Dynamic Invariant Generation for Multithreaded Programs

Arijit Chattopadhyay

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

April 30, 2014

Blacksburg, Virginia

Keywords: Concurrency, Likely Invariant, Dynamic Invariant Generation, Partial Order Reduction, Error Diagnosis, Atomic Region

Dynamic Invariant Generation for Multithreaded Programs

Arijit Chattopadhyay

(ABSTRACT)

We propose a fully automated and dynamic method for generating likely invariants from multithreaded programs and then leveraging these invariants to infer atomic regions and diagnose concurrency errors in the software code. Although existing methods for dynamic invariant generation perform reasonably well on sequential programs, for multithreaded programs, their effectiveness often reduces dramatically in terms of both the number of invariants that they can generate and the likelihood of them being true invariants. We solve this problem by developing a new dynamic invariant generator, which consists of a new *LLVM* based code instrumentation tool, an *Inspect* based thread interleaving explorer, and a customized inference engine inside *Daikon*. We have evaluated the resulting system on public domain multithreaded C/C++ benchmarks. Our experiments show that the new method is effective in generating high-quality invariants. Furthermore, the *state and transition invariants* generated by our new method have been proved useful in applications such as error diagnosis and inferring atomic regions in the concurrent software code.

Dedication

My Mother Chandana Chattopadhyay

and

My Father Sudhakar Chattopadhyay .

Acknowledgments

I would like to extend my deep gratitude to my advisor, Professor Chao Wang, for giving me the opportunity to work in Reliable Software and Systems (RSS) Lab at Virginia Tech. His endless support, motivation, guidance, and unflagging faith on me made this work possible. I am honored to have this opportunity to learn various aspects of research in software verification. At the same time, I am extremely proud to have Dr. Michael Hsiao and Dr. Sandeep Shukla on my MS thesis committee. Their insight, advice, and help contributed strongly to this work and kept me motivated and focused. My thesis work is supported in part by the NSF grant CCF-1149454 and the ONR grant N00014-13-1-0527. I would like to thank both the institutions for providing necessary funding. I would also like to acknowledge Dr. Changhee Jung for his guidance on using the *LLVM* compiler infrastructure.

I would like to express my sincere gratitude to my sister Ishita and brother-in-law Subhendu for inspiring me in doing scientific research. I would also like to thank Rajni, for being extremely supportive and helpful, despite of my surliness and frustration.

I would take this opportunity to thank Professor Suvrojit Das from NIT-Durgapur, who first ignites the passion for systems software in me. I would also like to thank Markus Kusano for providing test programs and technical details about the *Inspect* tool. I acknowledge the help and inspiration that I received from my friends in the RSS and PROACTIVE research groups, which engaged me in doing hard work with light heart. I would also like to thank all my friends in Blacksburg, who made my stay here even more enjoyable.

My implementation of the proposed methods uses various open source software tools and development frameworks. I have extensively used the *LLVM* compiler infrastructure, the *GCC* compiler, and *Java*, *Python* and *C/C++* languages. My research also builds upon research tools such as *Daikon* and *Inspect*. I have used the *VIM* text editor for writing the code, LaTeX text formatter for writing the thesis and other documents. I would like to thank contributors to these awesome software tools.

Contents

List of Figures

List of Tables

List of Abbreviations

LLVM	Low Level Vritual Machine
DPOR	Dynamic Partial Order Reduction
PCB	Pre-emptive Context Bounding
HaPSet	History Aware Predecessor Set

Chapter 1

Introduction

Multithreaded programs remain notoriously difficult to design and analyze due to the often subtle thread interactions and astronomically large number of possible thread interleavings. Existing methods for dynamically generating likely invariants from sequential programs have proved to be valuable in many settings including program understanding, software maintenance, verification, error diagnosis, and automated repair. However, effective methods for dynamically generating likely invariants from multithreaded C/C++ programs are still lacking.

Existing dynamic invariant generation methods work well on sequential programs, for concurrent programs, our experience shows that their effectiveness reduces dramatically in terms of both the number of invariants that they can generate and the likelihood of them being true invariants. The main reason is that dynamic invariant generation depends heavily on the set of execution traces fed to the inference engine. In general, if the execution traces are not diversified enough, the resulting likely invariants will be less representative of the actual program behavior. However, generating a set of sufficiently diverse execution traces is difficult in the context of multithreaded programs, since the program behavior depends not only on the program's data input but also on the thread schedules. In a typical execution environment, the thread scheduling is determined by the operating system and the actual implementation of the thread library. Therefore, naively executing the program many times would not necessarily increase the diversity of the thread interleavings.

For example, although `Daikon` is a highly successful tool for dynamically generating likely invariants for sequential programs written in languages such as Java, C, C++ and Perl, we will show in Section for multithreaded programs, `Daikon`'s invariant inference engine often gets confused and therefore produces wrong invariants.

Another limitation of existing dynamic invariant generation methods is that they tend to report too many invariants. Even if many of them are actually true invariants, it is unlikely that the developers will have time to sift through all of them and make sense of them. Furthermore, in the context of analyzing the behavior of multithreaded programs, the basic assumption often is that (1) the sequential part of the computation is implemented correctly, to an extent that the program behaves as expected most of the time, except for some rare thread interleavings; and (2) the concurrency control part of the program is potentially problematic, where rare but subtle thread interactions may cause the program to fail. In this case, we argue that the focus should be on analyzing a special type of invariants called the *transition invariants*. These are invariants over shared variables only and are capable of capturing programmer's intent regarding concurrency control, e.g. certain code block should be kept atomic, and certain operations should be made mutually exclusive.

1.1 Background

There is a large body of work on using abstract interpretation based static analysis [8, 9] for invariant generation. The main advantage of these techniques is that the reported invariants are true for every reachable state of the program. Typically, invariants generated by these techniques are predicates expressed in some linear abstract domains, such as difference logic, UTVPI, octagonal, polyhedral, etc. There is another line of research on static invariant generation, which relies on the constraint based approach [7, 31, 30]. The main advantage of these methods is that they are able to generate complex invariants, including polynomial invariants and non-linear invariants. Recent development along this line includes the work by Furia *et al.* [15] on generating loop invariants from post-conditions, and invariants related to integer arrays [4, 5, 19]. However, due to the inherent limitations of static program analysis, invariant generation based on purely static techniques tend to lack either in precise or in scalability. Our new method, in contrast, relies solely on dynamic analysis techniques.

There is also a large body of work on using dynamic analysis techniques for invariant generation. Tools developed using these techniques, such as Daikon [11, 12], have been highly successful in practice. The main advantage of dynamic invariant generation techniques is their versatility. They have been applied to realistic applications for which static techniques are difficult to handle and have front-end tools for a wide range of programming languages. `Daikon`, for example, has a front-end for C/C++ called Kvasir and a front-end for Java called Chicory. Other dynamic invariant generation tools along this line include DIDUCE [18], DySy [10], Agitator [3], and Iodine [17].

However, existing dynamic generation tools do not work well on multithreaded programs due to the nondeterminism in thread scheduling. We have solved this problem by developing a method based on a new code instrumentation tool, a systematic interleaving explorer, a trace classifier, and a customized invariant inference engine.

There are also hybrid techniques for invariant generation, which leverages both static analysis and dynamic analysis to improve performance. For example, Nguyen *et al.* [26] proposed a method for generating invariants expressed as polynomials and linear relations over a limited number array variables. Such invariants have been difficult to generate by existing methods. There are also hybrid techniques based on random testing [22] and guess-and-check [32], which first generate a set of candidate invariants from concrete execution data and then verify them using SMT solvers. These techniques are orthogonal to our work.

There are many existing methods for detecting and diagnosing atomicity violations in multi-threaded programs. Some of these methods are based on inferring likely atomic regions statically [38], whereas others are based on dynamic [23, 37, 6, 33] and hybrid [35] techniques. The CTrigger [27] tool, for instance, first generates ordering constraints over global events from good runs through profiling, and then leverages the constraints during test runs. However, existing methods focus exclusively on the event ordering constraints, e.g., whether a set of shared variable accesses from different threads should or should not appear in a certain order. In contrast, the transition invariants generated and used by our new method are thread-modular, in that they refer only to the current and past values of shared variables accessed from the same thread. In this sense, our method is orthogonal to the existing techniques.

1.2 Contribution

In this thesis, we propose a fully automated and dynamic method for generating high quality invariants from multithreaded C/C++ programs to address the aforementioned limitations in existing methods. We also show that the state and transition invariants produced by our method is useful in applications such as diagnosing concurrency errors and inferring likely atomic regions in the software code. Our method builds on a new `LLVM` based code instrumentation tool, an `Inspect` based thread interleaving explorer, and a customized invariant inference engine in `Daikon`.

The overall flow of our method is illustrated in Figure 1.1, which takes multithreaded C/C++ code as input and generates a set of likely invariants as output. First, the program is processed by the instrumentation tool, which adds monitoring and control capabilities to the program for dynamic analysis. Then, we run the instrumented executable under `Inspect`, which systematically explores the possible thread interleavings. The trace logs are then fed to a test oracle based classifier, which divides them into good traces and bad traces. Finally, a customized invariant inference engine in `Daikon` takes the subset of the traces (e.g., the good traces or bad traces) as input and returns the likely invariants as output.

Our new method has the capability of inferring both state invariants and transition invariants. A state invariant is a predicate that holds at a thread-local program location and is expressed in terms of the program variables at that location. For example in the program in Figure 1.2, the predicate `(A.x == 10)` is a state invariant at Line 4. In contrast, a transition invariant is a predicate that holds at the entry and exist points of an arbitrary code block and is expressed in terms of the two

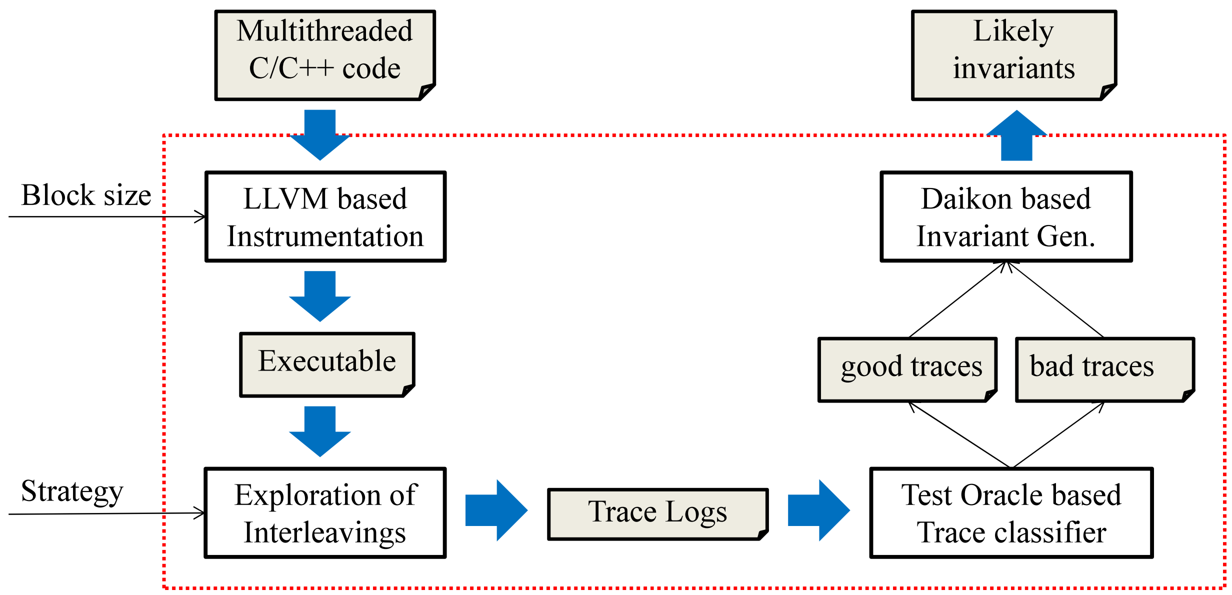

Multithreaded C/C++ code
Likely invariants
Block size
LLVM based Instrumentation
Daikon based Invariant Gen.
Executable
good traces
bad traces
Strategy
Exploration of Interleavings
Trace Logs
Test Oracle based Trace classifier

Figure 1.1: Our new dynamic invariant generation method.

```
Thread 1
1  p = & A;
2  ...
3  if (p!=NULL){
4    ...
5    p->x = 10;
6    ...
7  }
```

```
Thread 2
8  p = NULL;
9  ...
10 ...
11 ...
12 ...
13 ...
14 ...
```

Figure 1.2: The if-statement that is intended to be atomic.

copies of program variables at the entry and exit points, respectively. For example, in the program in Figure 1.2, the predicate `(p == orig(p))` is a transition invariant over the code block starting at Line 2 and ending at Line 3. Here `orig(p)` and `p` are values of variable `p` at the entry and exit points of this code block, respectively.

Transition invariants are important in multithreaded programs, since they often indicate whether the associated code block is *atomic* or *should be atomic*. Here, a code block is *atomic* only if the execution of instructions in this code block is not interfered by any conflicting instruction from a concurrent thread. In the above example, the transition invariant `(p == orig(p))` inferred from the code block starting at Line 2 and ending at Line 3 indicates that the value of pointer `p` did not change. Therefore, Line 2 and Line 3 belongs to the same atomic region. Although the baseline inference engine in `Daikon` can generate transition invariants, it only works at the method entry and exit points. Whereas our method has the capability of generating transition invariants over arbitrary code blocks. which our tool infers likely transition invariants is set by the user. This is important for multithreaded programs since the intended or actual atomic regions may not always coincide with the method boundary. Indeed, most of the atomic regions that we have encountered in the experiments are either code blocks inside a function body, or span across several methods.

In addition to generating a larger number of high-quality invariants as compared to existing methods such as `Daikon`, we also propose a new method for leveraging these invariants to diagnose concurrency errors and to infer likely atomic regions in the source code. In terms of error diagnosis, we can compare and contrast invariants learned from the good traces and bad traces – the difference is often useful in revealing the root cause of a concurrency bug. In terms of inferring atomic regions, we start by generating likely transition invariants for code blocks of increasing size. If the generated transition invariants holds in the original software code (which we check by manual inspection at this moment), then we can conclude that it indeed is an atomic region.

We have implemented our new methods and conducted experimental evaluation on open source multithreaded C/C++ programs. Our result shows that the state and transition invariants generated by our method are of significantly higher quality than existing methods such as `Daikon`; and at the same time, these invariants are useful in the context of diagnosing concurrency errors and inferring atomic regions.

To sum up, this thesis makes the following contributions:

- We develop a new method for dynamically generating high-quality state and transition invariants for multithreaded C/C++ programs.
- We propose new methods for leveraging these invariants to diagnose concurrency errors and to infer likely atomic regions in the software code.
- We implement the proposed methods and evaluate them on a set of public-domain benchmarks to demonstrate their effectiveness.

1.3 Organization

The remainder of this thesis is organized as follows.

Chapter 2, shows the technical foundations, needed to express the content of the thesis more formally.

Chapter 3, details the process of the dynamic invariant generation for concurrent programs and the algorithms associated to each individual stages.

Chapter 4, presents the two main usages of the generated invariants.

Chapter 5, shows the complete experimental results, that we have done with R_Tool, to compare it with `Daikon` as well as to evaluate other various features of this tool.

Chapter 6, provides in-depth step wise details, that user has to follow to install and run R_Tool, in various modes.

Chapter 7, concludes the thesis, with the note to the future extension of this work.

Chapter 2

Preliminaries

2.1 Multithreaded Programs

A multithreaded program consists of a set of *shared* variables SV and a set of threads $T_1 \ldots T_m$. Each thread T_i is a sequential program with a set of *thread-local* variables LV_i, where $1 \leq i \leq m$. Let st be an instruction. An execution instance of st is called an *event*, denoted $e = \langle tid, l, st, l' \rangle$, where tid is the thread index, l and l' are the thread program locations before and after executing st. An event is said to be *visible* if it access a shared variable or a thread synchronization object (mutex lock or condition variable). Otherwise, the event accesses only thread-local variables and it is said to be *invisible*. During systematic thread interleaving exploration and execution trace logging, invisible events will be ignored.

We model each thread T_i as a state transition system M_i. The transition system of the program, denoted $M = M_1 \times M_2 \times \ldots \times M_m$, is constructed using standard interleaving composition. Let $M = \langle S, R, s_0 \rangle$, where S is the set of global states, R is the set of transitions, s_0 is the initial state. Each state $s \in S$ is a tuple of thread program states. Each transition $e \in R$ is an event from one of the m threads.

An execution trace of M is a sequence $\pi = s_0 \overset{e_1}{\to} s_1 \overset{\dots}{\to} s_n$, where $s_0, \ldots, s_n$ are the states and $e_1, \ldots, e_n$ are the events. We shall use special event **halt** to denote normal program termination, and special event **abort** to denote faulty program termination. Event **abort** corresponds to a failed `R_assert()` statement.

Therefore, an event from thread T_i may have one of the following types:

- **halt**, which denotes normal program termination;
- **abort**, which denotes faulty program termination;
- $fork(j)$ for creating child thread T_j, where $j \neq i$;
- $join(j)$ for joining back thread T_j, where $j \neq i$.
- $lock(lk)$ for acquiring lock lk;
- $unlock(lk)$ for releasing lock lk;
- $signal(cv)$ for setting signal on condition variable cv;
- $wait(cv)$ for receiving signal on condition variable cv;
- $read(v)$ for reading of shared variable v;
- $write(v)$ for writing to shared variable v;
- $mEntry(method)$ for entering a method call;

- $mExit(method)$ for returning from a method call;
- $bEntry(block)$ for starting a code block;
- $bExit(block)$ for ending a code block.

Here, $mEntry()$ and $mExit()$ are supported by existing dynamic invariant generation tools such as `Daikon`, whereas $bEntry()$ and $bExit()$ are new additions in our method.

The reason why thread synchronization events are modeled is that we need to control the execution order of these events during thread interleaving exploration. In this work, we target multithreaded C/C++ applications written using the POSIX threads. We assume that the program uses thread creation and join, mutex locks, and condition variables for synchronization. Locks and condition variables are the most widely used primitives in mainstream platforms. If other blocking synchronizations are used, they also need to be modeled as events in a similar fashion. By doing this, we will be able to know at each moment during the program execution, which threads are in the blocking mode (*disabled*) and which threads are in the execution mode (*enabled*).

A currently enabled thread becomes disabled if (i) it attempts to execute $lock(lk)$ while lk is held by another thread; (ii) it attempts to execute $wait(cv)$ while the signal on cv has not yet been set; or (iii) it attempts to execute $join(j)$ while the child thread T_j is still running. Similarly, a currently disabled thread becomes enabled if (i) another thread releases the lock lk by executing $unlock(lk)$, (ii) another thread sets the condition variable cv by executing $signal(cv)$, or (iii) the waited child thread T_j terminates. It is important to compute the sets of enabled and disabled threads at run time, since attempts to schedule disabled threads while postponing the execution of enabled threads may lead to artificial deadlocks.

2.2 Dynamic Invariant Generation

Likely invariants generated from a program dynamically are properties that hold over the execution traces produced by a set of test cases. As such, it is not guaranteed that they would hold for all possible executions of the program. Furthermore, the invariant inference engine often uses a statistical analysis and, in theory, is neither sound nor complete. In practice, the number of invariants that can be generated, as well as the likelihood of them being true invariants, greatly depends on the quality of the test suite.

`Daikon` is a highly successful dynamic invariant discovery tool. It supports programs written in various programming languages including C/C++, C#, Eiffel, F#, Java, Lisp and Visual Basic. For each of these languages, `Daikon` provides a front-end tool for code instrumentation to add self-logging capability to the target program. All these front-end tools prepare the program to generate trace logs in a common format, which are then fed to the same backend invariant inference engine.

For example, the C/C++ front-end in `Daikon` is called Kvasir, which works directly on the compiled binary code. In this work, we have developed a new instrumentation tool based on the popular `LLVM` compiler platform to replace Kvasir. The main advantage of our new instrumentation tool is to leverage the large number of static program analysis procedures implemented in `LLVM`, which can potentially increase the performance of the dynamic analysis.

As we have mentioned earlier, `Daikon` do not have the capability of diversifying the execution traces in the presence of multithreading, and as a result, may generate many bogus invariants. Furthermore, `Daikon` is effective in generating linear invariants of the form ($ax + by < c$), but weak in generating more complex invariants such as polynomial invariants ($c_0 + c_1x^1 + c_2x^2 + ... < 0$) and array invariants. For the latest development along this line of research, please refer to the recent work by Nguyen *et al.* [26]. However, the focus of our work is not on improving the expressiveness of the generated invariants, but on improving the quality of the invariants that are relevant to concurrency. The vast majority of invariants generated by existing tools capture the sequential program behavior. But we want to focus only on invariants that refer to shared variables and therefore capture the behavior relevant to thread interference.

2.3 Motivating Examples

In this section, we will use a few examples to illustrate the problems with existing methods and highlight our main contributions.

First, consider the program in Figure 2.1, which has a global variable named `balance` being accessed by methods `getBalance()` and `setBalance()`. The third method `withdraw()` invoke the previous two methods to deduct a certain amount from `balance`. Since global variable `balance` is protected by a `setLock()` and `setUnlock()` pair every time it is accessed, there is no data-race. However, there can be atomicity violations. Although the method `withdraw()` is meant to be executed atomically – without other threads interleaved in between the calls to

```
int getBalance() {
  int bal;
  setLock();
  bal = balance;
  setUnLock();
  return bal;
}
void setBalance(int bal) {
  setLock();
  balance = bal;
  setUnLock();
}
void* withdraw(void *arg) {
  int bal = getBalance();
  bal = bal -100;
  setBalance(bal);
}
```

Figure 2.1: Motivating Example 1

`getBalance()` and `setBalance()` – it is not enforced by the programmer. For example, starting with `balance=400`. If two concurrent threads run `withdraw()` at the same time, the result may be either 300 or 200.

Existing dynamic invariant generation tools do not work well in this case. For example, if we instrument the compiled program with the Kvasir tool, and then run `Daikon`, most likely we will get a false invariant. The reason is that `Daikon` relies on the native execution environment to determine the program's thread schedule at run time. In this example, since the number of instructions in each thread is small (significantly smaller than the time slice of the Linux operating system), most likely, each thread will have ample time for completion before the OS forces a context switch.

Since the erroneous interleaving may never occur, `Daikon` may falsely report the invariant `::balance - orig(::balance) + 100 = 0`, where `::balance` denotes the value of `balance` at the exit point of `withdraw()` and `orig(::balance)` denotes the value of `balance` at the entry point.

This is not a true invariant, and it can make the developer believe that the execution of `withdraw()` is atomic despite of the fact that it is not the case. In other words, it has masked the symptom of a concurrency bug.

Our new method, in contrast, relies on a systematic concurrency testing tool called `Inspect` to diversify the thread interleavings before feeding the execution traces to the invariant inference engine. For the example in Figure 2.1, we will be able to cover both the good interleaving (leading to `balance=200`) and the bad interleaving (leading to `balance=300`). Consequently, the invariant inference engine will be able to produce the correct transition invariant `::balance < orig(::balance)`. This is the correct result since the best we can infer from the executions of this program (two threads running `withdraw()` concurrently) is that `balance` decreases.

There are many possible applications for the invariants such as `(::balance - orig(::balance) + 100 = 0)` and `(::balance < orig(:balance))`. In this thesis, we focus on two specific applications: diagnosing concurrency errors, and inferring atomic regions.

For error diagnosis, we rely on a user provided test oracle to distinguish failing executions from passing executions. During software testing, it is reasonable to expect the user to provide a test oracle, which separates bad test runs from good test runs. For our running example, assume that the oracle asserts that `(balance==200)` must hold at the end of the execution. This is illustrated in Figure 2.2. For the buggy program in Figure 2.1, if the method `withdraw()` is executed atomically by both threads, the assertion would pass; but if the method `withdraw()` is not executed atomically, the assertion would fail.

```
int main(void)
{
  ...
  pthread_create(&t1,0,withdraw,0);
  pthread_create(&t2,0,withdraw,0);
  ...
  pthread_join(t1,0);
  pthread_join(t1,0);
  R_assert(balance==200);  // test oracle
  return 0;
}
```

Figure 2.2: The main function for the example in Figure 2.1.

If we run our new invariant generation method on the passing traces only, according to our previous discussion, it would report the transition invariant `(::balance - orig(::balance) + 100 = 0)`. In contrast, if we run our new invariant generation method on the failing traces only, it would report the transition invariant `(::balance < orig(::balance))`. Presenting the discrepancy between these two set of results (invariants generated from passing runs versus invariants generated from failing runs) will help the user localize the root cause of the concurrency failure.

For atomic region inference, we present only the transition invariants obtained from the passing runs to the user. For our running example, the transition invariant generated for the code block Lines 27-32 is `(::balance - orig(::balance) + 100 = 0)`. To determine if the code block is *intended to be atomic*, we compute the thread-local transition relation of this code block. For the method `withdraw()`, the transition relation projected to the global variables is represented as follows:

```
::balance = orig(::balance) - 100
```

Therefore, the transition invariant generated by our method matches the thread-local transition relation – it is another way of saying that, in all the passing test runs, the method `withdraw()` is

```
void* withdraw(void *arg) {
  setLock();
  int bal = getBalance();
  bal = bal - 100;
  setBalance(bal);
  setUnlock();
}
```

Figure 2.3: Fixing `withdraw()` for the example in Figure 2.1.

executed atomically.

One way to fix the atomicity violation bug in `withdraw()` is to protect the method body with another pair of `setLock()` and `setUnlock()` – we make sure that a *reentrant* lock is used here. This is illustrated in Figure 2.3. To confirm that this is indeed a correct fix, we can run our new dynamic invariant generation tool again, which would only encounter passing runs, based on which it reports the expected transition invariant `(::balance - orig(::balance) + 100 = 0)`.

```
typedef struct { int a, b, c;} Data;
Data *A[128] ;
Data *p = NULL;
void* thr1 (void *arg) {
  ...
  ...
  Data *tmp = p;
  if (tmp != NULL) {
    p->a = 100;
    p->b = 200;
    p->c = 300;
    R_assert(tmp->a == 100 && tmp->b == 200 && tmp->c == 300 );
  }
  ...
  ...
}
void * thr2(void *arg) {
  int i = rand() % 127+1;
  p =  A[i];
  return 0;
}
```

Figure 2.4: Motivating Example 2

Although existing tools such as `Daikon` can already infer state and transition invariants, there is a major limitation: they only generate such invariants at the method entry and exit points.

```
=======================================
..main.c_10_15_0():::ENTER
::p == 36497856
=======================================
..main.c_10_15_0():::EXIT
::p == orig(::p)
::p == 36497856
=======================================
..main.c_12_15_1():::ENTER
::p == 36497856
=======================================
..main.c_12_15_1():::EXIT
::p == orig(::p)
::p == 36497856
=======================================
..main.c_15_15_3():::ENTER
::p == 36497856
=======================================
..main.c_15_15_3():::EXIT
::p == orig(::p)
::p == 36497856
```

Figure 2.5: Generated Invariants by R_Tool for Figure 2.4

Unfortunately, atomic regions in multithreaded programs often do not coincide with the procedural boundaries. Our new method, in contrast, has the capability of generating state and transition invariants for code blocks of arbitrary size – as shown in Figure 1.1, the user can set the block size from the command line.

Consider our second motivating example, shown in Figure 2.4, which has a global array of struct type `Data`. Within `thr1()`, the three fields are intended to be updated atomically – the programmer's intent is captured by the test oracle in `R_assert()`. In our dynamic invariant generation tool, this test oracle allows us to distinguish passing test runs from failing test runs. The question here is how to infer the *intended* atomic region, which spans from Line 10 to Line 15. In this case, R_Tool's capability of generating invariant for arbitrary code blocks is crucial.

Figure 2.5 shows the section of R_Tool's output regarding the transition invariant. Here, we start with a code block size of 2, which means that for any code region that has two consecutive accesses of a shared object, we will try to generate the transition invariants. The partitioning of the source code into code regions was performed by our new `LLVM` based instrumentation tool, as shown in Figure 1.1, which gradually increases the number of load/store instructions over global variables in the code block, to allow the determination of the largest possible atomic code region.

For example, in Figure 2.5, `..main.c_10_15_0()` means that R_Tool is considering the code block from Line 10 to Line 15 in Figure 2.4, whereas `..main.c_12_15_1()` means that R_Tool is considering the code block from Line 12 to Line 15. In both cases, R_Tool is able to generate invariant `(::p==orig(::p))`, which indicates that the value of the pointer `p` is never changed in between. By comparing these automatically generated invariants with the transition relation of the source code, the developer can reach a conclusion that, for the test runs to pass, the code block from Line 10 to Line 15 must be kept atomic.

Chapter 3

Dynamic Invariants Generation for Concurrent Programs

In this section, we present our new dynamic invariant generation method, which consists of three main components: an LLVM based code instrumentation tool, a systematic interleaving exploration tool, and a customized inference engine in Daikon. The overall flow of our method is illustrated by Algorithm 1, which takes the C/C++ source code of a multithreaded program as input and returns a set of likely invariants as output.

Algorithm 1 R_Tool for dynamically generating invariant.

```
$inst\_output \leftarrow$ inspect_pass($src\_code$)
$inst\_output \leftarrow$ daikon_pass($inst\_output$)
$inst\_output \leftarrow$ spacer_pass($inst\_output, spacer\_size$)
$trace\_file \leftarrow$ gen_traces($inst\_output$)
$thrd\_mod\_traces \leftarrow$ trace_classfier($trace\_file$)
$invariants \leftarrow$ inv_inference($thrd\_mod\_traces$)
```

3.1 LLVM Based Code Instrumentation

We have developed an LLVM based front-end for instrumenting multithreaded C/C++ code. As shown in Algorithm 1, the instrumentation consists of three code transformation passes.

The first pass, named inspect_pass, takes the C/C++ code as input and returns an instrumented version of the code as output. Inside this pass, we first identify all the program points where the thread schedule needs to be controlled by the Inspect tool during systematic interleaving exploration. These program points include the calls to thread synchronization routines, the read and write operations on shared memory locations, and the calls to low-level built-in atomic instructions in GNU, such as the *compare-and-set (CAS)* operations. At each program points, we inject new code before these *visible* operations, to allow the control of the thread at run time by the systematic interleaving explorer.

The second pass, named daikon_pass, takes the instrumented code as input and returns another instrumented version of the code as output. Inside this pass, we inject new code to add self-logging capabilities to the program at the execution time. The trace log generated by the program will conform to the common file format as required by the backend invariant inference engine in Daikon. By default, this pass instruments the code only at the method invocation and response points, which is compatible with the default front-end tool in Daikon, called Kvaisr. However, our implementation also has the capability of optimizing the runtime performance of trace logging by limiting the number of program points to be considered. For example, if we pass `dumpppt|` through the command line, the instrumentation tool will report the names of all functions in each program module into a text file. A user may choose to edit the text file, remove some program points, and therefore reduce the logging overhead at run time.

The third pass, named spacer_pass, takes the previously instrumented code as input and returns the final version of the code as output. Inside this pass, we also inject new code to add self-logging capabilities at the boundary of arbitrary code blocks, not just at the procedural boundaries as in Kvasir. This is accomplished by the tool inserting hook function calls to the entry and exit points of these code blocks, which in turn take care of the trace generation at run time. This is an important pass because it allows our method to generate state and transition invariants for arbitrary code blocks, which are crucial for inferring atomic regions in the software code. The size of the code block, around which hook function calls be inserted, can be controlled by the user. More specifically, this pass requires an integer value as *spacer_size*, based on which the pass garners a bunch of load/store instructions over global variables and inserts hook function calls at boundary of this block of instructions.

Note that the events logged as a result of the second and third passes are kept in the same format. From the standpoint of the backend invariant inference engine, there is no distinction between a pair of entry and exit points for a method, and a pair of entry and exit points for an arbitrary code block. Therefore, the backend inference engine do not have to be changed drastically in order to infer invariants at the boundary of arbitrary code blocks. By varying the value of the spacer count, we can dynamically change the locations where state and transition invariants are generated. This will help us to detect likely atomic regions in the code. We will discuss this application in details in Section 4.2.

Since the input program is a multithreaded program, during our instrumentation, we need to couple the `thread_id` information with each program point. The `thread_id` information is derived from the `pthread_t` field of a POSIX thread. However, we have made it a unique and persistent number associated to each thread in the program. By persistent, we mean that the `thread_id` remains the same across different program executions, despite that the value of the `pthread_t` field changes due to the thread being loaded into different memory locations. This allows the `thread_id` to be coupled with the program seamlessly, based on which the backend inference engine can deduce indexed predicates such as `(::balance - orig(::balance) + thread_id*10 = 0)`. Such invariants will be useful in applications such as thread modular verification [14, 21].

3.2 Systematic Thread Interleaving Exploration

Due to interleaving explosion, in general, we cannot afford to explicitly enumerate all possible thread schedules while diversifying the execution traces for the backend inference engine. In this work, we rely on a set of advanced search strategies implemented in the `Inspect` tool to produce a subset of representative thread interleavings. These search strategies are based on the theory of partial order reduction [16, 13]. They group the possible interleavings of a concurrent program into equivalence classes, and then pick one representative from each equivalence class.

Equivalence classes are defined using Mazurkiewicz traces [24]. Two sequences of events are said to be in the same equivalence class if we can create one sequence from the other by successively permuting the adjacent and *independent* events. Two events are *dependent* if they are from two different threads, access the same memory location, and at least one of them is a write or modify operation; otherwise, they are *independent*. It has been proved [16] that in the context of verifying concurrent systems, exploring one representative from each equivalence class is sufficient to catch all deadlocks and assertion violations.

In this work, we will leverage a state-of-the-art algorithm implemented in the `Inspect` tool, called dynamic partial order reduction (DPOR [13]). Here, the dependency relation is computed dynamically at run time, which is the main reason why it can robustly handle realistic applications written in C/C++ languages; for such applications, statically computing the dependency would have been too difficult due to the widespread use of pointers, method calls, and heap allocated data structures.

In addition to DPOR, we also employ two more reduction strategies: preemptive context bounding (PCB [25]) and history-aware predecessor set (HaPSet [36]) reductions. The idea of using context bounding to reduce complexity of software verification was first introduced for static analysis [29] and later for testing [25]. It has since become an influential technique, as in practice many concurrency bugs can be exposed by interleaving with fewer context switches. In this strategy, named PCB, the interleaving explorer generate only those interleavings with less than or equal to k preemptive context switches, where k is a small integer of 2 or 3. HaPSet can be viewed as a further improvement over DPOR and PCB.

3.3 Customized Invariant Inference Engine

We customized the invariant generation engine in `Daikon` to focus on two types of invariants in a multithreaded program : the state invariants and the transition invariants, both of which are expressed in terms of shared variables. A state invariant is a predicate expressed in terms of variable values at the current program location. A transition invariant is a predicate expressed in terms of variable values both at the current location and at some previous program locations. Therefore, a transition invariant is capable of capturing the impact of a code block.

More formally, we consider a program P as a state transition system $P = \langle S, R, I \rangle$, where S is the set of states, $I \subseteq S$ is the set of possible initial states, $R \subseteq S \times S$ is the transition relation. In general, a transitional invariant [28], denoted T, is an over-approximation of the transitive closure of R restricted to the reachable state space, i.e., $R^+ \cap (R^*(I) \times R^*(I)) \subseteq T$. In other words, a transitional invariant captures the relation between the pre- and post-conditions of a set of instructions.

In concurrent systems, transition invariants are extremely important because they would conform to the transition relation of a sequential code block in the absence of thread interference, but would deviate from the transition relation in the presence of thread interference. In testing and verification of concurrent systems, we typically assume that the sequential part of the computation is correct but the concurrency control part of the program is potentially buggy. In such cases, transition invariants will be more important than most of the other invariants generated by the inference engine. To accurately generate transition invariants, we need to fed to the inference engine a diverse and representative set of execution traces. This is the main reason why, without the help of systematic exploration techniques, existing tools would produce many false invariants. R_Tool, in contrast, can robustly generate transition invariants by taking thread scheduling into consideration.

Besides thread interleaving, another reason why the inference engine may produce wrong results is that the good and bad executions are typically mixed in the input. To solve this problem, we have developed a trace classification module that separates the execution traces into two groups: the passing traces and the failing traces, based on a test oracle. Specifically, R_Tool provides a macro called `R_assert`, through which the programmers can write correctness conditions that the program is expected to satisfy. During the dynamic analysis, if the assertion fails, the corresponding execution trace will be classified as a bad trace. While inferring likely invariant, the engine may choose to consider only the bad traces, only the good traces, or all traces. Then, we can compare and contrast the invariants generated in these three scenarios. We shall demonstrate in the following sections that the resulting information is useful in applications such as diagnosing concurrency bugs and inferring likely atomic regions in the software code.

Chapter 4

Applications of the Generated Invariants

In this section, we illustrate two applications where the dynamically generated likely invariants from multithreaded programs would be useful. The first application is diagnosing the concurrency bug in a multithreaded program. The second application is inferring likely atomic regions in the software code.

4.1 Diagnosing Concurrency Errors

In this application, we assume that the multithreaded program has some assertions that would be satisfied in most cases, but may fail in some rare thread interleavings. In this case, the execution traces of the program are classified into two groups: the good traces, and the bad traces. In order to identify the root cause of the concurrency error, we leverage the two sets of invariants generated by R_Tool and compare them to identify the discrepancy. Our conjecture is that the discrepancy

```
 8 const int NUM = 2;
 9 int sum = 0;
10 int array[128];
11 int array_index = -1;
12 void *thrd(void * arg) {
13   int temp = array_index;
14   ......
15   temp = temp+1;
16   array_index = temp;
17   array[array_index] = 1;
18   ......
19 }
20 int main(){
21   .......
22   //After Joining all the NUM threads
23   for(i=0; i<NUM; ++i) {
24     sum += array[i];
25   }
26   R_assert (sum==NUM);
27   ......
28 }
```

Figure 4.1: Example for diagnosing concurrency errors.

will provide valuable information for us to understand why the concurrency error occurs.

More formally, let Inv_{good} be the set of likely invariants generated from the good traces only, and Inv_{bad} be the set of likely invariants generated from the bad traces only. Let $Inv_{\mathsf{diff}} = (Inv_{good} \setminus Inv_{all})$ be the result of the *set-minus* operation. That is, Inv_{diff} consists of invariants that are satisfied by the good traces but not the bad traces.

Consider the example in Figure 4.1, where a parameterized number of threads share a global array and index named `array_index`. `NUM` is the number of threads that execute method `thrd()` concurrently. The test oracle provided by the programmer is shown in Line 26, which states that the expected result is `(sum==NUM)`. The assertion passes in good runs where each thread executes method `thrd()` atomically, but fails in bad runs where threads interfere with each other. Depending on how they interfere with each other, the value for `sum` ranges from 1 to NUM.

When given the good traces, our R_Tool will be able to generate transition invariants as reported

```
=========================
..main.c_13_17_0():::ENTER
::array_index < ::array
=========================
..main.c_13_17_0():::EXIT
::array == orig(::array)
.....
::array_index - orig(::array_index) - 1 == 0
```

Figure 4.2: Transitional invariants generated for Figure 4.1.

in Figure 4.2. In particular, the invariant on the last line shows that the combined effect on running Lines 13-17 in Figure 4.1 over variable `array_index` is that it will be increased by one.

When given the bad traces, however, our R_Tool will not be able to generate the aforementioned transition invariant. Instead, it will generate `::array_index > orig(::array_index)`, which also covers cases in which `array_index` is increased by 2, 3, ...

By comparing the two sets of invariants, we will be able to see the difference: `::array_index = orig(::array_index)+1` versus `::array_index > orig(::array_index)`, which highlights the root cause of this concurrency error.

4.2 Inferring Likely Atomic Regions

In this application, we assume that certain code blocks are either atomic or *meant to be atomic*, even if the atomicity is not properly enforced in the latter case. However, these code blocks are not known to us a priori, and we would like to identify them automatically with the help of the dynamically generated transition invariants.

When the transition relation of a code region is consistent with the transition invariant generated for the same code region, we say that section has been executed atomically. Furthermore, if the transition invariant is generated from good traces only – and they are not satisfied by the bad traces – we say that the code region is *meant to be atomic*. If the transition invariant is generated from all traces, we say with high confidence that the code region is *atomic*. For example, the method `thrd()` in Figure 4.1 and the method `withdraw()` in Figure 2.1 are *meant to be atomic*, whereas the fixed version of `withdraw()` in Figure 2.3 is *atomic*.

Another important notion in this context is the *maximal* atomic region. Although Lines 13-15 can be regarded as a likely atomic region, it is embedded in Lines 13-17, which is a larger atomic region. Since R_Tool can vary the size and location of the code regions for which it generates transition invariants, we are able to check code blocks of increasing number of instructions. Algorithm 2 shows the process of iteratively increasing the `spacer_size`, until we reach a block of instructions where the transition relation is no longer consistent with its transition invariant.

Algorithm 2 Computing the maximal likely atomic region.

```
$spacer\_size \leftarrow 1$
do
    $spacer\_size \leftarrow spacer\_size + 1$
    $tr\_inv \leftarrow \mathsf{R\_Tool}(src\_code, spacer\_size)$
    $res \leftarrow \mathsf{chk\_consistency}(tr\_inv, TR)$
while $res$ = true
```

Computing the Transition Relation The transition relation of an instruction is the relation between pre- and post-condition of that instruction. Similarly, the transition relation of a block of instructions is the relation between their pre- and post-conditions. Therefore, the transition relation, when being projected to the shared variables only, shows the cumulative effect of all the

```
                   Acc_1||Acc_2

       Transfer_x              Transfer_y
----------------------      -----------------------
1: Acc_1 = Acc_1-x           1': Acc_2 = Acc_2-y;
   ...                            ....
2: Acc_2 = Acc_2+x;          2': Acc_1 =  Acc_1+y
   ...                            ....
```

Figure 4.3: The bank account transfer example

instructions in the code block.

Consider a banking example for transferring fund from one account to another, shown in Figure 4.3. Here, `Transfer_x()` is called by one thread and `Transfer_y()` is called by another thread, where `x` and `y` as (thread-local) parameters, respectively. In a good run, the entire method has to be executed atomically by each thread, without interference from the other thread. The transition relation of the first statement in `Transfer_x()` is `Acc_1 = orig(Acc_1)-x`. Similarly, the transition relation of the entire method is `Acc_1+Acc_2 = orig(Acc_1+Acc_2)`. The transition relation for `Transfer_y()` is computed similarly.

At this moment, the transition relation is computed by the user and then provided to R_Tool. We plan to automate this process in the future, by leveraging existing static program analysis tools such as BLAST [21], which can compute transition relations using symbolic methods.

Chapter 5

Experiments

We have implemented the proposed methods in a software tool, called R_Tool , which can handle unmodified C/C++ code written for the Linux/Pthreads platform. R_Tool consists of an `LLVM` based instrumentation tool, an `Inspect` based interleaving explorer, a Java based trace classifier, and a customized inference engine in `Daikon`. We implemented R_Tool by writing 2126 lines of new C++ code, 776 lines of Java code, 229 lines of Python code, respectively.

We have evaluated our new methods on a set of public domain benchmark examples. Our targeted applications are low level systems code, such as multithreaded implementations of concurrent data structures and synchronization primitives. Due to this reason, we have extended the existing infrastructure in `Inspect` to handle the GNU built-in atomic instructions, which are crucial in directly and atomically accessing the shared memory. Our experimental evaluation is designed to answer the following research questions:

- How good is the quality of the invariants generated by our new method, especially when compared to existing tools such as Daikon?
- How useful are the invariants in the context of diagnosing concurrency errors and inferring likely atomic regions?

We have evaluated our new methods on two sets of concurrency related benchmarks. The SV-COMP'14 examples come from the 2014 Software Verification Competition [34], consisting of implementations of low level algorithms such as mutual exclusion and concurrent data structures. The SCTBench benchmark example simulates a multithreaded banking transaction process. The characteristics of these benchmark examples are summarized in Table 5.1. Note that some of these benchmarks, such as Dekker, are inherently non-terminating. To apply dynamic analysis, we have modified the source code of the test program slightly in order to turn the infinite loops to finite loops, while maintaining the program's intent.

5.1 Results

Table 5.2 shows the results of applying both R_Tool and Daikon to the benchmark examples. Column 1 shows the program name, Column 2 shows the number of program points, and Column 3 shows the number of shared variables. Since Daikon only generates invariants at the method entry and exit points, for comparison purposes, we have taken only the results of R_Tool at the same set of method entry and exit points – otherwise, R_Tool would always generate a lot more invariants than Daikon. Columns 4-7 show the statistics of running Daikon, including the total number of invariants reported, and among them, the true invariants and the bogus invariants, as well as the execution time in seconds. Columns 8-11 show the statistics of running R_Tool , including the total number of invariants reported, and among them, the true invariants and the bogus invariants, as well as the execution time in seconds. For both `Daikon` and R_Tool , we have manually classified the reported invariants (total) into true invariants (correct) and bogus invariants (incorrect). Due to the use of systematic interleaving exploration, R_Tool in general took more time to complete, but did not report any bogus invariants. In contrast, a large number of invariants reported by `Daikon` are bogus.

Table 5.3 shows the statistics of the invariants generated by R_Tool in more details. Since we are not comparing the result against `Daikon` (baseline), we will consider the invariants generated at all possible program points, not just the method entry and exit points. Therefore, the total number of invariants reported in this table is larger than that of Table 5.2. Here, we first classify the invariants generated by R_Tool into two groups: Transition invariants over shared variables and the rest (called the Regular Invariants). For each transition invariant reported by R_Tool , we also manually inspect the source code to see if we can manually construct the same transition invariant. If the reported transition invariants matches the manually generated ones, in the last column, we put *yes*; otherwise, we put *no*.

Our results show that, except for *Sync01*, R_Tool is always able to generate the correct transition invariants over shared variables. The reason why R_Tool cannot generate one of the two transition invariants for *Sync01* is because of the way the test program in this benchmark is written. Part of the code for *Sync01* is shown in Figure 5.1 (left), where the waiting thread at Line 20 only gets the

```
19  while (num > 0)
20    cond_wait(&empty,&m);
21  orig = num;
22  temp = num;
23  temp = temp+1;
24  num = temp;
```

```
40  while (num == 0)
41    cond_wait(&full,&m);
```

Figure 5.1: Sync_01: part of the code for the two threads.

signal to proceed when `num` is set to 0 by another thread, shown in Figure 5.1 (right). Even if we try to run the tests with different values for `num`, this code block (Lines 21-24) will never get the transition invariant `::num - orig(::num)-1 =0`, because the inference engine (in `Daikon` as well as in R_Tool) needs different values of `num` in order to generate a linear invariant. (Recall that, on a two dimensional plane, two points are needed to determine the line.) Thus, it fails to detect the transitional invariant in this case.

Table 5.4 shows the result of using the generated invariants to infer atomic regions. Column 1 shows the name of the benchmark. Column 2 shows the number of atomic regions inferred from the code. Columns 3-5 show the minimum, maximum, and average size of these atomic regions. We have manually inspected all reported atomic regions, and checked whether they are correct or not. Column 6 shows whether the atomic regions are indeed correct. Except for few cases, we have successfully detected all available atomic regions in the software code. However, in *Sync01, IncTrue, IncCas, LogProcSweep*, although the system detects only one atomic region, there are in fact two atomic regions.

Finally, we evaluate the scalability of our new method. In general, the computational complexity of R_Tool depends on the length of each individual execution trace, as well as the number of execution traces fed to the backend inference engine. Among them, the number of execution traces is the main bottleneck, since by using an `Inspect` based interleaving explorer, in theory, we subject R_Tool to the potentially exponential number of thread interleavings in the worst case, even with the help of partial order reduction techniques. In practice, however, we can still use more aggressive reduction heuristics, such as PCB and HaPSet (covered in Section 3.2), during interleaving exploration. The question is whether such interleaving reduction decreases the quality of the generated invariants.

Table 5.5 shows the comparative results of using the three different exploration techniques in R_Tool . Here, we compare both the number of runs that each strategy requires to explore the interleaving space, as well as the number of transition invariants generated by R_Tool as a result of the exploration. The first thing that we notice is that there is a significant reduction in the number of runs by changing from DPOR to PCB and HaPSet. But, at the same time, we notice that the number of transition invariants actually remains the same for all benchmarks. In order words, using a more aggressive (and unsound) exploration strategy (such as PCB or HaPSet) does not seem to noticeably reduce the effectiveness of the invariant generation algorithm. Figure 5.2 shows the scalability of the each algorithm, with increasing number of threads in the targeted program. This experimental results have been found, by running R_Tool with on Indexer Benchmark from SVCOMP'14 and varying the number of threads in the program. The x abscissa, denotes the number of threads present in the system, where as the ordinate denotes the number of iteration that `Inspect` tool is required, to find the necessary interleavings. Though DPOR times out very quickly but HaPSet, shows significant lower rate of increment in the number of the number of iterations, with the increasing number of threads in the system.

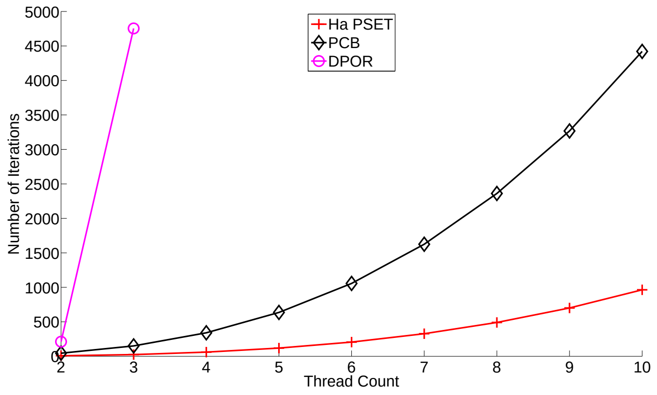


Figure 5.2: Scalability Comparison of DPOR, PCB and HaPSet algorithm.

5.2 Discussions

5.2.1 Threats to validity.

There are two threats to validity in our experimental evaluation. The first threat is the degree to which the SVCOMP'14 and SCTBench programs used in our experiment represent realistic applications. Although we believe that they are representative of the kind of low level systems code that we target, there can be discrepancies. Another threat is that, for applications, we only evaluated the use of transition invariants over shared variables for inferring likely atomic regions. It remains unclear if the vast majority of the remaining invariants are useful (we suspect that they are mostly sequential properties and therefore have nothing to do with concurrency, but this needs to be investigated further).

5.2.2 Limitations.

Our method is limited in three aspects. First, it relies on the user to provide test oracles in the form of `R_assert` conditions so that R_Tool can separate good traces from bad traces. One way to address this limitation is to design a static code analysis method for inserting assertions automatically. Second, our method relies on systematic interleaving exploration to improve the diversity of the execution traces. Although our experiments shows that the current implementation of interleaving exploration is good enough for low-level systems code whose size is small, scalability may become a problem for larger programs. Third, our method for automatically inferring likely atomic regions still relies on the user's manual inspection to confirm the results. One way to remedied this limitation is to use an automated verification procedure to formally prove the correctness of the result.

Table 5.1: The characteristics of the benchmark examples used in our experimental evaluation.

Benchmark Name	LoC	Program Points	Description
FibBench	55	`t1,t2,main`	This program computes the Fibonacci series using two shared variables, `i` and `j`, and two threads.
Lazy01	52	`thread1,thread2, thread3`	Method `thread1` increases the variable `data` by 1 whereas `thread2` increases it by 2. Method `thread3` checks the result. All methods are protected by lock.
Stateful01	55	`thread1, thread2, main`	Shared variables `data1` and `data2` are accesses by two threads. All accesses are protected by lock.
Sync01	64	`thread1, thread2, main`	Shared variable `num` is accessed by two threads and is protected by lock and condition variables.
Dekker	68	`thr1,thr2,main`	This is an implementation of Dekker's algorithm for for mutual exclusion.
Lamport	119	`thr1,thr2,main`	This is an implementation of Lamport's algorithm for mutual exclusion for two threads.
Peterson	55	`Peterson`	This is an implementation of Peterson's algorithm for mutual exclusion.
TimeVarMutex	55	`allocator,deallocator, main`	This is scaled-down version of the inode allocation procedure. Here, `inode` is a shared variable set `allocator` and reset by `deallocator`. Accesses are protected by two locks.
Szymanski	106	`thr1,thr2,main`	This is an implementation of the mutual exclusion algorithm by Szymanski for two threads.
IncTrue	55	`thr1,main`	This is an implementation of the atomic increment operation for an integer variable. The atomic increment is implemented using a shared integer flag.
IncCas	60	`thr1,main`	This is an another implementation of the atomic increment operation for an integer. This implementation mimics the atomic compare-and-set operation.
IncDec	58	`inc,dec,main`	A shared integer variable is atomically incremented and decremented by two threads, for which mutual exclusion is maintained by using a shared integer flag.
IncDecCas	89	`inc,dec,main`	Another implementation of the shred integer incrementing and decrementing procedure. Here, the increment and decrement follow a compare_and_set pattern.
ReOrder	63	`setThread, checkThread, main`	This example has two threads running `setThread` and `checkThread`, respectively. A shared variable is set by `setThread` and checked by `checkMethod`.
AccountBad	61	`deposit, withdraw, main, check_result`	This example simulates a banking transaction application, where `deposit` adds a fixed amount of money to the shared variable `balance`, whereas `withdraw` removes the same fixed amount. The third method `check_result` checks the result. All accesses to the shared variable are protected by lock.
LogProcSweep	83	`new_log_entry, add_log_entry, logging`	This sample code from Apache Foundation is a logger that maintains a list of Log entry. It has two threads, one of which adds new log entries and other resets the list.
pfscan	932	`worker`	This is a multithreaded implementation of the file system scanning utility in Linux for scanning a particular path in the file system.

Table 5.2: Invariant Comparison between `Daikon` and R_Tool

Benchmark			Daikon (baseline)				R_Tool (new)			
Name	Program points	global vars	Total Invs	Correct Invs	Incorrect Invs	Time (s)	Total Invs	Correct Invs	Incorrect Invs	Time (s)
FibBench	3	2	17	15	2	2.9	30	30	0	258.6
Lazy01	4	1	15	15	3	2.6	14	14	0	5.5
Stateful01	3	2	22	15	8	2.7	18	18	0	4.0
Sync01	3	1	14	12	2	1.6	9	9	0	2.0
Dekker	3	4	39	24	20	1.7	53	53	0	62.9
Lamport	3	5	48	17	31	5.0	76	76	0	9.1
Peterson	3	4	39	13	26	1.7	51	51	0	11.4
TimeVarMutex	3	3	27	23	4	1.6	22	22	0	1.7
Szymanski	3	3	29	14	15	1.6	35	35	0	40.4
IncTrue	2	2	15	11	4	2.2	19	19	0	12.6
IncCaS	2	1	18	16	2	2.7	10	10	0	3.8
IncDec	3	6	57	31	26	3.4	75	75	0	21.3
IncDecCaS	3	3	24	24	0	2.9	40	40	0	3.6
Reorder2	3	4	47	38	9	2.5	56	56	0	273.1
AccountBad	4	6	68	50	18	3.8	78	78	0	8.2
LogProcSweep	7	1	41	39	2	2.5	45	45	0	3.1
pfscan	1	1	0	0	0	1.4	1	1	0	11.4

Table 5.3: The quality of invariants generated by R_Tool .

Benchmark	R_Tool Gen			Manual Inspection	
Name	Regular Invs	Transition Invs	Total Invs	Transition Invs	Matched
FibBench	99	2	101	2	yes
Lazy01	39	2	41	2	yes
Stateful01	58	4	62	4	yes
Sync01	31	1	32	2	no
Dekker	198	2	200	2	yes
Lamport	399	2	401	2	yes
Peterson	188	2	190	2	yes
TimeVarMutex	51	2	53	2	yes
Szymanski	205	2	207	2	yes
IncTrue	33	1	34	1	yes
IncCaS	5	1	6	1	yes
IncDec	229	2	231	2	yes
IncDecCaS	25	2	27	2	yes
Reorder2	113	2	115	2	yes
AccountBad	174	2	176	2	yes
LogProcSweep	61	1	62	1	yes

Table 5.4: Size of atomic regions under dynamic invariants.

Benchmark	Atomic regs	Min size	Max size	Avg size	Correct
FibBench	2	2	5	3.5	yes
Lazy01	2	2	5	3.5	yes
Stateful01	4	3	5	4	yes
Sync01*	1	3	9	6	yes
Dekker	2	2	4	3	yes
Lamport	2	2	4	3	yes
Peterson	2	2	4	3	yes
TimeVarMutex	2	2	2	2	yes
Szymanski	2	2	10	6	yes
IncTrue*	1	3	8	5.5	yes
IncCaS*	1	2	19	10.5	yes
IncDec	2	2	4	3	yes
IncDecCaS	2	2	9	5.5	yes
Reorder	2	2	3	2.5	yes
AccountBad	2	2	6	4.5	yes
LogProcSweep*	1	2	3	2.5	yes

Table 5.5: Invariants generated with different search strategies.

Name	DPOR		PCB (k=2)		HaPSet	
Benchmark	# Runs	# Invs	# Runs	# Invs	# Runs	# Invs
FibBench	71624	2	252	2	2576	2
Lazy01	160	2	260	2	132	2
Stateful01	48	4	68	4	32	4
Sync01	7	1	16	1	4	1
Dekker	3896	2	15	2	519	2
Lamport	392	2	21	2	321	2
Peterson	730	2	15	2	150	2
TimeVarMutex	4	2	19	2	4	2
Szymanski	5980	2	23	2	3015	2
IncTrue	848	1	84	1	1140	1
IncCaS	132	1	100	1	256	1
IncDec	1936	2	240	2	1352	2
IncDecCaS	32	2	76	2	56	2
Reorder2	19425	2	832	2	71	2
AccountBad	160	2	260	2	36	2
LogProcSweep	9	1	84	1	36	1

Chapter 6

R_Tool User's Manual

R_Tool is a dynamic invariant generation tool for sequential and concurrent C/C++ programs. It is built upon the `LLVM` compiler front-end (*llvm-3.2*), the Inspect concurrency testing tool (*inspect-0.3*), a Java based trace classifier (*SimpleDeclParser*) and the Daikon dynamic invariant generation tool (*Daikon*).

6.1 Installation

The installation process consists of the following steps:

1. Check out the R_Tool git repository.
2. Install `LLVM` and the `DaikonPass`.
3. Install `Inspect`.

4. Install the trace classifier `SimpleDeclParser`.

5. Install `Daikon`.

6. Update the path in the `exports.sh`.

7. Install `Qt` (Optional – required only for using the GUI).

6.1.1 Checking out the R_Tool git repository

Check out the git repository into a local folder by using the following command,

```
git clone git@github.com:arijitvt/RTool.git
```

This will create a folder named R_Tool , which is a self-contained package with all dependencies. For rest of this document, R_Tool folder will denote this folder and all the relative paths will be calculated with respect to this folder.

6.1.2 Installing `LLVM` and the `DaikonPass`

Inside the R_Tool folder, there exists our customized version of `LLVM` (llvm-3.2.src.tar.bz2) compressed file. In order to install this version of `LLVM`,

- First, untar the package by running the command

  ```
  tar -xvf llvm-3.2.src.tar.bz2.
  ```

  This will create a folder named llvm-3.2.src and place the source inside the folder.

- Copy the Daikon folder (which is inside the LLVM_PASS folder) to the existing folder called R_Tool /llvm-3.2.src/lib/Transform.
- Then, go to the *build* directory, which should contain `conf.sh` file. Provide the executable permission to the script by running the command
  `chmod +x conf.sh.`
- Run the `conf.sh` from the build folder. This will build `LLVM` and all its code transformation passes that are needed to run R_Tool . The resulting files are inside the build/Release+Debug+Asserts folder.
- Finally, update the **LLVM** variables to have the complete path point to *Release+Debug+Asserts* inside the R_Tool /exports.sh.

6.1.3 Installing the `Inspect` Tool

Since our customized version of the `Inspect` tool uses the SMT solver, we need to install `smtdp` before the installation of the `Inspect`. Go to the *smt_dp* folder and run `make` from there. It will compile and also copy the required files to the proper location.

Installing `Inspect` is also straightforward. Go to the inspect-0.3 folder and run the `make` command from there. It will compile and create the `inspect` executable inside the R_Tool /bin folder. To test whether the `make` command has been executed properly, check if you have the `inspect` executable in the R_Tool /bin folder.

After the installation, remember to set the **insp** path variable with the location of `inspect-0.3` folder location in the R_Tool /exports.sh file.

6.1.4 Installing the `Trace Classifier`

The jar file for the trace classifier is present in the R_Tool /jars folder. Update the **CLASSPATH** with the location of the parser.jar in R_Tool /exports.sh.

Since the source code for that jar file is already present in the R_Tool /SimpleDeclParser folder, if needed, one can also create the jar file again from the java source, using the standard jar file making process, and then update the **CLASSPATH** in the R_Tool /exports.sh with the appropriate location of jar file. .

6.1.5 Installing `Daikon`

The original guide for *Daikon* installation can be found in this url

`http://plse.cs.washington.edu/daikon/download/doc/daikon.html`.

Below is a brief summary of the major steps associated with this process.

- Download and install `JDK 1.7` (I've found that jdk 1.6 does not work for the current version of daikon).
- Update the **JAVA_HOME** in the bashrc with the installation directory of `JDK1.7`.

- Populate **DAIKONDIR** variable with the top directory of the daikon folder and export the variable.

- Export the `daikon.bashrc` by running

  `source $DAIKONDIR/scripts/daikon.bashrc`

- Then run `make` from the daikon folder. This will install `Daikon` and update all the required the path.

6.1.6 Updating the path in `exports.sh`

During the installation process, we have already updated various path variables. You may want to double check if all the path variables are set properly at this stage.

Once that is done, update the **PATH** variable with the location of the R_Tool /bin folder and update the **daikon** variable with the source folder where the DaikonPass resides. DaikonPass folder exists in the llvm-3.2.src/lib/Transforms/Daikon folder. This folder should contain the hook.h and hook.c files, which contains the source for the **hook_assert** macro that we have defined.

6.1.7 Installing `Qt`

A user friendly interface for R_Tool has been developed using `Qt 5` framework. If the user choses to use the Gui option of R_Tool , then it is necessary to install `Qt 5` framework. `Qt` comes with standard installer , which can downloaded from the following url `http://qt-project.org/downloads`.

6.2 Using R_Tool

R_Tool can be used in three modes.

1. Use the existing daikon infrastructure through a quintessential interface.
2. Use the R_Tool infrastructure to generate invariant at all function entry-exit and around the global variable access locations.
3. Generate the invariants at various location varied dynamically using command line control.

Before using R_Tool , check if you can have access to the three scripts: controller.py , runner.py and r_tool.py . If it is not possible to access these scripts, then update the path variable properly in the R_Tool /exports.sh and check again.

6.2.1 R_Tool as the Frontend for `Daikon`

R_Tool can be used to run `Daikon` as it is, in a more comprehensive way and using much shorter commands. To compile the target program (the program for which you want to generate invariants) and dump the program points, run the command

```
controller.py daikon dump
```

This compiles the target program, creates the executable, and also creates the program points and/or variables of interest as well. You may want to edit these files, for example, to reduce the number of program points and/or variables of interest (on which the invariants are generated).

Once this step is over, then we should be able to run the instrumented target program, using the command

```
controller.py daikon rep #
```

Here # should be the run count/iteration count – that is, count of the time we are running the target program before giving the traces to Diakon for invariant generation.

Since daikon needs at least four different values of the variables for generating linear invariants, you may need to edit the program source so that the program runs under different program inputs (different values of global variables during different runs).

That is the main reason why we need to supply this iteration count. This process will also dump all the thread-modular trace files inside a trace folder, called *arijit*, where one can generate invariant using the *Daikon's* backend by running the command from *arijit* folder,

```
java daikon.Daikon *.dtrace
```

This invariant generation step using the *Daikon's* backend is same for all the other methods listed here.

6.2.2 R_Tool for Generating Invariants from Concurrent Programs

The main usage of the R_Tool is to generate invariants for not only sequential C/C++ programs, but also concurrent C/C++ programs. For this purpose, we should invoke R_Tool in a similar fashion as discussed in the previous section. However, we also need to run the target program under the supervision of the `Inspect` systematic concurrency testing tool.

To compile the program for `Inspect` and at the same time generate program points, use the below mentioned command.

```
controller.py rtool comp
```

Once the instrumented program is generated, we can use

```
controller.py rtool run $ # %
```

Here, $ corresponds to the iteration count (as mentioned in the previous section), and # corresponds to the algo choice. Since `Inspect` can use three strategies (DPOR, PCB or HaPSet) for systematic exploration of the possible thread interleavings, we can use the command-line argument to dynamically choose the algorithm `(0 -> DPOR, 1 -> PCB, 2->HaPSet).` This number corresponds to the algo choice. When we choose PCB or HaPSet, % denotes the number as required for PCB (context bounds) and HaPSet (size of HaPSet vectors). For DPOR, this number should be set to 0.

6.2.3 R_Tool for Inferring Likely Atomic Code Regions

R_Tool 's atomic section determination algorithm, relies on oracle information provided by the user. This requires additional manual work to be done, other than the automated instrumentation framework of R_Tool . For that purpose, user should include the `hook.h` and provide the oracle information in *hook_assert* macro. Compilation and linking of the program, has been taken care of automatically by the controller scripts. When R_Tool is used to automatically infer atomic sections in the program code, the compilation should follow the exact same procedure as above, except that the first argument will be 'cs':

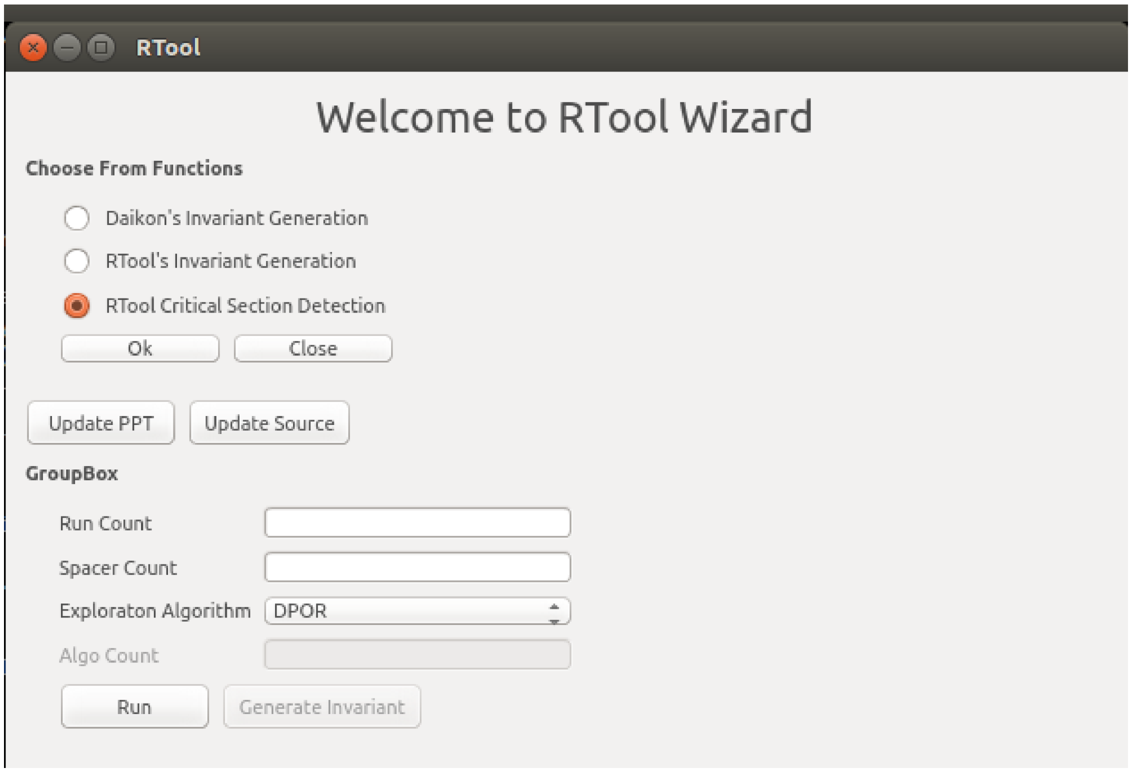
```
controller.py cs comp
```

To run the program,

```
controller.py cs run $ @ # %
```

The rest of the three special characters `($,\#,%)` remain the same as described in the previous section, except that @ should be replaced with the spacer count, which set the number of global instructions that should be included in the same atomic region, allowing the user to dynamically vary the locations of instrumentation to determine the atomic region.

All these utilities can be executed though a GUI built based on the qt framework. An sample of this GUI can be seen in figure 6.1.


RTool

Welcome to RTool Wizard

Choose From Functions

- ( ) Daikon's Invariant Generation
- ( ) RTool's Invariant Generation
- (●) RTool Critical Section Detection

Ok | Close

Update PPT | Update Source

GroupBox

Run Count	
Spacer Count	
Exploraton Algorithm	DPOR
Algo Count	

Run | Generate Invariant

Figure 6.1: Screenshot of the User Interface for R_Tool

Chapter 7

Conclusions and Future Work

In this thesis, I have presented a new method for dynamically generating likely invariants from multithreaded programs, as well as leveraging these invariants to diagnose concurrency errors and infer likely atomic regions in the software code. Compared to existing methods, which often work well on sequential programs but become ineffective on multithreaded programs, our new method can generate a significantly large number of high-quality invariants.

We have implemented the new methods in a software tool, called R_Tool , which builds upon the *LLVM* compiler frontend, the *Inspect* interleaving exploration tool, and a customized invariant inference engine inside *Daikon*. Our experimental results show that the proposed methods are effective in generating invariants from multithreaded C/C++ programs, and these invariants are useful in applications such as diagnosing concurrency related bugs and identifying atomic code regions.

This work can be extended in many ways. First, the *state and transition invariants* generated by R_Tool have strong potential in boosting the performance of software verification tools that are based on *predicate abstraction*. For example, tools such as SLAM [2, 1] and BLAST [20, 21] rely on proper predicates to construct effective models for sequential and concurrent software. However, computing these predicates often requires many costly refinement iterations. Invariants generated from R_Tool can be provided to these tools as candidate predicates, which has the potential of significantly accelerating the refinement process.

Another possible extension to R_Tool is to increase the program path coverage of the execution traces by leveraging automated test input generation techniques. Right now, the user needs to diversify the test inputs manually. Having a fully automated method for generating quality test input can increase the diversity of the traces, which in turn will increase the quality of the generated invariants. Finally, right now the user still needs to provide the test oracle that R_Tool can use to classify execution traces into good traces and bad traces. It would be very useful if we could automate this process, for example, by developing a software tool that can automatically insert test oracles into the software code.

Bibliography

[1] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 268–283, 2001.

[2] T. Ball and S. K. Rajamani. The SLAM Toolkit. In *International Conference on Computer Aided Verification*, pages 260–264, 2001.

[3] M. Boshernitsan, R. Doong, and A. Savoia. From daikon to agitator: Lessons and challenges in building a commercial tool for developer testing. In *International Symposium on Software Testing and Analysis*, pages 169–180, New York, NY, USA, 2006. ACM.

[4] M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar. Automatic verification of integer array programs. In *International Conference on Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2009.

[5] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 427–442,

Berlin, Heidelberg, 2006. Springer-Verlag.

[6] F. Chen, T. Serbanuta, and G. Rosu. jPredictor: a predictive runtime analysis tool for java. In *International Conference on Software Engineering*, pages 221–230, 2008.

[7] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In *In Computer Aided Verification*, pages 420–432. Springer Verlag, 2003.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[10] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *International Conference on Software Engineering*, pages 281–290, New York, NY, USA, 2008. ACM.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, New York, NY, USA, 1999. ACM.

[12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, Dec. 2007.

[13] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 110–121, New York, NY, USA, 2005. ACM.

[14] C. Flanagan and S. Qadeer. Thread-modular model checking. In *International SPIN Workshop on Model Checking Software*, pages 213–224, 2003.

[15] C. A. Furia and B. Meyer. Fields of logic and computation. chapter Inferring Loop Invariants Using Postconditions, pages 277–300. Springer-Verlag, Berlin, Heidelberg, 2010.

[16] P. Godefroid. Model checking for programming languages using verisoft. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, New York, NY, USA, 1997. ACM.

[17] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty. Iodine: a tool to automatically infer dynamic invariants for hardware designs. In *Design Automation Conference*, pages 775–778, 2005.

[18] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM.

[19] T. A. Henzinger, T. Hottelier, L. Kovács, and A. Voronkov. Invariant and type inference for matrices. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 163–179, Berlin, Heidelberg, 2010. Springer-Verlag.

[20] T. A. Henzinger, R. Jhala, and R. Majumdar. The blast software verification system. In *International SPIN Workshop on Model Checking Software*, pages 25–26, 2005.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *International Conference on Computer Aided Verification*, pages 262–274, 2003.

[22] M. Li. A practical loop invariant generation approach based on random testing, constraint solving and verification. In *International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, pages 447–461, Berlin, Heidelberg, 2012. Springer-Verlag.

[23] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, New York, NY, USA, 2006. ACM.

[24] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 279–324, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[25] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, 2008.

[26] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *International Conference on Software Engineering*, pages 683–693, Piscataway, NJ, USA, 2012. IEEE Press.

[27] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, New York, NY, USA, 2009. ACM.

[28] A. Podelski and A. Rybalchenko. Transition invariants. *Logic in Computer Science, Symposium on*, 0:32–41, 2004.

[29] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–24, 2004.

[30] E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *J. Symb. Comput.*, 42(4):443–476, Apr. 2007.

[31] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using gr&#246;bner bases. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 318–329, New York, NY, USA, 2004. ACM.

[32] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, pages 574–592, 2013.

[33] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 37–46, 2010.

[34] 2014 software verification competition. URL: http://sv-comp.sosy-lab.org/2014/.

[35] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 328–342, 2010.

[36] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *International Conference on Software Engineering*, pages 221–230, 2011.

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

[38] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 2005.