With GPredict, it is fairly easy to specify the `UnsafeIterator` property (as shown in Fig. 3) and to predict violations. Based on a *normal run* of the test driver provided in the bug repository (which does not manifest the bug), GPredict captured 90 property events and 140 model events and predicted 20 violations within a second. We manually inspected all these violations and empirically confirmed that these 20 violations were all real. Note that each violation is unique with respect to the event sequences defined in the property. We did not further prune redundant violations with the same event signature.

Derby bug #2861

```
class TableDescriptor
public String getObjectName() {
      if(referencedColumnMap ==null) {...}
      else {
           if(referencedColumnMap.isSet(...)){...}
      }
}
public void setReferenceColumnMap(..) {
      referencedColumnMap=null;
}
```

```
NullPointerDereference (Object o) {
   event deRef before(Object o):
          get(TableDescriptor.referenceColumnMap)&&target(o);
   event setNull before(Object o, Object value):
          set(* TableDescriptor.referenceColumnMap)
             && target(o)&&args(value)&&if(value==null);

   pattern: deRef(t1) || setNull(t2)
}
```

Fig. 10. Null-pointer dereference specification

2) Null-Pointer Dereferences: Null-pointer dereference errors are common in multithreaded programs. Though they are not unique to concurrency, they are much harder to detect in multithreaded programs. Fig. 10 shows the concurrency bug #2861 in Apache Derby [2]. This bug is concerned with a thread safety issue in the `TableDescriptor` class. The shared data `referencedColumnMap` is checked for `null` at the beginning of the `getObjectName` method and later dereferenced if it is not `null`. Due to an erroneous interleaving, another thread can set `referencedColumnMap` to `null` in the `setObjectName` method and causes the program to crash by throwing a `NullPointerException`.

This bug is in fact an atomicity violation, but it can be specified more intuitively as a null-pointer dereference. Users need only to declare two events, `deRef` and `setNull`, on the variable `referencedColumnMap`, and specify the pattern as `deRef(t1)||setNull(t2)`, meaning that the two events are from different threads and can be run in parallel. Because `deRef` is declared as an event on dereferencing `referencedColumnMap`, and `setNull` an event setting `referencedColumnMap` to `null` on the same `TableDescriptor` object (represented by the property parameter *o*), a null-pointer dereference can happen if the pattern is satisfied. We ran GPredict on Derby with this property. GPredict collected a trace with around 12K model events and 27 property events, and found 5 violations in 5s.

Note that although our specification in this example (Fig. 10 bottom) only concerns about the field `referencedColumnMap`, it could be written for *arbitrary*

TABLE I
EXPERIMENTAL RESULTS

Program	LOC	Property	#Threads	#Model events	#Property events	#Violations	Time
jfreechart	51k	UnsafeIterator	21	140	90	20	0.85s
h2	136k	UnsafeDatabaseAccess	5	112	14	16	0.67s
derby1	302k	NullPointerDereference	3	12527	27	5	4.7s
derby2	368k	Check-Then-Act	3	19889	1325	4	8.5s
jigsaw	101k	Data race	12	17089	38	29	17.6s
stringbuffer	358	Atomicity violation	3	58	6	2	0.4s
jdk-logger	2587	Deadlock	2	509	61	1	2.6s

```
UnsafeDatabaseAccess(Connection conn, String table) {
  event open after() returning(Connection conn):
    call(Connection DriverManager.getConnection(String));
  event close before(Connection conn):
    call(void Connection.close())&&target(conn);
  event create before(Connection conn, String table):
    call(* createTable(Connection conn, String table))&&args(conn,table);
  event delete before(Connection conn, String table):
    call(* deleteTable(Connection conn, String table))&&args(conn,table);
  event update before(String table, String sql):
    call(boolean Statement.execute(String))
    &&args(sql)&&if(sql.contains(table));

  pattern : !(open create update delete close)
}
```

Fig. 11. Unsafe database access specification

or *all* reference fields. If the user wants to detect *all* null-pointer dereferences, she can simply replace the parameter of the get/set pointcuts with a wild card "*". In that case, our algorithm will enumerate all fields.

3) UnsafeDatabaseAccess: Database applications typically follow some safe patterns to access data. For example, an authenticated connection must be established first before any other operation, a table must be created before any update or query accesses on the table, all operations must finish before closing the database connection, etc. There is likely a bug if the application violates such patterns. Fig. 11 shows the specification of an `UnsafeDatabaseAccess` property, which defines five property events (open connection, create table, update table, delete table, and close connection) over two parameters (the connection and the table name). The pattern `!(open create update delete close)` specifies that the property is violated when these events happen in an order different from the written one. The negation symbol "`!`" is interpreted as a logical `not` in the property constraints.

The `UnsafeDatabaseAccess` property cannot be handled by existing techniques such as serializability violation detectors [21], [40] (even with event abstraction) or typestate-based detectors [38], [39], because any violation instance of this property contains events over multiple objects. In our experiment, we wrote a simple multithreaded client program for testing the H2 database server with GPredict. GPredict found 16 violations in less than a second based on a trace of the client program with 112 model events and 14 property events[4].

4) Check-Then-Act: Collections are frequently used following the `Check-Then-Act` idiom: the code first checks

[4]As H2 was run in the server model, we did not log its model events.

Derby bug #3786

```
class GenericLanguageConnectionContext
public void removeStatement(...) {
    Cacheable cachedItem =
        statementCache.findCached(statement);
    if (cachedItem != null)
        statementCache.remove(cachedItem);
}
```

```
CheckThenAct (Map m, Object key) {
  event check after(Map m, Object key):
      call(* Map+.get(Object))&&target(m)&&args(key);
  event act before(Map m, Object key) :
      call(* Map+.remove(Object,..))&&target(m)&&args(key,..);

  pattern: check(t1) act(t2)+ act(t1)
}
```

Fig. 12. Check-Then-Act specification

a condition, and then acts based on the result of the condition. However, in concurrent programs, misuse of this idiom can easily lead to bugs [23]. Fig. 12 shows another bug in Derby [3]. The method `removeStatement` first checks if the `statementCache` contains a statement, and if yes it removes the statement by calling the method `remove`. To support concurrent accesses, the data structure for maintaining the `statementCache` is implemented as a `ConcurrentHashMap`. However, due to some bad interleaving, more than one thread might still execute the `remove` method concurrently, causing an assertion failure eventually.

Fig. 12 (bottom) shows the `Check-Then-Act` property in our specification language. In the pattern, the `check` event and the second `act` event are bound to thread `t1`, and the first `act` event to `t2`. The pattern means that a violation happens if two `check` and `act` events consecutive in a certain thread can bracket another `act` event from a different thread. GPredict collected a trace of 20K model events and 1.3K property events, and found 4 violations in around 8s.

5) Races, atomicity violations, and deadlocks: Our technique also works seamlessly for predicting conventional errors such as data races, atomicity violations, and deadlocks, without doing anything specific for them. Moreover, these errors can be specified more intuitively with our specification language by high level events than previous techniques that rely on checking low level shared data accesses. We have also applied GPredict to predict data races in Jigsaw and a deadlock bug [4] in JDK logging package. Due to space limit we omit

TABLE II
RUNTIME PERFORMANCE COMPARISON

Program	#Threads	Base	Global	GPredict
avrora	4	1.8s	2m20s	2m6s (**-10%**)
batik	2	2.1s	2m28s	1m2s (**-58%**)
xalan	11	1.7s	6m14s	1m9s (**-82%**)
lusearch	10	1.4s	15m17s	3m16s (**-78%**)
sunflow	25	1s	19m48s	11m28s (**-42%**)

the discussions. All details and more examples can be found at `http://parasol.tamu.edu/~jeff/gpredict/`.

B. Runtime performance

To understand the performance improvement of our technique, *i.e.*, enabling the use of thread-local traces instead of a global trace, we quantify GPredict using a set of widely used third-party concurrency benchmarks from DaCapo (shown in Table 2). All these benchmarks are real world applications containing intensive shared data accesses and synchronizations by concurrent threads. Previous PTA techniques can incur significant runtime slowdown on these benchmarks because logging a global trace requires synchronizing every model event with a global lock protecting the logging data structure. However, logging thread-local traces allows the recording computation by different threads to be done in parallel, which is much cheaper as no extra synchronization is required.

To perform an unbiased comparison, we also implemented in GPredict the ability of logging global traces, to ensure the same events are logged at runtime. In our experiment, we configure GPredict to run each benchmark with three different modes for logging the model events: no loggging, logging per-thread traces, and logging a global trace. Table 2 shows the results. All data were averaged over five runs. Compared with logging a global trace, recording the thread-local traces incurs significantly less runtime overhead with respect to the base execution. On average, GPredict reduces the runtime overhead by 54% for the evaluated benchmarks, ranging between 10% and 82%. For `xalan`, GPredict is even more than four times faster. Although the overhead of GPredict is still large (because there is a myriad of model events to log in these benchmarks), compared to previous techniques [15], [16], [10], [34], [36], [27] that require logging a global trace, GPredict significantly improves the runtime performance of predictive trace analysis. To further reduce the overhead, we can use static analysis techniques [7], [14] to eliminate redundant events during instrumentation.

VI. RELATED WORK

As discussed in Section I, although a large body of predictive analysis work has been developed [29], [30], [10], [12], [26], [32], [15], [21], [28], a common difference between these work and GPredict is that their algorithms are typically tailored to low-level memory access errors such as data races and atomicity violations and do not work for the generic concurrency properties we address in this work. With GPredict, developers are able to specify high-level properties using aspects and regular expressions and to predict bugs related to specified code regions that are suspected to be buggy.

Besides the specification language, another important contribution of this work is our sound causal model based on thread-local traces, which is realized with constraints. A few different causal models have been proposed before [31], [34], [27], [18], however, all based on global traces rather than local traces. Our own prior work CLAP [17] also uses thread-local reasoning to help reproducing bugs. However, the thread-local concept there is different. In [17], the thread-local control flow is captured to reconstruct a buggy schedule, whereas our thread-local tracing in this work is concerned with the model events (*not control flow*) for building the causal model.

Typestate-based concurrency bug detectors [38], [39] can also detect high-level program semantics bugs as typestate allows event abstraction. A key limitation is that typestate is non-parametric and only characterizes single-type or single-object properties. For example, the `UnsafeDatabaseAccess` property in Fig. 11 cannot be expressed with typestate. Moreover, existing detectors [38], [39] can often produce false alarms because they do not have a sound causal model. 2ndStrike [39] prunes false alarms via re-execution. However, it may miss real bugs due to the scheduling non-determinism.

Many runtime verification frameworks have been developed to detect program errors dynamically, such as JavaMOP [8], PQL [25], Tracematches [5], etc. Users of these frameworks can specify events and patterns to monitor. When a pattern is satisfied or violated at runtime, users can provide extra handlers to perform additional function, *e.g.*, to recover from bad states. Our technique is complimentary to runtime verification, as it can predict errors by inferring other feasible schedules.

Different from predicting property violations, a few techniques have also been proposed to enforce properties at runtime. Vaziri *et al.* [33] develop a language for defining data-centric synchronizations over high-level data race patterns, and generate code to obey the properties at runtime. In our prior work EnforceMOP [24], we have also developed a language and runtime system that allows the users to define and enforce general properties for multithreaded programs.

VII. CONCLUSION

We have presented GPredict, a new predictive trace analysis technique that works for high-level, more generic concurrency properties. Our technique advances the state of the art in three aspects: 1. We develop a general constraint model that enables uniformly reasoning about the causal ordering between high-level declarable events in concurrent program executions. 2. We provide an expressive language to specify and to predict generic property violations based on existing constraint solvers. 3. Our technique does not require a global trace of events as input but only the set of local traces by each thread. With GPredict, users can specify concurrency errors more expressively and at the same time predict these errors with much smaller runtime overhead. Our evaluation with GPredict on real world applications demonstrates the effectiveness and the performance of our technique.

Acknowledgments

We would like to thank our anonymous reviewers. This research was supported in part by the NSF grant CCF-1218605, the NSA grant H98230-10-C-0294, the DARPA HACMS program as SRI subcontract 19-000222, the Rockwell Collins contract 4504813093, and the (Romanian) SMIS-CSNR 602-12516 contract no. 161/15.06.2010.

References

[1] http://sourceforge.net/p/jfreechart/bugs/1051/.

[2] https://issues.apache.org/jira/browse/DERBY-2861.

[3] https://issues.apache.org/jira/browse/DERBY-3786.

[4] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6487638.

[5] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making trace monitors feasible. In *OOPSLA*, 2007.

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[7] Eric Bodden, Laurie Hendren, and Ondrej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP*, 2007.

[8] Feng Chen and Grigore Rosu. Mop: an efficient and generic runtime verification framework. In *OOPSLA*, 2007.

[9] Feng Chen and Grigore Rosu. Parametric and sliced causality. In *CAV*, 2007.

[10] Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. jPredictor: a predictive runtime analysis tool for Java. In *ICSE*, 2008.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[12] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *PLDI*, 2007.

[13] Azadeh Farzan, P. Madhusudan, Niloofar Razavi, and Francesco Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *FSE*, 2012.

[14] Cormac Flanagan and Stephen N. Freund. Redcard: Redundant check elimination for dynamic race detectors. In *ECOOP*, 2013.

[15] Jeff Huang and Charles Zhang. PECAN: Persuasive Prediction of Concurrency Access Anomalies. In *ISSTA*, 2011.

[16] Jeff Huang, Jinguo Zhou, and Charles Zhang. Scaling predictive analysis of concurrent programs by removing trace redundancy. *TOSEM*, 22(1), 2012.

[17] Jeff Huang, Charles Zhang, and Julian Dolby. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *PLDI*, 2013.

[18] Jeff Huang, Patrick Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. *PLDI*, 2014.

[19] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *FSE*, 2010.

[20] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, 2009.

[21] Zhifeng Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE*, 2010.

[22] Patrick Lam, Eric Bodden, and Laurie Hendren. The soot framework for Java program analysis: a retrospective, 2011.

[23] Yu Lin and Danny Dig. CHECK-THEN-ACT Misuse of Java Concurrent Collections. In *ICST*, 2013.

[24] Qingzhou Luo and Grigore Rosu. EnforceMOP: a runtime property enforcement system for multithreaded programs. In *ISSTA*, 2013.

[25] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, 2005.

[26] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.

[27] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. Generating data race witnesses by an SMT-based analysis. In *NFM*, 2011.

[28] Koushik Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[29] Koushik Sen, Grigore Rosu, and Gul Agha. Runtime safety analysis of multithreaded programs. *FSE*, 2003.

[30] Koushik Sen, Grigore Rosu, and Gul Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *FMOODS*, 2005.

[31] Traian Florin Serbanuta, Feng Chen, and Grigore Rosu. Maximal causal models for sequentially consistent systems. In *RV*, 2012.

[32] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *FSE*, 2010.

[33] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[34] Kahlon Vineet and Chao Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *CAV*, 2010.

[35] Chao Wang, Sudipta Kundu, Malay K. Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, 2009.

[36] Chao Wang, Rhishikesh Limaye, Malay K. Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In *TACAS*, 2010.

[37] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP*, 2006.

[38] Pallavi Joshi and Koushik Sen. Predictive Typestate Checking of Multithreaded Java Programs. In *ASE*, 2008.

[39] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndStrike: Toward Manifesting Hidden Concurrency Typestate Bugs. In *ASPLOS*, 2011.

[40] Chang-Seo Park, and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, 2008.