SHF:Small: Scalable and Maximal Predictive Runtime Verification for Concurrent Software

Project Description

1 Introduction

Detecting and repairing software defects, or bugs, are increasingly becoming extremely labor-intensive tasks, because of the rapidly increasing size and complexity of today's software. The situation is exacerbated by the increasingly widespread use of multicore processors, or central processing units, whose computing power can only be unleashed by concurrent software. Formal software verification, which consists of producing rigorous correctness proofs for formal models associated to target systems, is the highest-assurance method known. Unfortunately, in spite of all its desirable benefits and a half-century of research, formal program verification is still hard and almost inaccessible to most computer scientists and programmers.

Runtime verification is an approach that extracts information from a running system and uses it to detect and possibly react to observed behaviors satisfying or violating certain properties, which can be formally specified in terms of trace predicate formalisms, such as finite state machines, context-free patterns, linear temporal logics, etc, and checked at runtime. Because runtime verification works directly with the concrete executions of the actual system and can correct errors on-the-fly, it is a promising and practical alternative to improving the reliability and security of concurrent software, without incurring the complexity of traditional formal verification techniques such as model checking and theorem proving.

This project aims to develop a scalable predictive runtime verification framework for concurrent software running on commodity hardware. A major technical advancement of this framework over prior art is that it not only detects errors when they occur at runtime, but is able to predict general security and safety property violations before they actually surface to bite, preventing bad behaviors from happening by taking proper actions defined by the users. Moreover, our approach builds upon a *sound* and *maximal* causal model, hereby providing the provably maximum possible prediction power on the observed trace with no false alarms. The core research insight of this work is to explore the maximal causality of concurrency with *automated constraint solving*, which has been studied for decades and is becoming increasingly powerful due to the recent advances of theorem provers and decision procedures. This way, the proposed framework will offer the best possible predictive runtime analysis technology for improving the quality of real world multicore computing systems.

1.1 Motivation

In order to explain the technique we will use to detect general security and safety property violations, we will describe the technique in terms of data races, which are, themselves, an extremely important class of concurrency bugs that have caused some of the worst concurrency problems in multithreaded systems today.

A data race occurs when there are unordered conflicting accesses in the program without proper synchrornization. Consider, for instance, an execution of the program in Figure 1. The program contains a race between lines (`3,10`) that may cause an authentication failure of resource z at line `12`, which in consequence causes an error to occur when z is used at line `15`. Data races are particularly problematic because they manifest non-deterministically, often appearing only on very rare executions, making them notoriously difficult to test and debug.

Although researchers have proposed a wide spectrum of techniques [5, 12, 17, 19, 23, 25, 32, 38, 42, 43, 50] to combat races, existing techniques either suffer from unsoundness (i.e., report false alarms) or have a limited detection capability. The school of *lockset*-based techniques [19, 25, 38, 43] popularized by Eraser [43] is known to be unsound, whereas the *happens-before* (HB) based approaches [5, 17, 23, 32] are often very limited in detecting races, due to extra overly conservative HB edges. Additionally, it should be noted that these techniques focus *only* on data race detection, rather than building a model usable with other types of properties.

Even though a recent development, causally-precedes (CP) [50], improves the detection power by soundly relaxing the HB edges between critical sections that have no conflicting accesses, it can still miss many races. For example, in Figure 1, supposing the execution follows an order denoted by the line numbers, CP cannot detect the race (3, 10) since line 3 causally-precedes line 10, for the reason that the two lock regions contain conflicting accesses to *y*. PECAN [25], another representative technique that uses a hybrid algorithm combining lockset and a weaker form of HB, is able to detect this race by ignoring the HB edges between critical regions. However, the hybrid algorithm is unsound in general. For instance, if we switch lines 1 and 2, (3,10) is no longer a race (because then line 10 will always happen-after line 3), but PECAN will still report it.

```
     initially   x=y=0      resource z=0

  Thread t1                  Thread t2
 1.  fork t2
 2.  lock l
 3.  x = 1
 4.  y = 1
 5.  unlock l
                            6.  { //begin
                            7.    lock l
                            8.    r1 = y
                            9.    unlock l
                            10.   r2 = x
                            11.   if(r1 == r2)
                            12.       z = 1    (auth)
                            13. } //end
 14. join t2
 15. r3 = z    (use)
 16. if(r3 == 0)
 17.     Error
```

Figure 1: An program with a race (3,10).

1.2 Our Approach

The framework we propose in this project will be the first sound causal model construction technique that is maximal. What we mean by maximal is that there is no way to predict more thread interleavings from a single execution that is sound (i.e., does not produce false positives). A key observation of our technique is that the control flow information between events in the execution (which is often ignored by existing techniques) can help significantly improve the error prediction power of the causal model.

Consider the simple scenario in Figure 2 where line 3 has two cases: *r*1 = *y* and *while*(*y* == 0). For case ①, (1,4) is a race; while for case ②, it is not, because line 4 is control-dependent on the while loop at line 3. However, without considering the control dependence between operations, the dynamic execution traces for these two cases are identical (both following lines 1-2-3-4). Hence, a sound technique must conservatively assume that a value read by a thread influences all subsequent values produced by the same thread, which, in consequence, creates a HB edge from line 2 to line 3 and misses the race in case ①. However, with the control flow information, we can tell that, in case ①, line 4 is not control-dependent on line 3. In other words, regardless of what value line 3 reads, line 4 will always be executed. Therefore, we can safely drop the HB edge from line 2 to line 3, which enables detecting the race (1,4). Similarly, we are able to detect the race (3,10) in Figure 1 by dropping the HB edge from line 4 to line 8, because there is no control flow from line 8 to line 10 and hence no need to ensure line 8 should read value 1 (written by line 4).

```
        initially   x=y=0

   Thread t1       Thread t2

 1.  x = 1
 2.  y = 1
                  3. ① r1 = y   ② while(y == 0);
                  4.   r2 = x
```

Figure 2: The two cases ① and ② produce the same read/write trace. However, (1,4) is a race only in case ①.

We introduce a new type of events—*branch*—into the execution model. Observing branch events is cheap at runtime, however, it provides an abstract view of the control flow information between events that enables a higher race detection power. Moreover, inspired by the theoretical maximal causal model [48], we develop a weaker maximal causal model that incorporates control flow information under the sequential consistency memory model. Underpinned by the new model, race detection can be formulated as a constraint solving problem by encoding the control flow and all the valid trace reorderings as a set of first-order logic constraints. Specifically, we introduce an order variable O for each event in the trace such that O_a represents the order of an event a in a possible feasible schedule. Clearly, if two events a and b are conflicting and $O_a = O_b$, then (a,b) is a data race. Let the line number denote the corresponding event. Figure 3 shows the trace corresponding to our example in Figure 1.

We further model the schedule feasibility as a set of constraints over the order variables, and employ a solver to solve them. If the solver returns a solution, (a,b) is a data race; otherwise, not. Figure 4 shows our constraint modeling of the example trace. Let O_i refer to the order variable of the event at line i. The constraints consist of three parts: (A) the must happen-before (MHB) constraints, (B) the locking constraints, and (C) the race constraints. A and B are common for all races, whereas C is race specific. For instance, the MHB constraints for the *fork* event at line 1 and the *join* event at line 14 are written as $O_1 < O_6 \wedge O_{14} > O_{13}$, meaning that the *fork* event should happen before the *begin* event of $t2$ at line 6, and the *join* event should happen after the *end* event of $t2$ at line 13, which are determined by the must happen-before relation. The locking constraints encode lock mutual exclusion consistency over *acquire* and *release* events. For example, $O_5 < O_7 \vee O_9 < O_2$ means that either $t1$ acquires the lock l first and $t2$ second, or $t2$ acquires l first and $t1$ second. If $t1$ first, then the *acquire* at line 7 must happen after the *release* at line 5; otherwise if $t2$ first, the *acquire* at line 2 should happen after the *release* at line 9.

```
initially   x = y = z = 0

 1. fork(t1, t2)
 2. acquire(t1, l)
 3. write(t1, x, 1)
 4. write(t1, y, 1)
 5. release(t1, l)
                              6. begin(t2)
                              7. acquire(t2, l)
                              8. read(t2, y, 1)
                              9. release(t2, l)
                             10. read(t2, x, 1)
                             11. branch(t2)
                             12. write(t2, z, 1)
                             13. end(t2)
14. join(t1, t2)
15. read(t1, z, 1)
16. branch(t1)
```

Figure 3: Trace of example in Figure 1.

A. MHB (Φ_{mhb}) $\quad O_1 < O_2 < \ldots < O_5 \quad O_{14} < \ldots < O_{16}$; $\quad O_6 < O_7 < \ldots < O_{13}$; $\quad O_1 < O_6 \wedge O_{13} < O_{14}$

B. Locking (Φ_{lock}) $\quad O_5 < O_7 \vee O_9 < O_2$

C. (3,10) Race (Φ_{race}) $\quad O_{10} - O_3 = 1$

C. (12,15) Race (Φ_{race}) $\quad O_{15} - O_{12} = 1$; $\quad O_3 < O_{10} \wedge O_4 < O_8$

Figure 4: Constraint modeling of the example trace in Figure 3.

The race constraints encode the race and control flow conditions specific to each conflicting pair of accesses by different threads. For example, for (3,10), the race constraint is written as $O_{10} - O_3 = 1$, and its control-flow condition is empty, because there is no *branch* event before the two events at lines 3 and 10. For (12,15), however, because there is a *branch* event (at line 11) before line 12, in addition to the race constraint $O_{15} - O_{12} = 1$, we need to ensure that the control-flow condition at the *branch* event is satisfied. Our control-flow constraint requires that all *read* events by $t2$ before this *branch* event read the same value as that in the original trace. Hence, we add the control-flow constraints $O_3 < O_{10} \wedge O_4 < O_8$ to ensure that the *read* event at line 10 reads value 1 on x, and that the *read* event at line 8 reads value 1 on y. This guarantees that the event at line 12 is feasible. Putting all these constraints together, we invoke an SMT solver (e.g.,Z3 [15], CVC [2], Yices [16]) to compute a solution for these unknown order variables. For (3,10), the solver returns a solution which corresponds to the schedule 1-6-7-8-9-2-3-10, so (3,10) is a race. For (12,15), the solver reports no solution exists, so it is not a race.

Although we exemplified our approach using data-races, the same idea extends to predicting violations of any properties that can be formally specified as sets of traces (using for example finite state machines, regular expressions, temporal logics, context-free grammars, and so on).

1.3 Proposed Work

This project will build upon the existing research described above by making the following contributions:

Predictive Runtime Verification of Generic Properties Developing a predictive runtime verification and analysis framework that is able to handle *generic* properties in concurrent software. This framework will predictively verify both functional correctness of programs as well as verify that necessary security policies are followed. In addition, the prototype described above only handles data races, but the general methodology is extensible to other common concurrency errors such as atomicity violations and deadlocks.

Expressive Property Languages and Property Database Designing an expressive and concurrency-aware specification language for writing the general properties. Property specification languages in most existing runtime verification frameworks such as MOP [10] are not designed for concurrency. In this project, we will design a new language for expressing generic concurrency properties based on MOP and integrate it into our framework. Moreover, formalizing properties is a non-trivial task that will take up a large amount of time for engineers using our tool. We plan to provide a large database of safety and security properties for different programming languages (Java, C, C++), to remove much of this barrier to entry.

Relaxed-Memory Concurrency Developing predictive runtime verification techniques for relaxed memory model-based concurrent software. To improve performance, virtually all modern processors (e.g. x86, Power, SPARC, ARM, Itanium) and programming languages (C, C++, Java) do not provide the sequentially consistent shared memory, but subtle relaxed (or weak) memory models, exposing behavior that arises from hardware and compiler optimizations to the programmer. Our goal is to also develop usable techniques for these. We will focus on three processor architectures (x86, Power, and ARM), on the recent revisions of the C++ and C languages (C++11 and C11), and on predictive runtime verification upon these models.

Scalable Constraint Solving Developing a specialized and highly-efficient constraint solving component in the framework. A core part in our proposed framework is to reason about concurrency with constraint solving. The state of the art SMT solvers such as Z3, CVC3, and Yices are fast and powerful with numerous decision procedures for different constraint-solving problems. Nevertheless, no existing solver is optimal for our problem because there is no off-the-shelf procedure targeting specifically at our constraints, which consists of simple but large partial orders. Our goal is to develop customized constraint solvers.

1.4 Related Work

There is a rich body of predictive analysis and runtime verification work in the literature. Fundamentally, the proposed work distinguishes in two ways: **(1)** It addresses the prediction problem of high-level expressive generic properties in concurrent software in addition to low-level concurrency errors such as races and atomicity violations. Moreover, our framework is not only theoretically sound but also practically feasible. The techniques we develop works for shared-memory concurrency in commodity hardware with mainstream programming languages and its scalability is further boosted by high-performance customized solvers. **(2)** Our core methodology of reasoning about concurrency with constraint solving is based on a theoretical foundation of sound and maximal causal model. Moreover, we introduce novel control flow information to extend the classical execution model, which achieves a provably higher prediction power than prior art.

Predictive Trace Analysis Sen et al. [45, 46] present one of the first predictive analysis techniques for detecting violations of safety properties in concurrent programs. The approach extracts an abstract model of the execution based on happens-before and traverses the computation lattice to find errors. However, due to the complexity of the generated models, it was not applied in real world applications. jPredictor [12] present a predictive analysis framework that predicts races and atomicity violations based on sliced causality [8], which is a sound causal model concerning precise data and control dependencies. Differently, jPredictor requires expensive static dependence analysis (hard to implement soundly in practice) and it is non-maximal. Goldilock [19] and O'Callahan et al. [38] use a hybrid model that combines happens-before and the lock-based approaches to predict races based on an execution. Penelope [51] presents a testing framework for predicting and exposing atomicity violations in concurrent programs. PECAN [25] proposes a predictive analysis algorithm for a school of concurrency access anomalies. In addition, it generates bug-manifesting schedules that can be enforced to validate the predicted anomalies. Lai et al. [28] combines predictive analysis with randomized active testing [44] for detecting atomic-set serializability violations. A major difference between these works and our proposed approach is that their algorithms are tailored to specific errors.

Logical Constraint Solving has also been proposed to increase the predictive analysis coverage. Wang et al. [42, 53, 55, 56] develop a series of symbolic analyses for finding concurrency errors based on the execution trace and constraint solving. A representative technique in this line is the work by Said et al. [42], which also relies on efficient encoding of the trace constraints and modern SMT solvers to explore feasible reorderings. Unlike our approach, [42] does not consider control flow and it is non-maximal. To ensure soundness, [42] requires *full read-write consistency*: every read returns the same value as that in the original trace. However, this requirement limits the race exploration to only a subset of the feasible traces. For example, [42] cannot detect the race (`3`,`10`) in Figure 1, as line `10` can only read value 1 on *x* written by line `3`. Instead, our technique is concerned with the read-write consistency from the perspective of control dependence, and generates only the constraints with respect to the events that have control flow to the race related operations, hence, to detect races in all feasible incomplete traces as well. ExceptioNULL [22] develops another technique that predicts null-pointer dereferences using constraint solving. Similar to [42], it encodes the full data-validity constraints and does not achieve maximality. Another key difference of our work is that we generalize the properties to any trace formal specifications, and thus are not limited to data races or null-pointer exceptions.

Runtime Verification Many runtime verification frameworks have been developed to detect program errors dynamically, such as JavaMOP [10], PQL [33], Tracematches [1], etc. Users of these frameworks can specify events as well as event patterns to monitor at runtime. Moreover, users can provide additional code to be executed once a pattern is satisfied or violated, e.g., to recover from bad states. To define and enforce synchronization properties for multithreaded programs, Vaziri et al. [52] propose a number of patterns for high level races and a language for defining data-centric synchronization constructs. Moreover, they also generate code to obey those constraints at runtime. Luo and Rosu [31] propose another framework to define and enforce general properties in multithreaded programs. In contrast, our proposed framework empowers runtime verification with the ability to predict property violations, by reasoning about the causality of concurrent program executions to infer the maximal set of feasible schedules based on an observed trace.

Runtime Error Detection and Model Checking Runtime techniques are typically designed for efficiency and do not perform a comprehensive exploration of feasible trace permutations. Numerous tools have been proposed to detect concurrency errors dynamically with improved performance. For example, FastTrack [23] proposes an adaptive lightweight happens-before representation for detecting races, and IFRit [17] uses static analysis to identify interference-free regions that reduce redundant instrumentation at compile time. Pacer [5], LiteRace [32], and DataCollider [20] use sampling-based approaches to detect races with negligible runtime overheads. An alternative way to achieve maximality in predictive analysis is to exhaustively explore the thread scheduling space, employed by model checking techniques [37, 49, 54]. For instance, CHESS dynamically explores different thread schedules of the target program in a context-bounded way. Shacham et al. [49] use a model checker to construct the witness for races reported by the lockset algorithm. Unfortunately, facing the exponentiality of both the program path and the scheduling space, it is still hard for model checking techniques to scale to large multithreaded programs. As our technique focuses on exploring schedules with respect to a single dynamic trace, it is much more scalable than model checking.

2 Approach

Here we describe our approach and current prototype for constructing sound causal models from a single execution trace. Although highly experimental, our current prototype brings confidence that our goals in this project can be achieved.

2.1 Maximal Causal Model

$$\begin{array}{rclcl}
\mathit{Event} & ::= & \mathit{begin}(t) & \mid & \mathit{end}(t) \\
 & \mid & \mathit{write}(t,x,v) & \mid & \mathit{read}(t,x,v) \\
 & \mid & \mathit{acquire}(t,l) & \mid & \mathit{release}(t,l) \\
 & \mid & \mathit{fork}(t,t') & \mid & \mathit{join}(t,t') \\
 & \mid & \mathit{branch}(\mathrm{t}) & \mid & \mathit{property}(t,p)
\end{array}$$

$$t,t' \in \mathit{Thread};\ x \in \mathit{Variable};\ l \in \mathit{Lock}$$
$$v \in \mathit{Value};\ p \in \mathit{Property};$$

Figure 5: Event types

Our approach builds upon the *maximal causal model* (MCM) [48] foundation for sequential consistency. In MCM, multithreaded programs $\mathcal{P}$ are abstracted as the prefix-closed sets of finite traces that they can produce when completely or partially executed, called *$\mathcal{P}$-feasible* traces. A *trace* is abstracted as a sequence of events. *Events* are operations performed by threads on concurrent objects such as variables, locks, signals, or threads, abstracted as tuples of *attribute-value* pairs. For example, $(\mathit{thread}=t_1, \mathit{op}=\mathit{read}, \mathit{target}=x, \mathit{data}=1)$ is a read event by thread t_1 to memory location x with value 1. Additionally, we must augment this with the events our security and safety properties refer to. These can be anything, e.g., a property regarding Iterators may have calls to the next() and hasNext() methods as property events. Figure 5 shows an overview of these operations (by a thread t):

- $\mathit{begin}(t)$ / $\mathit{end}(t)$: the first/last event of thread t;
- $\mathit{read}(t,x,v)$ / $\mathit{write}(t,x,v)$: read/write a value v on a variable x;
- $\mathit{acquire}(t,l)$ / $\mathit{release}(t,l)$: acquire/release a lock l;
- $\mathit{fork}(t,t')$: fork a new thread t';
- $\mathit{join}(t,t')$: block until thread t' terminates;
- $\mathit{branch}(t)$: determine the control flow according to the value of a certain variable.
- $\mathit{property}(t,p)$: property event in a thread t with property information p.

A notable difference in our model from prior work is the inclusion of a new branch operation $\mathit{branch}(t)$. This new operation serves as a guard of a possible control flow change, which determines what the next operation to execute in a thread is. The choice of the control flow depends on the value of a certain variable which we do not know. It is a computation result local to the operating thread and is not visible to other threads. Therefore, conservatively, we assume that the choice of $\mathit{branch}(t)$ depends on all the previous $\mathit{read}(t,x,v)$ operations executed by the same thread. For the rest of the discussion we will focus purely on race detection, because general property prediction algorithms are not yet completed in out current prototype.

The sets of $\mathcal{P}$-feasible traces must obey some basic consistency axioms. We proposed two axioms in [48]: *prefix closedness* and *local determinism*. The former says that the prefixes of a $\mathcal{P}$-feasible trace are also $\mathcal{P}$-feasible. The latter says that each thread has a deterministic behavior, that is, only the previous events of a thread (and not other events of other threads) determine the next event of the thread, although if that event is a read then it is allowed to get its value from the latest write. These two axioms allow us to associate a causal model $\mathit{feasible}(\tau)$ to any consistent trace τ, which comprises precisely the traces that can be generated by any program that can generate τ. As shown in [48], $\mathit{feasible}(\tau)$ is both *sound* and *maximal*: any program which can generate τ can also generate all traces in $\mathit{feasible}(\tau)$, and for any trace τ' not in $\mathit{feasible}(\tau)$ there exists a program generating τ which cannot generate τ'. Comparatively, conventional happens-before causal models consisting of all the legal interleavings of τ and their prefixes are *not* maximal [48].

2.2 Constraint Modeling

In our approach, we realize MCM using constraints and represent *feasible*(τ) by a formula Φ of first order logic clauses over a set of order variables, each of which corresponds to an event in τ. Any solution to Φ denotes a legal schedule that can produce a corresponding trace in *feasible*(τ). From a high level view, Φ contains only variables of the form O_e corresponding to events e, which denote the order of the events in a trace in *feasible*(τ). Φ is constructed by a conjunction of three sub-formulas: $\Phi = \Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi_{rw}$.

***Must happen-before constraints* (Φ_{mhb})** The must happen-before (MHB) constraints require that (1) the total orders of the events in each thread are always the same; (2) a *begin* event can happen only as a first event in a thread and only after the thread is forked by another thread; (3) an *end* event can happen only as the last event in a thread, and a *join* event can happen only after the *end* event of the joined thread. MHB yields an obvious partial order $\prec$ on the events of τ which must be respected by any trace in *feasible*(τ). We can specify $\prec$ easily as constraints over the O variables: we start with $\Phi_{mhb} \equiv$ *true* and conjunct it with a constraint $O_{e_1} < O_{e_2}$ whenever e_1 and e_2 are events by the same thread and e_1 occurs before e_2, or when e_1 is an event of the form *fork*(t, t') and e_2 of the form *begin*(t'), etc.

***Locking Constraints* (Φ_{lock})** Lock mutual exclusion semantics requires that two sequences of events protected by the same lock do not interleave. Φ_{lock} captures the ordering constraints over the lock *acquire* and *release* events. For each lock l, we extract the set S_l of all the corresponding pairs (a, r) of *acquire*/*release* events on l, following the program order locking semantics: the *release* is paired with the most recent *acquire* on the same lock by the same thread. Then we conjunct Φ_{lock} with the formula

$$\bigwedge_{(a,r),(a',r')\in S_l} (O_r < O_{a'} \vee O_{r'} < O_a)$$

***Read-write constraints* (Φ_{rw})** The read-write constraints ensure that every event in the trace is feasible. For an event to be feasible, all the events that must happen-before it should also be feasible. Moreover, any read event that must happen-before it should read the *same value* as that in the original trace. Consider a read event r, say *read*(t, x, v), we let W^r be the set of *write*$(\_, x, \_)$ events in τ (here '_' denotes any value), and W^r_v the set of *write*$(\_, x, v)$ events in τ, then we have the formula defining its feasibility as following:

$$\Phi_{rw}(r) = \bigvee_{w\in W^r_v} (\Phi_{rw}(w) \wedge O_w < O_r \bigwedge_{w\neq w'\in W^r} (O_{w'} < O_w \vee O_r < O_{w'}))$$

The above states that the read event $r = read(t, x, v)$ may read the value v on x written by any *write* event $w = write(\_, x, v)$ in W^r_v (the top disjunction), subject to the condition that the order of w is smaller than that of r and there is no interfering *write*$(\_, x, \_)$ in between. Moreover, w itself must be concretely feasible, which is ensured by $\Phi_{rw}(w)$. Similarly, $\Phi_{rw}(w)$ is defined by requiring all the reads that must happen-before it are feasible. Φ_{rw} is a conjunction of $\Phi_{rw}(r)$ for all reads in the considered trace.

2.3 Current Prototype

To preliminarily assess the feasibility and impact of our framework, we have implemented a prototype system that we call RVPredict for predicting data races in Java programs. RVPredict first logs a sequentially consistent trace online, including shared data accesses, thread synchronizations, and branch events, and then performs offline predictive race analysis. To support long running programs, traces are first stored event by event into a database. Note that trace logging can be performed at various levels, e.g., via static or dynamic code instrumentation, inside the VM, or at the hardware level. Our implementation is based on static instrumentation and is not optimized. Nevertheless, ideally, we can use hardware tracing techniques to minimize the runtime perturbation. In predictive race analysis, we first use a hybrid lockset and weaker HB algorithm (similar to PECAN [25]) to perform a quick check on each conflicting operation pair (COP). Only if a COP passes the quick check, do we proceed to build constraints for it. We set the default constraint solving time to one minute for each COP. If the solver returns a solution within one minute, we report a race. In addition, to avoid redundant computation on races that have the same signature (from the same program locations), once a COP is reported as a race, we prune away all the other COPs with the same signature with no further analysis.

Handling long traces From an engineering perspective, handling long traces is challenging for any race detection technique. For real world applications, the trace is often too large to fit into the main memory. Moreover, for our approach, the generated constraints for long traces can be difficult to solve. Even with a high performance solver like Z3 or Yices, the constraints may still be too heavy to solve in a reasonable time budget. For practicality, we employ in RVPredict a windowing strategy similar to CP [50]. We divide the trace into a sequence of fixed-size windows (typically 10K events in a window) and perform race analysis on each window separately. This simple strategy has two advantages for performance optimization: First, each time only a window size of events are processed, which can be easily loaded in memory. Second, the generated constraints for a window instead of the whole trace become much smaller, so that Z3 and Yices can solve them much easier. The downside of this strategy is that a race between operations in different windows will not be detected. However, this windowing strategy does not affect the soundness of our implementation. All detected races by RVPredict are real, i.e., it does not report any false positive.

2.3.1 Prototype Evaluation

To properly compare our technique with the state-of-the-art, we have also implemented HB [29], CP [50], and Said *et al.* [42] in RVPredict. We compared our technique with these techniques on an extensive collection of widely used multithreaded benchmarks as well as several real world large concurrent systems, mostly from previous studies [3, 12, 21, 25, 42, 50, 51]. To perform a fair comparison, for each benchmark, we collected one trace and ran different techniques on the same trace.

All experiments were conducted on a 8 hardware thread 3.50GHz Intel i7 machine with 32G memory and Linux version 3.2.0. The VM configuration is OpenJDK 64-Bit Server VM with version 1.7.0 and 32G heap space. For all benchmarks, we set the window size to 10K and the timeout threshold of Z3 to 5 minutes. We next discuss our experimental results in detail as reported in Table 1.

Benchmarks and Trace Statistics Columns 1-4 list our benchmarks and metrics of the corresponding traces. The total source lines of code of these programs is more than 1.7M. The first row shows our example program in Figure 1; the second set of small benchmarks are from IBM Contest benchmark suite [21]; the third set contains three popular multithreaded Java Grande benchmarks; the last set contains real world large applications. The most substantial real systems include:

- FTPServer - Apache's high-performance FTP server;
- Jigsaw - W3C's web server;
- Derby - Apache's widely used open source Java RDBMS;
- Sunflow, Xalan, Lusearch, Eclipse - multithreaded applications from Dacapo benchmark suite 9.12 [3].

2.3.2 Race Detection Capability and Scalability

Column 5 reports the number of potential races that pass the quick check of a hybrid lockset and weaker HB algorithm. These races comprise a superset of all the real races that can be detected from the trace. Because the hybrid algorithm is unsound, some races in this set may be false positives. For instance, there

1	2	3	4	5	6	7	8	9	10	11	12	13
Program	**LOC**	**#Thrd**	**#Event**	**Data race**					**Time**			
				QC	**RV**	Said	CP	HB	**RV**	Said	CP	HB
example	64	2	20	1	**1**	0	0	0	0.4s	0.8s	0.8s	0.8s
critical	63	3	38	8	**8**	7	4	4	0.9s	0.8s	0.8s	0.8s
airline	83	11	214	9	**9**	6	8	8	0.9s	1.0s	0.8s	0.8s
account	87	3	126	9	**5**	5	3	3	0.9s	0.9s	0.9s	0.9s
pingpong	124	5	64	7	**4**	4	3	3	0.8s	0.8s	0.8s	0.8s
bbuffer	334	4	1.7K	15	**13**	10	5	5	4.1s	6.8s	1s	1s
bubblesort	274	26	4.5K	17	**8**	8	7	7	7.4s	44s	1.3s	1.2s
bufwriter	199	5	411	18	**2**	0	2	2	1s	1.2s	0.8s	0.8s
mergesort	298	5	5.5K	16	**9**	7	3	3	2.5s	7.9s	1.2s	1.2s
raytracer		2	23.4K	5	**5**	5	3	3	1.2s	3.4s	1.2s	1.2s
montecarlo	2.9K	2	11.7M	0	**0**	0	0	0	13.5s	13.7s	14.2s	12s
moldyn		2	273K	509	**11**	5	2	2	53.5s	170s	1.2s	1.6s
Total: bench	4.3K	-		614	**75**	57	40	40	-			
ftpserver	32K	12	59K	233	**37**	3	31	27	34s	96s	1.6s	1.3s
jigsaw	101K	12	4.6M	54	**15**	14	7	7	81s	92s	6.2s	5.9s
derby	302K	3	856K	469	**118**	15	14	12	1820s	1h	24s	23s
sunflow	109K	9	14.8M	66	**2**	2	2	2	14.6s	14.5s	14.4s	14.9s
xalan	180K	9	12.5M	108	**108**	107	5	3	14.6s	14.6s	13.5s	12.5s
lusearch	410K	10	11.3M	87	**16**	15	15	15	12.1s	12.8s	12.3s	11.9s
eclipse	560K	10	14.3M	8	**3**	2	2	2	17.3s	23.6s	37.2s	14.5s
Total: real	1.7M	-		1025	**299**	158	76	68	-			

Table 1: Overall results - Columns 3-4 report the number of threads (*#Thrd*) and total number of events (*#Event*)) in the traces. Column 5 (*QC*) reports the number of potential races that passes the quick check algorithm (a hybrid of lockset and weak HB). Columns 6-9 reports the number of real races detected by our technique (*RV*), Said *et al.* [42] (*Said*), Causally-Precedes [50] (*CP*), and Happens-Before [29] (*HB*), respectively, and Columns 10-13 report the total race detection time taken by the corresponding techniques. For all the evaluated programs, our technique detected more or at least the same number of races as the other techniques. For the efficiency, HB and CP are faster than the other two, and our technique is faster than Said.

are 18 potential races detected in *bufwriter*, but only 2 of them are real races. Columns 6-9 report the number of real races detected by different sound techniques.

The results show that, for every benchmark, our technique is able to detect more or at least the same number of races (i.e., a super set) as the other sound techniques. For instance, for *derby*, our technique (*RV*) detected 118 races, while Said *et al.* detected 15, CP detected 14, and HB detected 12. This demonstrates that our technique achieves a higher race detection capability not only theoretically, but also in practice. For Said *et al.*, it detected more races than HB and CP in most benchmarks, with a few exceptions, though. For instance, for *ftpserver*, CP and HB detected 31 and 27 races, respectively, whereas Said *et al.* only detected 3. The reason for this, as explained in Section 1, is that the all read-write consistency prevents Said *et al.* from detecting races in feasible incomplete traces, though its SMT-based solution is able to explore more valid whole trace re-orderings than CP and HB. Between CP and HB, they detected the same number of races in the small benchmarks. This was because the lock regions in these small benchmarks typically have conflicting accesses. However, this does not hold for the real systems. In *ftpserver*, *derby*, and *xalan*, CP detected a few more races than HB.

For the real systems, our technique detected a total number of 299 real races. Notably, among these races, a number of them are previously unknown. For instance, we found three real races in *eclipse*, one is on the field variable *activeSL* of class *org.eclipse.osgi.framework.internal.core.*

StartLevelManager, and the other two happen on the field *elementCount* of class *org.eclipse.osgi.framework.util.KeyedHashSet*. Interestingly, *KeyedHashSet* is documented as thread unsafe. The Eclipse developers misused this class and created a shared instance by multiple threads without external synchronization. Shortly after we reported these races, the developers fixed them and also contacted us for adopting our tool. Now the team is using RVPredict to detect races in the codebase of Virgo. We also found eight previously unknown races in *lusearch*, which happen in the class *org.apache.lucene.queryParser.QueryParserTokenManager*. We first reported these races in the *lucene* bug database. However, the developer pointed out that *QueryParserTokenManager* is documented as thread unsafe. It turned out that this class was misused by the Dacapo developers in writing the *lusearch* benchmark.

Scalability With the high performance SMT solvers and our windowing strategy, our technique shows good scalability when dealing with large traces. Column 10 reports the total time for our technique to detect races in each program using Yices. The performance of Z3 was comparable with only slight variances. For most small benchmarks, our technique was able to finish in a few seconds. For most real systems, our technique finished within around a minute. The most time consuming case is *derby*, which our technique took around 30 minutes to process. The reason is that the trace of *derby* has a lot more potential races (469 COPs) and also it contains many fine-grained critical sections (38K synchronizations), making the generated constraints much more complex.

Columns 11-13 report the race detection time for the other three techniques. Among the four techniques (including ours), HB and CP are comparable and are typically faster than Said *et al.* and our technique. This is expected because HB and CP do not rely on SMT solving and explore a much smaller set of trace re-orderings. Between our technique and Said *et al.*, our technique typically has better performance. For instance, for the *derby* trace, Said *et al.* took more than one hour (timeout) without finishing, while our technique finished within around 30 minutes. The reason is that our technique generates less constraints to solve than Said *et al.* for capturing the read-write consistency. While Said *et al.* generate constraints for all read events in the trace to ensure the whole trace read-write consistency, our technique concerns only the read events that have control flow to the race events.

Note that our technique is sound and fully automatic. Unlike many unsound techniques that report false warnings or even sound techniques that require manual post-processing for most races (e.g., CP [50]), every race detected by our technique is real. This has been supported by our manual inspection: every reported race has been checked and confirmed to be real. On the other hand, because the maximality of our technique is concerned with sound race detection only, it is possible that our technique may miss some real races that can be reported by an unsound race detector. For example, not all the potential races reported in Column 5 are necessarily false alarms if they are not reported in Column 6 as well. However, if such a race exists, our technique guarantees that it cannot be reported by any sound technique using the same input trace. We also note that the race detection result of our technique (actually, of any dynamic technique) is sensitive to the observed execution trace. The results for different traces are incomparable. Therefore, it is possible for our technique to miss certain races reported in other studies, because the traces in our experiments may be different from those used in other's work. These results will generalize to arbitrary safety and security properties.

3 Proposed Work

The major objective of this project is to develop a scalable predictive runtime verification framework that is able to predict violations of general purpose properties in concurrent software at runtime, to detect concurrency errors earlier, to prevent bad behaviors from actually happening, and finally to improve the reliability and security of modern computer systems. The foundation of this framework is a sound and maximal causal model that exploits the maximal causality of concurrent software execution, providing the maximal possible predic-

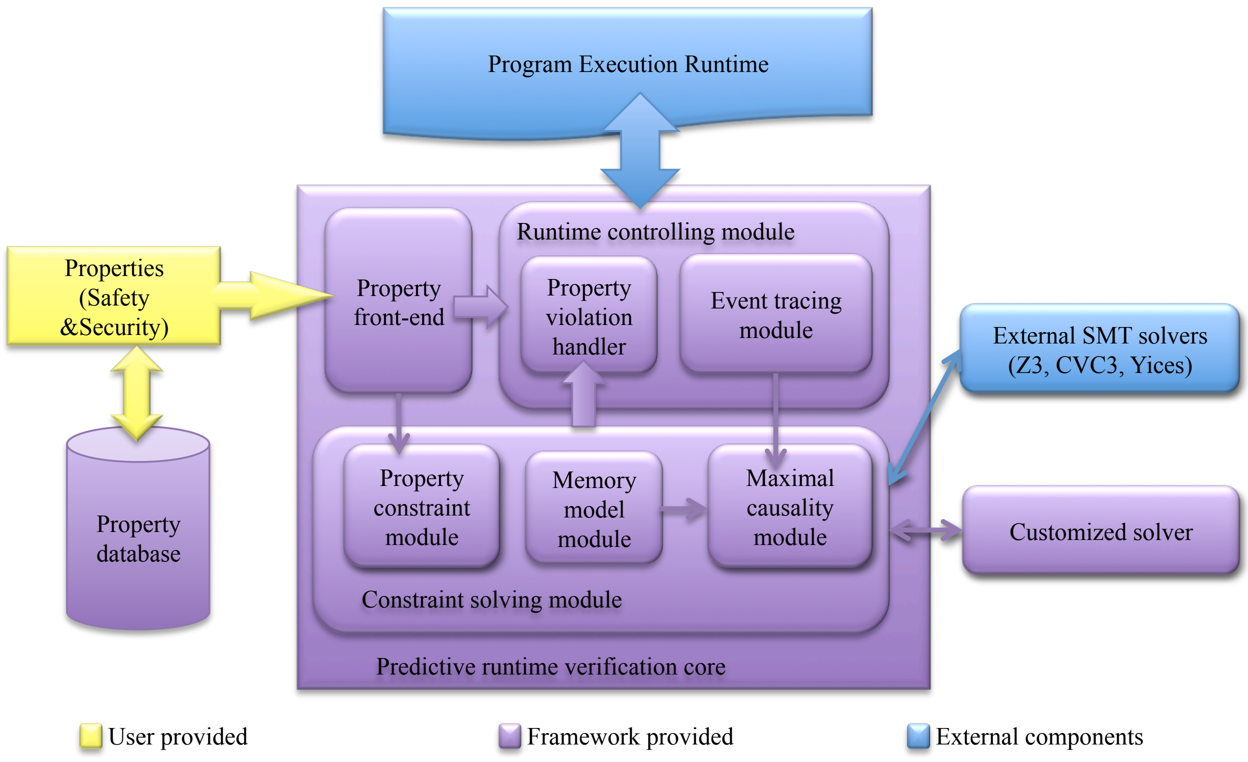
Program Execution Runtime
Properties (Safety &Security)
Property database
Property front-end
Runtime controlling module
Property violation handler
Event tracing module
External SMT solvers (Z3, CVC3, Yices)
Property constraint module
Memory model module
Maximal causality module
Customized solver
Constraint solving module
Predictive runtime verification core
User provided
Framework provided
External components

Figure 6: Architecture of the Predictive Runtime Verification Framework

tion power with no false alarms. The concrete realization of the maximal causal model is achieved through the development of automated constraint solving, resulting in a powerful tool for reasoning about concurrency.

Figure 6 depicts the proposed architecture of such a system. The framework itself consists of three main components: the property front-end, the runtime controller, and the constraint solving module.

The runtime controller consists of an event tracing module (that observes critical events in the execution and emits them to the constraint solving module for constructing the constraints) and a property violation handler (that interacts with both the constraint solving module and the external program execution runtime to execute property actions when property violations are predicted).

The constraint solving module itself consists of a property constraint model (that constructs property constraints), a memory model module (that plugins the memory model semantics into constraint construction), and the maximal causality module (that constructs the maximal causality constraints as modeled in Section 2.2). The constraint solving module conjuncts the property constraints and the maximal causality constraints and may generate queries to external theorem provers, like SMT solvers (Z3, CVC3, Yices), or invoke an internal customized solver (that is specialized in solving the causality constraints to maximize efficiency).

Finally, the property front-end analyzes the property specification and yields the interested property information (patterns and actions) to the runtime controller and to the property constraint module. The property specification is the only information provided by the user that should suit his or her needs in expressing general safety and security properties. In addition to design an expressive property specification language, to facilitate the property formalization, as part of this project we also plan to create a property database for reasoning about the most common properties encountered in concurrent programs. The property database may be

<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>)}

Figure 7: A property specification language (adding to MOP [10] the new, grayed syntactic constructs)

placed in a centralized server that will grow with the uses of this framework. Inferring properties of arbitrary concurrent software seems to be a very hard problem, which we will not attempt to solve as part of this project.

3.1 Proposed Work: A Predictive Runtime Verification Framework for Generic Properties

One of our main goals is to develop a predictive runtime verification framework that supports high-level generic concurrency properties. The plan is to engineer it in such a way that its user will only need to pass the desired properties of the concurrent runtime system to it, defined using a specification language, and then immediately be able to use it to predictively verify properties about the program. Moreover, whenever a property violation is predicted, before it actually occurs, the system will execute the corresponding actions (specified in the property handler) to prevent it or recover the system to a safer state defined by the user.

One crucial part of this framework is to support general properties. To understand the necessity and importance of generic properties, let us consider a simple resource `authenticate-before-use` idiom as example. The programmers may want a method `authenticate` to be always called before a method `use` that uses a resource. Any violation of this property is considered to be a serious security bug. However, it cannot be characterized by conventional data races, because there are no 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 (i.e., calling `next()` on the iterator), otherwise a `ConcurrentModificationException` will be thrown. Again, this property is neither a race nor an atomicity violation, but a more generic contract on the use of Java Iterators.

Our first plan is to design an expressive specification language for properties. We have considered a pattern specification language with the support of regular expressions (RegExp), and conducted preliminary research. We choose RegExp because it is very natural and convenient in reflecting the ordering relation between property events. Nevertheless, our foundamental technique should work with any formalisms whose properties or formulae can be monitored using finite-state machine monitors (e.g., linear temporal logic). Figure 7 shows the syntax of our current specification language, which will likely change. It is an extension of the MOP specification [10], 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.

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

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

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

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

Figure 8: Atomicity violation and data-race property specifications

To explicitly support concurrency related properties, we allow the event identifiers 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 of an atomic region can be specified by the <*Begin*> and <*End*> attributes. 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. We have experimented with this notation and were able to write specifications for various useful concurrency properties, which should not be difficult to translate into logical constraints. For example, Figure 8 shows how typical atomicity and data-race property violations can be specified.

3.2 Proposed Work: Predictive Runtime Verification for Relaxed Memory Models

The power of predictive runtime verification would not be fully unleashed unless it can work for relaxed memory models, because, for performance reasons, virtually all modern processors and programming languages expose relaxed memory behaviors (they are no longer sequentially consistent). However, relaxed memory models are much more subtle and complex to reason about. Consider, for example, the Total Store Order (TSO) memory model supported by both x86 and SPARK. It guarantees that the sequence in which store, FLUSH, and atomic load-store instructions appear in memory for a given processor is identical to the sequence in which they were issued by the processor. However, a store followed by a load may be completed out of program order. Therefore, algorithms relying on rigid program order (such as the Dekker's mutual exclusion) which work under the sequentially consistent memory model will no longer work under TSO.

Since the relaxed memory model semantics differ on different shared memory architectures, there is a wide variety of existing multi-processors (e.g., x86, Power, SPARC, ARM, Itanium), and probably even more will be proposed in the near future for various platforms. In this project, to flexibly support various relaxed memory models we plan to make the memory model effect (which governs the relationship among read and write events) parametric to our constraint model. Specifically, we intend to take a formal declarative specification of the memory model as a parameter to our constraint generation algorithm, and then combine it with the concrete program actions from the trace to produce a set of constraints that determine which reads can see values from which writes in the program. We will develop a memory model specification module in our framework that works as a plugin for the maximal causality module and is responsible for formulating the memory model effect in terms of the ordering constraints between events. Building upon our experience with defining formal semantics of low-level languages and memory models for them [18], we will focus on formulating the relaxed memory model semantics for three common processor architectures (x86, Power, and ARM) and for the recent revisions of the C++ and C languages (C++11 and C11). Like in [18], we will probably use our K framework (`http://kframework.org`) for this task, enriched with a constraint formula generator.

3.3 Proposed work: Scalable Verification with Customized Constraint Solving

Another main objective of this project is to make the predictive runtime verfication framework scalable and practical. The performance of our technique largely depends on the complexity of the constraints and the speed of the constraint solver, as the core computation takes place in the constraint solving module. Although existing solvers such as Z3, CVC3, Yices are becoming increasingly powerful with the advancement of decision procedures and proof strategies, they do not achieve optimal performance for solving our constraints because no off-the-shelf decision procedures is targeting our specific problem. Recall from Section 2.2 that our maximal causality constraints consist of the conjunction and disjunction of many simple Boolean expressions over the abstract order variables for events, specifically Boolean expressions over atomic predicates which are just simple ordering comparisons. The most efficient existing decision procedure for this class of constraints is the Integer Difference Logic (IDL) (provided in both Z3 [15] and Yices [16]). However, IDL incurs unnecessary overhead because it uses numerals everywhere for difference arithmetic.

Our goal is to develop a specialized and highly-efficient constraint solving component to maximize the scalability of our framework. Our initial investigation of this problem reveals that the constraint solving task has a large potential to be parallelized. It is natural to be formulated as a graph reachability problem on a large graph, which can be clustered into groups and the computation for each group can be done independently on a separate core or processor. We will continue to investigate this direction and to develop a customized internal constraint solver in our proposed framework. We anticipate this customized solver will not only address the specific problem of predictive runtime verification, but is also general for solving a large class of concurrency related problems in software testing, debuggging, model checking, and formal symbolic verification.

4 Results from Prior NSF Support

The PI's NSF award which is the most related to this proposal is: NSF CCF-0448501 (PI) - CAREER: Runtime Verification and Monitoring, \$400k for the period 2005-2010. (other awards in which the PI was involved were about programming language semantics, or uses of monitoring in other domains, without predictive capabilities). CCF-0448501 investigates generating monitors from formal specifications and weaving them into Java applications. The resulting program monitors its own execution and can recover if specifications are violated, and can predict violations from correct executions. Special effort went into how to do all these very efficiently, with a runtime overhead of < 10%. The following papers resulted from this award: [4, 6–14, 24, 26, 27, 30, 34–36, 39–41, 47]. Two runtime verification software systems resulted from this award, both open source: JavaMOP (`http://fsl.cs.uiuc.edu/MOP`) and jPredictor (`http://fsl.cs.illinois.edu/JPredictor`). Both of these approaches generated significant interest in the runtime verification community, accumulating more than 3,000 citations during the last 5 years (according to Google scholar).

Intellectual Merit: It showed that runtime verification is a viable alternative to traditional formal verification approaches. It also showed that runtime verification narrows the gap between specification and implementation, by allowing the former to play an active role in the execution of programs via monitoring and guiding. Several monitor synthesis algorithms were designed and implemented in JavaMOP, as well as one of the first viable and practical predictive runtime analysis tools for Java, jPredictor.

Broader Impact: Runtime verification (RV) blossomed during the few years, with tracks in major conferences and with its own international conference (the PI chaired its first iteration in 2010). New researchers cite papers in the list above as seminal RV work that opened their domain of research interests. RV modules where introduced in courses at various universities. Students funded by this project use their RV knowledge in companies like Microsoft and Samsung, and even co-founded (jointly with the PI) a startup company called RV (`http://runtimeverification.com`), which currently develops RV products for Toyota and NASA.

5 Curriculum Development Activities

UIUC has a long-term tradition in software engineering, programming languages and formal analysis methods, both in research and in teaching. For example, there are 6 undergraduate courses in these areas taught every year, and three weekly seminars. The topic of this proposal lays at the boundaries between these areas. The PI teaches programming language and formal methods courses at UIUC, and is one of the two co-founders (together with Klaus Havelund of NASA JPL) of the Runtime Verification series of events (initially workshops, now international conferences) who also coined the name "runtime verification". He has helped colleagues at other universities to introduce runtime verification modules into their courses, providing code (JavaMOP [10]), slides and assistance. He has not taught runtime verification topics at UIUC during the last decade, but starting with Fall'14 he is scheduled to co-teach a project-based 150-undergraduate-student class together with Prof. Tao Xie. He plans to include several modules related to runtime verification in general and predictive runtime analysis in particular in that class. The proposed predictive runtime verification project, due to its modular design and infrastructure, is a good source of projects for software engineering students with moderate formal training.

The PI started a book on Runtime Verification more than 5 years ago, which he intends to complete during the time frame of this project. Currently, the book focuses on runtime monitoring only, but the proposed predictive analysis techniques and tools will smoothly complement the existing material, thus making the book give a comprehensive exposition of the state-of-the-art in runtime verification. Also, a user manual will be written for the resulting framework, which we expect to complement the already successful JavaMOP [10] project developed by PI's group. Finally, a filmed tutorial will be developed, following the same style like the K tutorial, which accumulated more than 2,000 accesses per month since its inception in October 2012.

6 Broader Impact

Advancing Discovery and Understanding The proposed work will significantly improve the reliability of production systems, by providing more guarantees before production level is reached by predicting errors from runs of a system that do not actually manifest the error. Results will be published promptly and the tools will be made available on the internet together with appropriate documentation. The current status of the websites for JavaMOP and for K, attest to our thorough support of our tools. Our involvement in international scientific events and in organizing specialized events will facilitate the rapid dissemination of the proposed research to the scientific community, and will encourage the use of the proposed tools both in practice and in the classroom. The PI's prior experience founding, organizing, chairing and steering the Runtime Verification conference over the last decade attests to his ability and willingness to spread awareness of research in runtime verification.

Students supported by this project will be given a solid scientific foundation for designing and implementing support tools for high quality software development, which will greatly improve their skills at developing safe and robust software, and at investigating new analysis technologies.

The PI funds on average two undergraduate students on a regular basis, involving them in research at early stages. One of them is currently a sophomore and is a US citizen; we intend to write a supplemental NSF Research Experiences for Undergraduates (REU) proposal to fund him as part of this project. The PI is currently advising one female graduate student and made an offer to a second. One previous female student co-advised by the PI is now a tenure track faculty member at the University of Toronto.

Benefits to society. Program verification is difficult and expensive. Critical software systems like aeronautics and biomedical instrumentation must have high safety standards, however, because human life depends on them. The cost of program verification for this technologies results in fewer and more expensive products on the market. Our proposed system is much less expensive than program verification, yet is able to provide many of the same guarantees at a fraction of the production costs, resulting in cheaper, yet safe, products.

References

[1] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *OOPSLA*, 2007.

[2] C. Barrett and C. Tinelli. CVC3. In *CAV*, 2007.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA*, 2006.

[4] E. Bodden, F. Chen, and G. Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *Aspect-Oriented Software Development (AOSD'09)*, pages 3–14. ACM, 2009.

[5] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: proportional detection of data races. In *PLDI*, 2010.

[6] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 546–550, 2005.

[7] F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261, 2009.

[8] F. Chen and G. Roșu. Parametric and sliced causality. In *CAV'07*, volume 4590 of *LNCS*, pages 240–253.

[9] F. Chen and G. Roșu. Parametric and termination-sensitive control dependence. In *SAS'06*, volume 4134 of *LNCS*, pages 387–404, 2006.

[10] F. Chen and G. Roșu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, 2007.

[11] F. Chen, M. D'Amorim, and G. Roşu. Checking and correcting behaviors of java programs at runtime with java-mop. In *Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*, pages 3–20, 2005.

[12] F. Chen, T. F. Șerbănuță, and G. Roșu. jpredictor: a predictive runtime analysis tool for java. In *ICSE'08*, pages 221–230, 2008.

[13] F. Chen, P. Meredith, D. Jin, and G. Rosu. Efficient formalism-independent monitoring of parametric properties. In *Automated Software Engineering (ASE'09)*, pages 383–394. IEEE, 2009.

[14] M. d'Amorim and G. Roşu. Efficient monitoring of ω-languages. In *Computer-aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2005.

[15] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[16] B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, 2006.

[17] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. Ifrit: interference-free regions for dynamic data-race detection. In *OOPSLA*, 2012.

[18] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012. doi: 10.1145/2103656.2103719.

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

[20] J. Erickson, M. Musuvathi, sebastian burckhardt, and kirk olynyk. Effective data-race detection for the kernel. In *OSDI*, 2010.

[21] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. *IPDPS*, 2003.

[22] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *FSE*, 2012.

[23] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[24] S. Ghoshal, S. Manimaran, G. Roşu, T. F. Şerbănuţă, and G. Ştefănescu. Monitoring IVHM systems using a monitor-oriented programming framework. In *NASA Langley Formal Methods Workshop (LFM'08)*, 2008.

[25] J. Huang and C. Zhang. PECAN: Persuasive Prediction of Concurrency Access Anomalies. In *ISSTA*, 2011.

[26] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Roşu, and b. . F. y. p. d. p. Darko Marinov, title=Improved Multithreaded Unit Testing.

[27] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu. Garbage collection for monitoring parametric properties. In *Programming Language Design and Implementation (PLDI'11)*, pages 415–424. ACM, 2011. doi: doi:10.1145/1993316.1993547.

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

[29] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 1978.

[30] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *International Conference on Software Engineering (ICSE'11)*, 2011. to appear.

[31] Q. Luo and G. Rosu. EnforceMOP: a runtime property enforcement system for multithreaded programs. In *ISSTA*, 2013.

[32] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *PLDI*, 2009.

[33] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, 2005.

[34] P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. In *Automated Software Engineering (ASE '08)*, pages 148–157. IEEE, 2008.

[35] P. O. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. *J. Automated Software Engineering*, 17(2):149–180, June 2010.

[36] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of monitoring oriented programming. *J. on Software Tools for Technology Transfer*, 2010. to appear.

[37] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[38] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.

[39] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time System Symposium (RTSS'08)*, pages 481–491. IEEE, 2008.

[40] R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha. Handling mixed-criticality in SoC-based real-time embedded systems. In *Embedded Software (Emsoft'09)*, pages 235–244. ACM, 2009.

[41] G. Roşu and S. Bensalem. Allen linear (interval) temporal logic –translation to ltl and monitor synthesis–. In *Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2006.

[42] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an smt-based analysis. In *NFM*, 2011.

[43] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *TOCS*, 1997.

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

[45] K. Sen, G. Rosu, and G. Agha. Runtime safety analysis of multithreaded programs. *FSE*, 2003.

[46] 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 (FMOODS'05)*, pages 211–226, 2005.

[47] K. Sen, G. Rosu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. *J. Software Techniques for Technology Transfer*, 8(3):248–260, 2006.

[48] T. F. Serbanuta, F. Chen, and G. Rosu. Maximal causal models for sequentially consistent systems. In *Runtime Verification (RV'12)*, volume 7687 of *LNCS*, pages 136–150, 2013.

[49] O. Shacham, M. Sagiv, and A. Schuster. Scaling model checking of dataraces using dynamic information. In *PPoPP*, 2005.

[50] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *POPL*, 2012.

[51] F. Sorrentino, A. Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *FSE*, 2010.

[52] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

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

[54] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA*, 2004.

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

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