Predicting Concurrency Failures in the Generalized Execution Traces of x86 Executables

Chao Wang[1] and Malay Ganai[2]

[1] Virginia Tech, Blacksburg, VA 24061, USA
[2] NEC Laboratories America, Princeton, NJ 08540, USA

Abstract. In this tutorial, we first provide a brief overview of the latest development in SMT based symbolic predictive analysis techniques and their applications to runtime verification. We then present a unified runtime analysis platform for detecting concurrency related program failures in the x86 executables of shared-memory multithreaded applications. Our platform supports efficient monitoring and easy customization of a wide range of *execution trace generalization* techniques. Many of these techniques have been successfully incorporated into our in-house verification tools, including BEST (Binary instrumentation based Error-directed Symbolic Testing), which can detect concurrency related errors such as deadlocks and race conditions, generate failure-triggering thread schedules, and provide the visual mapping between runtime events and their program code to help debugging.

1 Introduction

Parallel and concurrent programming is rapidly becoming a mainstream topic in today's corporate world, propelled primarily by the use of multicore processors in all application domains. As the CPU clock speed remains largely constant, developers increasingly need to write concurrent software to harness the computing power of what soon will be the tens, hundreds, and thousands of cores [1]. However, manually analyzing the behavior of a concurrent program is often difficult. Due to the scheduling nondeterminism, multiple runs of the same program may exhibit different behaviors, even for the same program input. Furthermore, the number of possible interleavings in a realistic application is often astronomically large. Even after a failure is detected, deterministically replaying the erroneous behavior remains difficult. Therefore, developers need more powerful analysis and verification tools than what they currently have, in order to deal with concurrency problems such as deadlocks and race conditions.

Although static and dynamic methods for detecting concurrency bugs have made remarkable progress over the years, in practice they can still report too many false alarms or miss too many real bugs. Furthermore, most of the existing bug detection tools target *application-level* software written in languages such as Java or C#. Tools that can directly check the x86 executables of the *system-level* software are lagging behind. Software in the latter category are often more critical to the reliability of the entire system. They may be developed using a wide range of programming languages, including C/C++, and may use external libraries whose source code are not available. This is one reason why we need tools to directly verify x86 executables. Another reason is that x86 executables more accurately reflect the instructions that are executed by the multicore hardware. Almost all microprocessors today are based on the multicore architecture and employ some form of relaxed memory model. Programming languages such as Java and C++ are also in the process of incorporating language-level relaxed memory models [2, 3]. In this case, the behavior of the x86 executable may be drastically different from the source code due to compiler optimizations, especially when the program has concurrency bugs. Therefore, analyzing only the source code is no longer adequate.

We present a runtime analysis and verification platform that can work directly on the x86 executables of Linux applications. We use PIN [4] to instrument both the executables and all the dynamically linked libraries upon which the applications depend. The additional code injected during this instrumentation process are used to monitor and control the synchronization operations such as lock/unlock, wait/notify, thread create/join, as well as the shared memory accesses. We then use a logical constraint based symbolic predictive analysis [5–8] to detect runtime failures by generalizing the recorded execution trace. Our trace generalization model is capable of capturing all the possible interleavings of events of the given trace. We check whether any of the interleaving can fail, by first encoding these interleavings and the error condition as a set of quantifier-free first-order logic formulas, and then deciding the formulas with an off-the-self SMT solver.

Our trace generalization model can be viewed as a kind of lean program slice, capturing a subset of the behaviors of the original program. By focusing on this trace generalization model rather than the whole program, many rigorous but previously expensive techniques, such as symbolic execution [9, 10], become scalable for practical uses.

The remainder of this paper is organized as follows. We give a brief overview of the existing predictive analysis methods in Section 2. We introduce our symbolic predictive analysis in Section 3. The major analysis steps of our BEST tool are presented in Section 4, followed by a discussion of the implementation and evaluation. We review related work in Section 6, and give our conclusions in Section 7.

2 A Brief Overview of Predictive Analysis Methods

Concurrency control related programming errors are due to incorrectly constrained interactions of the concurrent threads or processes. Despite their wide range of symptoms, these bugs can all be classified into two categories. Bugs in the first category are due to *under-constraining*, where the threads have more freedom in interacting with other threads than they should have, leading to *race conditions*, which broadly refer to data races, atomicity violations, and order violations. Bugs in the second category are due to *over-constraining*, where the threads are more restricted than they should be, leading to either deadlocks or performance bugs. Since these bugs are scheduling sensitive, and the number of possible thread interleavings is often astronomically large, they are often rare events during the program execution. Furthermore, in a runtime environment where the scheduling is controlled by the underlying operating system, merely running the same test again and again does not necessarily increase the chance of detecting the bug.

Fig. 1 shows an example of two concurrent threads sharing a pointer p. Due to under-constraining, an atomicity violation may be triggered in some interleaved executions. More specifically, the statements e_2-e_5 in Thread T_1 are meant to be executed

```
Thread T1              Thread T2
{                      e1 : p = &a;
e2 : b = p;
e3 : if(b ≠ 0) {
       ...
e4 :   q = p;
e5 :   * q = 10;
     }                    ...
}                      e6 : p = 0;
```

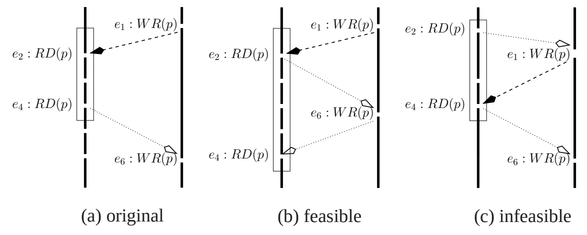


Fig. 1. The given trace $e_1 e_2 \ldots e_6$ in (a) is not buggy, but the alternative interleaving in (b) has an atomicity violation, leading to the null pointer dereference failure at e_5. Assuming that p=0 initially, the interleaving in (c) is bogus since e_4 cannot be executed when $b = 0$.

atomically. Note that such *atomicity* property (between the check $e_2 e_3$ and the use $e_4 e_5$) always holds in sequential programs, but may be broken in a concurrent program unless it is enforced explicitly using synchronizations such as locks. For example, assume that $p = 0$ initially in Fig. 1, then executing e_6 in between e_2, e_4 would lead to a null pointer dereference at e_5. Atomicity violations are different from data races, i.e. a situation where two threads can access the same memory location without synchronization. In Fig. 1, for example, even if we add a lock-unlock pair to protect each access to p (in e_1, e_2, e_4, e_6), the failure at e_5 due to atomicity violation will remain.

Runtime concurrency bug detection and prediction have become an active research topic in recent years [11–18]. Broadly speaking, these existing techniques come in two flavors. When the goal is to detect runtime errors exposed by the given execution, it is called the *monitoring* problem (e.g. [12, 13, 18]). When the goal is to detect errors not only in the given execution, but also in other possible interleavings of the events of that execution, it is called the *prediction* problem. For example, in Fig. 1, the given trace in (a) does not fail. However, from this trace we can infer the two alternative interleavings in (b) and (c). Both interleavings, if feasible, would lead to a runtime failure at e_5. A more careful analysis shows that the trace in (b) is feasible, meaning that it can happen during the actual program execution, whereas the trace in (c) is infeasible, i.e. it is a false alarm.

Depending on how they infer new interleavings from the given trace, predictive analysis methods in the literature can be classified into two groups. Methods in the first group (e.g. [19, 11, 20, 21, 15, 22]) detect must-violations, i.e. the reported violation must be a real violation. Methods in the second group (e.g. [23, 24, 16, 14, 25, 26]) detect may-violations, i.e. the reported violation may be a real violation. Conceptually, methods in the first group start by regarding the given trace ρ as a totally ordered set of events (ordered by the execution sequence in ρ), and then removing the ordering constraints imposed solely by the nondeterministic scheduling. However, since the type of inferred interleavings are limited, these methods often miss many real bugs. In contrast, methods in the second category start by regarding the given trace ρ as an unordered set of events, meaning that any permutation of ρ is initially allowed, and then filtering out the obviously bogus ones using the semantics of the synchronization primitives. For example, if two consecutive events $e_1 e_2$ in one thread and e_3 in another thread are both protected by lock-unlock pair over the same lock, then the permutation $e_1 e_3 e_2$ is forbidden based on the mutual exclusion semantics of locks.

The entire spectrum of predictive analysis methods is illustrated in Fig. 2. Given an execution trace ρ, the left-most point represents the singleton set containing trace ρ itself, whereas the right-most point represents the set of all possible permutations of trace ρ, regardless of whether the permutations are feasible or not. Therefore, the left-most point denotes the coarsest under-approximated predictive model, whereas the right-most point denotes the coarsest over-approximated predictive model. The left-to-middle horizontal line represents the evolution of the under-approximated analysis methods in the first group – they all report must-violations, and they have been able to cover more and more real bugs over the years. This line of research originated from the happens-before causality relationship introduced by Lamport [19]. The right-to-middle horizontal line represents the evolution of the over-approximated analysis methods in the second group – they all report may-violations, and they have been able to steadily reduce the number of false alarms over the years. Some early developments of this line of research include the Eraser-style lockset analysis [23] and the lock acquisition history analysis [27]. Although significant progress has been made over the years, it is still the case that these existing methods may either miss many real bugs or generate many false alarms. For example, if an over-approximated method relies solely on the control flow analysis while ignoring data, it may report Fig. 1 (c) as a violation although the interleaving is actually infeasible. If an under-approximated method strives to avoid false alarms, but in the process significantly restricts the type of inferred traces, it may miss the real violation in Fig. 1 (b).

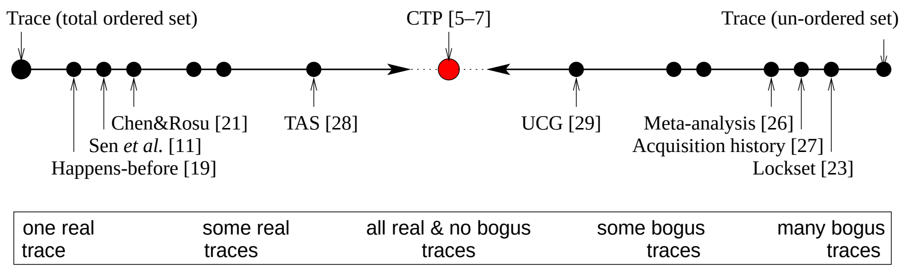


Fig. 2. The landscape of predictive analysis methods

In our recent work [5–7], we introduced a precise trace generalization model together with an efficient logical constraint based symbolic analysis. Our model, called the Concurrent Trace Program (CTP), captures all the interleavings that can possibly be inferred from a given trace, without introducing any bogus interleavings. As illustrated in Fig. 2, CTP represents the theoretically optimal point, where the two long lines of research on predictive methods converge. However, the practical use of CTP as a predictive model depends on how efficient its error detection algorithm is. We believe that the key to its widespread use will be the judicious application of *symbolic analysis* techniques and *interference abstractions*. SMT based symbolic analysis will help combat *interleaving explosion*, the main bottleneck in analysis algorithms based on explicitly enumerating the interleavings. Explicit enumeration is avoided entirely in our SMT-based symbolic analysis. Interference abstraction refers to the over- or under-approximated modeling of the thread interactions with a varying degree of precision. The main idea is that, for the purpose of deciding a property at hand, we often do not need to precisely model all the details of the thread interactions. Our SMT based symbolic analysis provides a flexible and unified framework for soundly performing such over- or under-approximations, without forcing us to worry about the validity of the analysis results.

3 SMT-based Symbolic Predictive Analysis

Recall that the Concurrent Trace Program (CTP) is the optimal predictive model because it can catch all real bugs that can possibly be predicted from a given trace, without introducing any bogus bug. Fig. 3 shows how the CTP is derived from a concrete execution trace. Here the main thread T_0 creates threads T_1 and T_2, waits for their termination, and asserts $(x \neq y)$. This given execution does not violate the assertion. From this trace, however, we can derive the model on the right-hand side, which is a parallel composition of the three bounded straight-line threads. In this model, we remove all the execution ordering constraints (of the given trace) imposed solely by the nondeterministic scheduling. For example, e_{15} can execute after e_{21} although it was executed before e_{21} in the given trace. However, not all interleavings are allowed: interleaving $e_1 e_2 e_{21} e_{26} e_{27} e_{28} e_{11} \ldots e_{15} e_{18} e_3 \ldots e_5$ is not allowed, because the assume condition in e_{26} is invalid, and as a result, we cannot guarantee the feasibility of this interleaving. In other words, this interleaving may be bogus.

Thread T_0

```
      int x = 0, y = 0;
      pthread_t t1, t2;
      main() {
e1        pthread_create(t1, ..., foo);
e2        pthread_create(t2, ..., bar);
e3        pthread_join(t2);
e4        pthread_join(t1);
e5        assert( x != y);
      }
```

Thread T_1

```
      foo() {
e11       int a=y;
e12       if (a==0) {
e13           x=1;
e14           a=x+1;
e15           x=a;
e16       }else
e17           x=0;
e18   }
```

Thread T_2

```
      bar() {
e21       int b=x;
e22       if (b==0) {
e23           y=1;
e24           b=y+1;
e25           y=b;
e26       }else
e27           y=0;
e28   }
```

(Execution Trace)

```
e0: x=0,y=0;
e1: fork(1)
e2: fork(2)
------>
  e11: a=y;
  e12: assume(a=0)
  e13: x=1;
  e14: a=x+1;
  e15: x=a;
  e18:
  ------>
    e21: b=x;
    e26: assume(b≠0)
    e27: y=0;
    e28:
<------
e3: join(2)
e4: join(1)
e5: assert(x ≠ y)
```

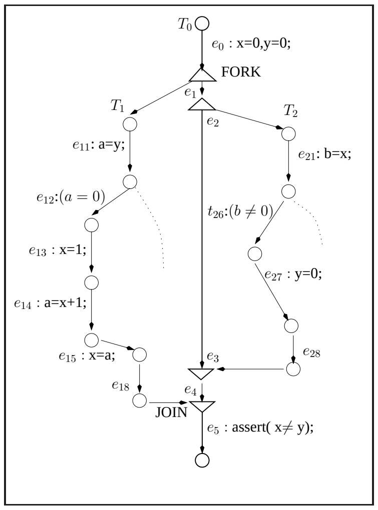


Fig. 3. A multithreaded C program, an execution trace, and the concurrent trace program (CTP).

CTP is ideally suited for detecting concurrency bugs since it is a concurrency control *skeleton*, with most of the complications of typical sequential code removed. For example, pointers have been dereferenced, loops have been unrolled, and recursion has been applied during the concrete execution. Assignment such as $(*p) := 10$ is modeled as $\mathsf{assume}(p{=}\&a);\, a{:=}10$ if p points to variable a in the given execution. As a result, the interleavings in which p does not match the memory address $\&a$ are excluded from the model. In other words, pointer p has been replaced by one of its constant value $\&a$. The variables and expressions whose valuations are insensitive to thread scheduling can be replaced by their concrete values in the given trace. The only source of nondeterminism in a CTP comes from the thread interleaving.

3.1 SMT-based Symbolic Encoding

We check for property violation by formulating this verification problem as a constraint solving problem. That is, we build a quantifier-free first-order logic formula Φ such that Φ is satisfiable if and only if there is an erroneous interleaving in the CTP. Conceptually, $\Phi := \Phi_{TM} \wedge \Phi_{SC} \wedge \Phi_{PRP}$, where Φ_{TM} is the thread model encoding the individual behaviors of all threads, Φ_{SC} is the sequential consistency model encoding all the valid thread interactions, and Φ_{PRP} is the property constraint encoding the failure condition. Central to the analysis is Φ_{SC}, which specifies, in a valid interleaving, which shared memory read should be mapped to which shared memory write and under what condition. For example, each shared memory read r_x must match a preceding shared memory write w_x for the same memory location x; and if r_x matches w_x, then any other write w'_x to the same location must happen either before w_x or after r_x. Synchronization operations such as lock-unlock and wait-notify are modeled similarly. The logic formula Φ is then decided by an off-the-shelf SMT solver.

Compared to existing methods, our constraint-based approach provides a unified analysis framework with the following advantages. First, it is flexible in checking a diverse set of concurrency related properties; there is no longer a need to develop separate algorithms for detecting deadlocks, data races, atomicity violations, etc. All these properties can be modeled in our framework as a set of logical constraints. Second, it is efficient since the program behaviors are captured *implicitly* as a set of mathematical relations among all synchronization operations and shared-memory accesses, therefore avoiding the interleaving explosion. Third, our analysis is more precise and covers more interleavings. It also allows easy exploitation of the various trade-offs between the analysis precision and the computation overhead, simply by adding or removing some logic constraints. This is crucially importantly because, as we have mentioned earlier, not all the inference constraints (in Φ_{SC}) may be needed for deciding the property at hand. Forth, our symbolic encoding is compositional in that the behaviors of the individual threads are modeled as one set of logical constraints (in Φ_{TM}), while the thread interference is modeled as another set of logical constraints (in Φ_{SC}). The parallel composition is accomplished by conjoining these two sets of constraints together.

3.2 Interference Abstractions

While the CTP model and the associated symbolic predictive analysis provide a solid theoretical foundation, their practical use will hinge upon the judicious application of proper interference abstractions. Interference abstraction refers to the over- or under-approximated modeling of the thread interactions with a varying degree of precision. In our symbolic analysis framework, interference abstractions are manifested as the over-

or under-approximations of formula Φ_{SC}. Since modeling the thread interactions is the most expensive part of the concurrent program analysis, without abstraction, symbolic analysis will not be able to scale to large applications. Our main hypothesis is that, since concurrency bugs typically involve a small number of unexpected thread interferences, they can often be captured by succinct interference abstractions.

In a previous work [29], we proposed an over-approximated interference abstraction, called the *Universal Causality Graph (UCG)*, where the shared-memory accesses are abstracted away while the control flow and the synchronization primitives are retained. We represent the happens-before causality relationship among trace events as a graph, where the nodes are the events and the edges are must-happen-before relations between the events, as imposed by the thread-local program order, the synchronization primitives, and the property. Checking whether a property holds can be reduced to the problem of checking whether these causality edges can form a cycle. The existence of a cycle means that none of the interleavings of the CTP can satisfies the property. However, due to over-approximations, this analysis is conservative in that it guarantees to catch all violations that can possibly be predicted from a given trace, but may report some false alarms. Our UCG based analysis is provably more accurate than the existing methods in the same category, e.g. the widely used lockset based methods [23, 24, 16, 14, 25, 26]. The reason is that lockset analysis typically models locks precisely, but cannot robustly handle synchronization primitives other than locks, such as wait-notify and fork-join. In contrast, our UCG based method precisely model the semantics of all common synchronization primitives, as well as the synergy between the different types of primitives.

In another work [28], we proposed an under-approximated interference abstraction called the *Trace Atomicity Segmentation (TAS)*, which can soundly restrict the search space that needs to be considered to detect the most general form of atomicity violations. More specifically, the TAS is a trace segment consisting of all the events in the surrounding areas of an atomic block, such that these events are sufficient for checking whether this atomicity property can be violated. Different from most existing work, our method can detect violations that involve an arbitrary number of variables and threads, rather than the simplest atomicity violations involving a single variable and three memory accesses. As illustrated in Fig. 2, TAS is regarded as an under-approximation. The case for using TAS in practice is when the runtime analysis does not have access to the program code, or cannot afford to monitor every instruction, but is still required to guarantee no false alarms. Our preliminary experiments in [28] show that the TAS is typically small even in an otherwise long execution trace.

We also proposed an algorithm to automatically find the interference abstraction that is optimal to the property at hand. Unlike the ones with a prescribed precision, a property specific interference abstraction can be more efficient since it only needs a minimal set of interference constraints. The rationale behind is that sometimes we can prove a property using an over-approximated abstraction, e.g. the control-state reachability analysis [26, 29]. Sometimes we can detect real bugs with an under-approximated abstraction, e.g. by artificially bounding the number of context switches, since the bugs may be scheduling-insensitive, and therefore may show up even in serial executions or when threads interleave only sporadically [30, 31]. However, it is generally difficult to decide *a priori* which abstraction is more appropriate. To solve this problem, we proposed an iterative refinement algorithm [32]. We will start with a coarse initial abstraction which is either over- or under-approximated, based on whether the property likely holds or not. Depending on the initial abstraction, this refinement process may be either under-approximation widening or over-approximation refinement. It is interesting to point out that, *optimal* interference abstraction, defined as the most succinct abstraction that is sufficient to decide the property, may not be a purely over-approximated model or a purely under-approximated model, but a hybrid model as represented by the dots in the middle of Fig. 4. In this figure, we have bent over the right-to-middle horizontal line in Fig. 2 to make it the bottom-to-top vertical line. Most of the points in this two-dimensional plane correspond to the hybrid models. As we have shown in [32], with a careful analysis, such hybrid models can still be used to accurately decide the property at hand, despite the fact that they are considered as neither sound nor complete in the traditional sense. Fig. 5 shows that small interference abstractions are often sufficient for checking properties in realistic applications, and that their use can drastically improve the scalability of our symbolic analysis.

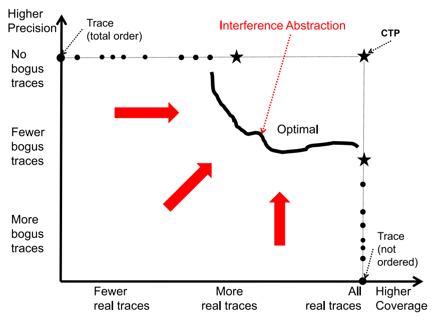


Fig. 4. Finding the optimal interference abstraction: identifying the smallest set of interference constraints that are sufficient for deciding the property.

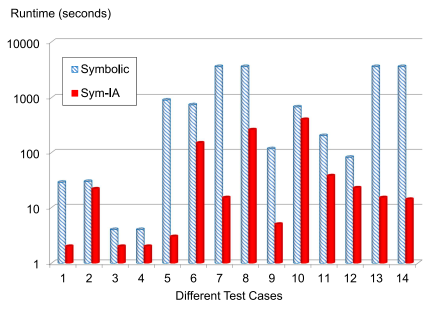


Fig. 5. Experimental results from [32]: using interference abstraction can lead to faster property checking than using the full-blown interference constraints.

4 The BEST Platform

Our *Binary instrumentation-based Error-directed Symbolic Testing (BEST)* tool implements some of the symbolic predictive analysis techniques introduced in the previous sections, and is capable of detecting concurrency errors by directly monitoring an unmodified x86 executable at runtime. In the remainder of this paper, we shall use atomicity violations as an example to illustrate the features of our framework. As shown in Fig. 6, the predictive analysis in BEST consists of the following stages:

- Stage I, recording the execution trace and building the predictive model;
- Stage II, simplifying the model using sound program transformations;
- Stage III, inferring and then statically pruning the atomicity properties;
- Stage IV, predicting the violations of the atomicity properties;

- Stage V, replaying the erroneous interleaving, to see if it can cause runtime failures.
- Go back to Stage I.

Before using this tool, the developer needs to provide an execution environment for the program under test, i.e. a test harness. Details of the stages are illustrated as follows.

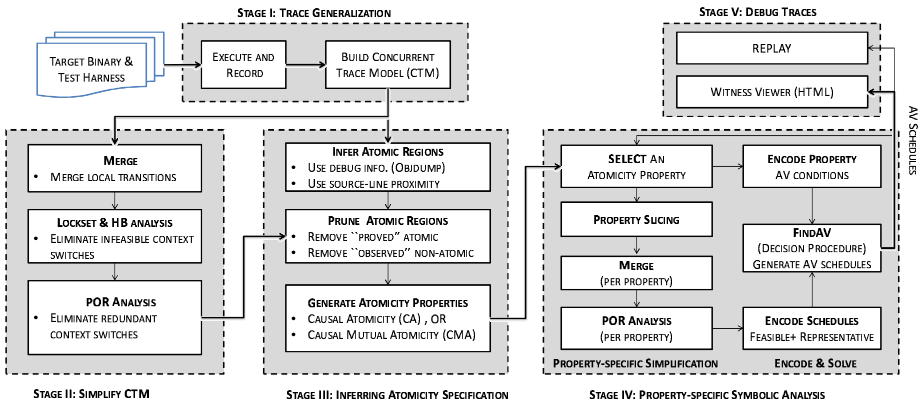

Stage I: Trace Generalization
Target Binary & Test Harness
Execute and Record
Build Concurrent Trace Model (CTM)
Stage V: Debug Traces
Replay
Witness Viewer (HTML)
AV Schedules
Merge
• Merge local transitions
Lockset & HB analysis
• Eliminate infeasible context switches
POR Analysis
• Eliminate redundant context switches
Stage II: Simplify CTM
Infer Atomic Regions
• Use debug info. (Objdump)
• Use source-line proximity
Prune Atomic Regions
• Remove ``proved'' atomic
• Remove ``observed'' non-atomic
Generate Atomicity Properties
• Causal Atomicity (CA) , or
• Causal Mutual Atomicity (CMA)
Stage III: Inferring Atomicity Specification
SELECT An Atomicity Property
Property Slicing
Merge (per property)
POR Analysis (per property)
Encode Property AV conditions
FindAV (Decision Procedure) Generate AV schedules
Encode Schedules Feasible+ Representative
Property-specific Simplification
Encode & Solve
Stage IV: Property-specific Symbolic Analysis

Fig. 6. BEST architecture

4.1 The Staged Analysis

Stage I. While testing the concurrent application, we use PIN to instrument the executable at run time to record the sequence of global events generated by the execution. The global events include both synchronization operations such as lock-unlock and the shared memory reads and writes. From this sequence of events, we derive a concurrent trace model (CTM), which may be an over-approximation of the CTP. The model can be viewed as a *generator* of traces, including both the given trace ρ and all the other interleavings that can be obtained by relaxing the ordering constraints (in ρ) imposed by the non-deterministic scheduling. Even if the given execution trace ρ does not fail, a runtime failure may still occur in some of the alternative interleavings.
Stage II. Given the initial model, we perform the following simplifications. First, we identify the operations over only thread-local variables, where the thread-local variables are identified by checking whether their memory locations are accessed by more than one concurrent threads. Then, we merge consecutive thread-local operations into a single operation. Next, we perform constant value propagation to simplify all the expressions that are scheduling-insensitive. These simplifications can lead to orders-of-magnitude reduction in the model size, measured in terms of the number of trace events. Finally, we use sound static analysis techniques such as lockset analysis and simple happen-before (HB) analysis to quickly identify the ordering constraints imposed by synchronizations (which must be satisfied by all valid interleavings) and then eliminate the obviously infeasible interleavings.

Stage III. On the simplified model, we infer the likely atomic regions based on the structure of the program code. Note that these atomic regions may involve multiple shared variable accesses. We also assume that the given trace is good (unless it fails) and therefore remove any region that is not atomic in the given execution. The remaining regions are treated as atomic. We use the notion of *causal atomicity* as in [33] as well as the notion of *causal mutual atomicity* (CMA) as in [34]. In the latter case, we check the violation of two pair-wise atomic regions from different threads with at least two conflicting transitions.
Stage IV. For each atomicity property, we perform a property specific program slicing, followed by another pass of simplifications and merging of the consecutive thread-local events. We check for violations of the atomicity properties by formulating the problem as a constraint solving problem. That is, we build a quantifier-free first-order logic formula Φ such that Φ is satisfiable if and only if there is an erroneous interleaving. The logic formula Φ is then decided by an off-the-shelf SMT solver.
Stage V. Once our SMT based analysis detects a violation, it will generate an erroneous thread schedule. To replay it, we use PIN to instrument the executables at runtime, and apply the externally provided schedule. After Stage V, we go back to Stage I again. The entire procedure stops either when a runtime failure (e.g. crash) is found, or when the time limit is reached.

Our BEST tool can provide the visualization of the failure-triggering execution. If the executable contains the compiler generated debugging information, BEST can also provide a mapping from the trace events to the corresponding program statements. On the Linux platform, for example, we use a gnu utility called `objdump` to obtain the mapping between processor instructions and the corresponding source file and line information.

4.2 Inferring Atomicity Properties

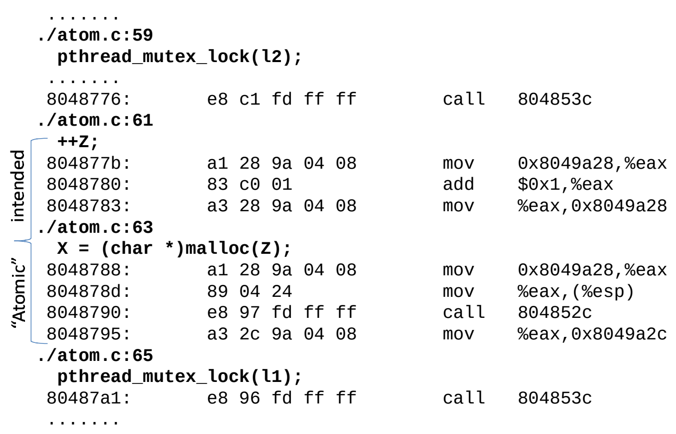
Programmers often make some implicit assumptions regarding the concurrency control of the program, e.g. certain blocks are intended to be mutually exclusive, certain blocks are intended to be atomic, and certain instructions are intended to be executed in a specific order. However, sometimes these implicit assumptions are not enforced using synchronization primitives such as locks and wait-notify. Concurrency related program failures are often the result of these implicit assumptions being broken, e.g. data races, atomicity violations, and order violations. There are existing methods (e.g. [12]) for statically mining execution order invariants form the program source code. There are also dynamic methods (e.g. [13]) for inferring invariants at runtime. For example, if no program failure occurs during testing, then the already tested executions often can be assumed to satisfy the programmer's intent.

Our BEST tool heuristically infers such likely atomicity properties from the x86 executables. Our approach is an application of the existing methods in [12, 13] together with the following extensions. Let a global access denote either a synchronization operation or a shared memory accesses. When inferring the likely atomic regions, we require each region to satisfy the following conditions:

- the region must contain at least one shared memory read/write;
- the first and/or last global access must be a shared memory read/write;
- the global accesses must be within a procedure boundary;

– the global accesses must be close to each other in the program code;

In additional, the region should not be divided by blocking synchronization operations such as thread creation/join or the wait/notify, which will make the region non-atomic.


intended
“Atomic”

```
 .......
./atom.c:59
  pthread_mutex_lock(l2);
 .......
 8048776:       e8 c1 fd ff ff        call   804853c
./atom.c:61
  ++Z;
 804877b:       a1 28 9a 04 08        mov    0x8049a28,%eax
 8048780:       83 c0 01              add    $0x1,%eax
 8048783:       a3 28 9a 04 08        mov    %eax,0x8049a28
./atom.c:63
  X = (char *)malloc(Z);
 8048788:       a1 28 9a 04 08        mov    0x8049a28,%eax
 804878d:       89 04 24              mov    %eax,(%esp)
 8048790:       e8 97 fd ff ff        call   804852c
 8048795:       a3 2c 9a 04 08        mov    %eax,0x8049a2c
./atom.c:65
  pthread_mutex_lock(l1);
 80487a1:       e8 96 fd ff ff        call   804853c
 .......
```

Fig. 7. Inferring atomicity with `objdump` using code structure

Fig. 7 shows an example of inferring the likely atomic regions from the program code, by following the above guidelines. This figure contains the output of *objectdump* for a small C program called *atom.c* at Lines 59, 61, 63, and 65. The entire execution trace, together with its CTM and interleaving lattice, can be found in [35]. The transition corresponding to `pthread_mutex_lock(l2)` is assigned a tag $\langle atom.c, 59\rangle$. Similarly, the transitions corresponding to `++Z` is assigned a tag $\langle atom.c, 61\rangle$. Using the rules for inferring atomic regions, we mark the transitions corresponding to statements `++Z` and `X=(char*)malloc(Z)` as the likely atomic region. In other words, if we can find an interleaved execution which breaks this atomicity assumption, the execution will be regarded as risky – it is more likely to lead to a program failure. In Stage V of our BEST tool, we will replay such interleavings in order to maximize the exposure of the real failures.

5 Implementation and Evaluation

Our tool has been implemented for x86 executables on the Linux platform. We use PIN [4] for dynamic code instrumentation and the YICES [36] solver for symbolic predictive analysis. Our BEST tool can directly check for concurrency failures in executables that use the POSIX threads. Whenever the program source code are available, for example, in C/C++/Java, we use *gcc/g++/gcj* to compile the source code into x86 executables before checking them. With the help of dynamic instrumentation form PIN, we can model the instructions that come from both the application and the dynamically linked libraries. Specifically, we are able to record all the POSIX thread synchronizations such as wait/notify, lock/unlock, and fork/join, as well as the shared memory accesses.

For efficiency reasons, BEST may choose to turn off the recording of the thread-local operations such as stack reads/writes. This option in principle may lead to a further over-approximation of the trace generalization model, meaning that some of the violations reported by our analysis may be spurious. As a result, replay in Stage V may fail (our bailout strategy is to start a free run as soon as the replay fails). However, such cases turn out to be rare in our experiments.

We have experimented with some public domain multi-threaded applications from the sourceforge and freshmeat websites. The size of these benchmarks are in the range of 1K-33K lines of C/C++ or Java code. They include *aget* (1.2K LOC, `C`), *fastspy* (1.5K LOC, `C`), *finalsolution* (2K LOC, `C++`), *prozilla* (2.7K LOC, `C++`), *axel* (3.1K LOC, `C`), *bzip2smp* (6.4K LOC, `C`), *alsaplayer* (33K LOC, `C++`), and *tsp* (713, `Java`). The length of the execution trace ranges from a few hundreds to 34K events, with 4 to 67 threads. Most of the inferred atomic regions involve more than one variable accesses. Due to the use of interference abstractions and the various model simplification and search space reduction techniques, the CPU time per check by our analysis is a few seconds on average.

Our BEST tool found several previously known/unknown atomicity violations. The bug list can be found in `http://www.nec-labs.com/∼malay/notes.html`.

6 Related work

We have reviewed the existing methods for runtime monitoring and prediction of concurrency failures in Section 2. It should be clear that for such analysis to detect a failure, a *failure-inducing* execution trace should be provided as input, which contains all the events that are needed to form a *failure-triggering* interleaving. While we have assumed that this failure-inducing execution trace is available, generating such trace can be a difficult task in practice, since it requires both the *right* thread schedule and the *right* program input.

When the thread scheduling is controlled by the operating system, it is difficult to generate a failure-inducing thread schedule – repeating the same test does not necessarily increase the coverage. Standard techniques such as load/stress tests and randomization [37] are not effective, since they are highly dependent on the runtime environment, and even if a failure-inducing schedule is found, replaying the schedule remains difficult. CHESS-like tools [38, 31, 39] based on stateless model checking [40] are more promising, but too expensive due to interleaving explosion, even with partial order reduction [41] and context bounding [42, 43]. A more practical approach is to systematically, but also selectively, test a subset of thread schedules while still cover the common bug patterns. Similar approaches have been used in CalFuzzer [44], PENELOPE [45], and our recent work in [46].

Generating the failure-inducing execution trace also requires the right data input. In practice, test inputs are often hand crafted, e.g. as part of the testing harness. Although DART-like automated test generation techniques [47–54] have made remarkable progress for sequential programs, extending them to concurrent programs has been difficult. For example, ESD [55] extended the test generation algorithm in KLEE [52] to multithreaded programs; Sen and Agha [20] also outlined a concolic testing algorithm for multithreaded Java. However, these existing methods were severely limited by interleaving explosion – it is difficult to systematically achieve a decent code and interleaving coverage within a reasonable period of time. In ESD, for example, heuristics are used to artificially reduce the number of interleavings; however, the problem is that the reduction is arbitrary and often does not match the common concurrency bug patterns. This leads to missed bugs, and also makes it difficult to identify which part of the search space is covered and which part is not. Therefore, we consider scalable and efficient test input generation for concurrent programs as an interesting problem for a future work.

7 Conclusions

In this paper, we have provided a brief overview of the latest development in SMT-based symbolic predictive analysis. We have also presented our BEST tool for detecting runtime failures in unmodified x86 executables on the Linux platform using POSIX threads. BEST uses a staged analysis with various simplifications and model reduction techniques to improve the scalability of the symbolic analysis. It infers likely atomicity properties and then checks them using the symbolic analysis. Thread schedules that violate some of these likely atomicity properties are used to re-direct the testing toward the search subspaces with a higher risk. BEST also provides the visualization of trace events by mapping them to the program statements to help debugging. We believe that these SMT-based symbolic predictive analysis techniques hold great promise in significantly improving concurrent program verification.

References

1. Ross, P.E.: Top 11 technologies of the decade. IEEE Spectrum **48**(1) (2011) 27–63
2. Manson, J., Pugh, W., Adve, S.V.: The java memory model. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. (2005) 378–391
3. Boehm, H.J., Adve, S.V.: Foundations of the c++ concurrency memory model. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. (2008) 68–78
4. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: PIN: Building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, ACM (2005) 190–200
5. Wang, C., Kundu, S., Ganai, M., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: International Symposium on Formal Methods. (2009) 256–272
6. Wang, C., Chaudhuri, S., Gupta, A., Yang, Y.: Symbolic pruning of concurrent program executions. In: ACM SIGSOFT Symposium on Foundations of Software Engineering. (2009) 23–32
7. Wang, C., Limaye, R., Ganai, M., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems. (2010)
8. Kundu, S., Ganai, M.K., Wang, C.: CONTESSA: Concurrency testing augmented with symbolic analysis. In: International Conference on Computer Aided Verification. (2010) 127–131
9. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7) (1976) 385–394
10. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Software Eng. **2**(3) (1976) 215–222

11. Sen, K., Rosu, G., Agha, G.: Runtime safety analysis of multithreaded programs. In: ACM SIGSOFT Symposium on Foundations of Software Engineering. (2003) 337–346
12. Xu, M., Bodík, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. (2005) 1–14
13. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. In: Architectural Support for Programming Languages and Operating Systems. (2006) 37–48
14. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Software Eng. **32**(2) (2006) 93–110
15. Chen, F., Serbanuta, T., Rosu, G.: jPredictor: a predictive runtime analysis tool for java. In: International Conference on Software Engineering. (2008) 221–230
16. Flanagan, C., Freund, S.N.: Atomizer: A dynamic atomicity checker for multithreaded programs. In: Parallel and Distributed Processing Symposium. (2004)
17. Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. (2008) 293–303
18. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: International Conference on Computer Aided Verification. (2008) 52–65
19. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565
20. Sen, K., Rosu, G., Agha, G.: Detecting errors in multithreaded programs by generalized predictive analysis of executions. In: Formal Methods for Open Object-Based Distributed Systems. (2005) 211–226
21. Chen, F., Rosu, G.: Parametric and sliced causality. In: International Conference on Computer Aided Verification, Springer (2007) 240–253 LNCS 4590.
22. Sadowski, C., Freund, S.N., Flanagan, C.: Singletrack: A dynamic determinism checker for multithreaded programs. In: European Symposium on Programming. (2009) 394–409
23. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4) (1997) 391–411
24. von Praun, C., Gross, T.R.: Object race detection. In: ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications. (2001) 70–82
25. Farzan, A., Madhusudan, P.: The complexity of predicting atomicity violations. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems. (2009) 155–169
26. Farzan, A., Madhusudan, P.: Meta-analysis for atomicity violations under nested locking. In: International Conference on Computer Aided Verification. (2009) 248–262
27. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: International Conference on Computer Aided Verification. (2005) 505–518 LNCS 3576.
28. Sinha, A., Malik, S., Wang, C., Gupta, A.: Predictive analysis for detecting serializability violations through trace segmentation. In: International Conference on Formal Methods and Models for Codesign. (2011)
29. Kahlon, V., Wang, C.: Universal Causality Graphs: A precise happens-before model for detecting bugs in concurrent programs. In: International Conference on Computer Aided Verification. (2010) 434–449
30. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems, Springer (2005) 93–107
31. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI. (2008) 267–280
32. Sinha, N., Wang, C.: On interference abstractions. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. (2011) 423–434

33. Farzan, A., Madhusudan, P.: Causal atomicity. In: International Conference on Computer Aided Verification. (2006) 315–328
34. Ganai, M., Kundu, S., Gupta, R.: Partial order reduction for scalable testing of SystemC TLM designs. In: Design Automation Conference. (2008)
35. Ganai, M.K., Arora, N., Wang, C., Gupta, A., Balakrishnan, G.: BEST: A symbolic testing tool for predicting multi-threaded program failures. In: IEEE/ACM International Conference On Automated Software Engineering. (2011)
36. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: International Conference on Computer Aided Verification, Springer (2006) 81–94 LNCS 4144.
37. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: Parallel and Distributed Processing Symposium. (2003) 286
38. Godefroid, P.: Software model checking: The VeriSoft approach. Formal Methods in System Design **26**(2) (2005) 77–101
39. Yang, Y., Chen, X., Gopalakrishnan, G.: Inspect: A runtime model checker for multithreaded C programs. Technical Report UUCS-08-004, University of Utah (2008)
40. Godefroid, P.: VeriSoft: A tool for the automatic analysis of concurrent reactive software. In: International Conference on Computer Aided Verification. (1997) 476–479
41. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. (2005) 110–121
42. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. (2004) 14–24
43. Musuvathi, M., Qadeer, S.: Partial-order reduction for context-bounded state exploration. Technical Report MSR-TR-2007-12, Microsoft Research (December 2007)
44. Joshi, P., Naik, M., Park, C.S., Sen, K.: CalFuzzer: An extensible active testing framework for concurrent programs. In: International Conference on Computer Aided Verification. (2009) 675–681
45. Sorrentino, F., Farzan, A., Madhusudan, P.: PENELOPE: weaving threads to expose atomicity violations. In: ACM SIGSOFT Symposium on Foundations of Software Engineering. (2010) 37–46
46. Wang, C., Said, M., Gupta, A.: Coverage guided systematic concurrency testing. In: International Conference on Software Engineering. (2011) 221–230
47. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems. (2003) 553–568
48. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Programming Language Design and Implementation. (June 2005) 213–223
49. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Network and Distributed System Security Symposium. (2008)
50. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ACM SIGSOFT Symposium on Foundations of Software Engineering. (2005) 263–272
51. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: ASE. (2008) 443–446
52. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. (2008) 209–224
53. Lewandowski, G., Bouvier, D.J., Chen, T.Y., McCartney, R., Sanders, K., Simon, B., VanDeGrift, T.: Commonsense understanding of concurrency: computing students and concert tickets. Commun. ACM **53** (July 2010) 60–70
54. Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: International Conference on Software Engineering. (2011) 1066–1071
55. Zamfir, C., Candea, G.: Execution synthesis: a technique for automated software debugging. In: EuroSys. (2010) 321–334