Detecting Bugs of Concurrent Programs With Program Invariants

Rong Wang, Zuohua Ding, Ning Gui, and Yang Liu

***Abstract*—Concurrency bug detection is a time-consuming activity in the debugging process for concurrent programs. Existing techniques mainly focus on detecting data race bugs with pattern analysis; however, the number of interleaving patterns could be huge, only the most suspicious write–read pattern is given, and an oracle is needed, which is not available in the operational phase. This paper proposes a program-invariant-based technique to detect a class of concurrent program bugs. By unit testing of the components of a concurrent program, we obtain a set of program invariants, which can be used as an oracle to obtain "bad" invariants when the program is online. By using the function call graph of the components and applying a reduction technique to the invariants, we find the candidates of suspicious functions and rank them. From the interactions among components, we analyze the causes to the concurrency bugs. Experimental results show that our proposed technique is effective in concurrency bug detection.**

***Index Terms*—Bug detection, concurrent program, function call graph, program invariants, suspicious rate.**

I. Introduction

CONCURRENT programs are becoming prevalent due to the proliferation of parallel hardware. The bugs in concurrent programs are hard to be detected since they are nondeterministic, i.e., they depend upon the occurrence of certain bad interleavings of threads and processes. Concurrency bugs can cause severe damages in the real world [15]. Therefore, effective bug-detection techniques to expose concurrency bugs are highly desirable.

Most existing concurrency bug detection work focuses on one subclass: data race detection [6], [7], [24], [26]. A data race occurs when two accesses, at least one of which being a write, from different threads to the same memory location execution without proper synchronization. Many dynamic detection tools have been proposed to address this problem. The basic idea is the pattern analysis. A data race bug is reported when memory accesses violate some write–read patterns. However, they are far from providing an effective solution for detecting concurrency bugs, since

1) the pattern analysis only gives the most suspicious write–read patterns, therefore serious efforts are still needed to reach and confirm the real bugs;
2) multithreaded programs with interactions amongst threads potentially have a large number of interleavings posing challenges in the analysis; and
3) it needs an oracle for the bug location, but no testing oracle is available in the operational phase.

To address the above issues, this paper presents an invariant-based technique to detect concurrency bugs. The workflow of our proposed approach is as follows. First, by performing unit testing of some selected components of a concurrent program, we obtain a set of program invariants as reference invariants using tool *Daikon*,[1] which can be used as an Oracle to get "bad" invariants when testing the whole concurrent program. Second, we use tools *Doxygen*,[2] *graphviz*,[3] and *AspectJ*[4] to build the function call graph of the selected component. Based on the graph, redundant or ineffective invariants are removed to simplify the future computation. Third, we traverse the function call graph to record all possible fault positions. Finally, for all the fault positions, we propose a formula to compute their suspiciousness rates and rank them as well according to the depths in the function call graph and the orders of function calls. By further analyzing the program invariants and the component interactions, we roughly locate the causes of concurrency bugs.

Using program invariants as a way to locate faults is already a proven approach [1], [2]. For example, Abreu *et al.* [1] used bit mask and range invariants to detect bugs, but it is for sequential program analysis. Lu *et al.* [18] detected concurrency bugs with interleaving invariants, but it focuses on bugs in a situation that during the correct runs, two consecutive accesses from one thread to the same shared variable are interleaved by an unserializable access from another thread.

We propose a new technique, which combines program invariants with searching strategies on the function call graph, to perform fault localization in concurrent programs via the suspiciousness ranking of possible fault positions. We apply our technique to detect various types of concurrency bugs, including order violation, atomicity violations, deadlock problem, and bad composition problem.

Manuscript received June 25, 2016; revised November 14, 2016 and January 15, 2017; accepted February 25, 2017. This work was supported by the National Natural Science Foundation of China under Grant 61210004 and Grant 61572441. Associate Editor: S. Li. *(Corresponding author: Zuohua Ding.)*

R. Wang, Z. Ding, and N. Gui are with the School of Information Science, Zhejiang Sci-Tech University, Hangzhou 310018, China (e-mail: 821990651@qq.com; zouhuading@hotmail.com; ninggui@qq.com).

Y. Liu is with the School of Computer Engineering, Nanyang Technological University, Singapore 639798 (e-mail: yangliu@ntu.edu.sg).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2017.2681107

[1] http://plse.cs.washington.edu/daikon/

[2] http://www.doxygen.nl

[3] http://www.graphviz.org

[4] http://www.eclipse.org/aspectj/

This technique can be applied in the following three situations to detect concurrency bugs:

1) A component communicates with other components, and it is well tested. We want to detect if this component gets infected by other components.
2) A set of interacting components of a program is well tested. If this program is extended to a more complicated program by adding some components in the set to the program, we need to know if this combination can work well.
3) A multithread program has been working well for some time, we need to check if there are some hidden errors.

Compared with the existing methods [18], [23], our technique has several advantages. First, our technique can be applied to detect not only specific bugs like atomicity violations or order violations but also other concurrency bugs like deadlock problem and bad composition problem, while the existing methods only focus on order violations or atomicity violations. Second, we do not need an oracle to check the output if some components come from the third side. Third, unlike other methods that only report violation patterns, the proposed technique directly reports fault positions in the code.

This paper is organized as the follows. Section II is the overview of our approach. Section III describes bug types and program invariants. Section IV presents how to detect bugs based on invariants. Section V shows the experimental results based on the examples from open resources. Section VI explains the treats to validity. Section VII is a discussion of related work. Section VIII concludes the paper.

II. Approach Overview

This section gives an overview of our approach. Fig. 1 shows the work flow of our method with four steps.

Step 1: Select a component from concurrent software. By a component we mean that it has a process or a thread, which can communicate with other components through message passing, or resource sharing. Let this component be A. Assume that this component has no bugs (at least we cannot find bugs from unit testing). By executing the inputs, we will obtain a set of invariants I_1 from a trace file from *Daikon*. These invariants are regarded as correct and the whole set of invariants can be used as the Oracle, namely reference invariants, for concurrent program testing. Meanwhile, we obtain a function call graph from the trace file.

Step 2: We test the concurrent program with a test suite which should cover the tests for component A. When running the test cases, we monitor those selected invariants and have another set I_2 of invariants for component A, which are called online invariants. Comparing online invariants I_2 with reference invariants I_1, we obtain the different invariants in I_2, which are regarded as bad invariants.

Step 3: Combining the function call graph, we develop a reduction technique which will be used to reduce redundant or ineffective invariants for both sets of invariants to short our detecting time.

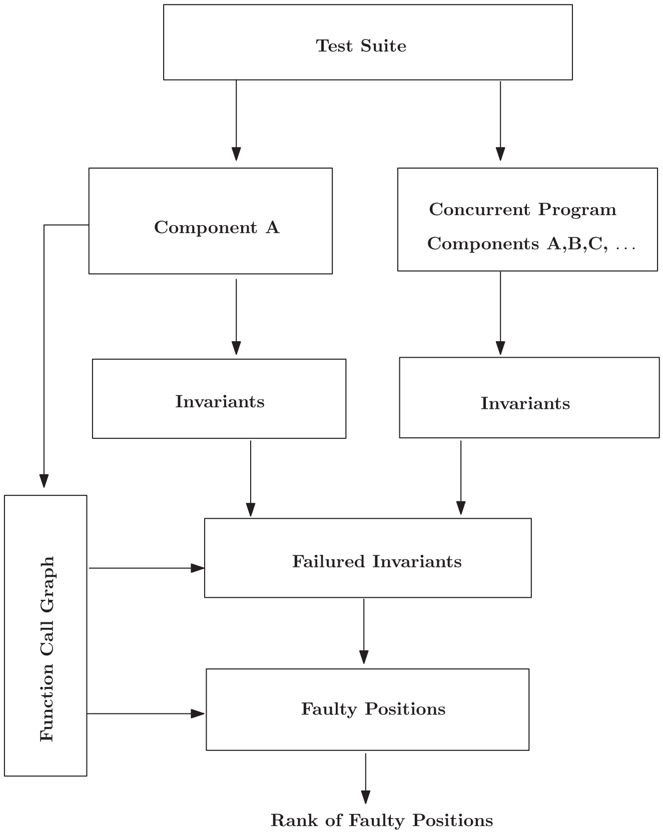
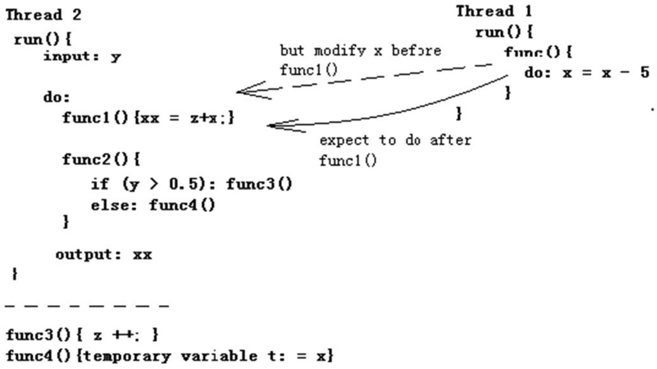


Fig. 1. Work flow of our method.



```
Thread 2                                      Thread 1
 run(){                                         run(){
    input: y                                       func(){
                                                     do: x = x - 5
    do:                                            }
      func1(){xx = z+x;}                        }

      func2(){
         if (y > 0.5): func3()
         else: func4()
      }

     output: xx
 }

- - - - - - - -

func3(){ z ++; }
func4(){temporary variable t: = x}
```

but modify x before
func1()
expect to do after
func1()

Fig. 2. Running example.

Step 4: Based on the bad invariants and function call graph, we compute the suspicious rates of the functions and determine which one has the highest probability to cause concurrency bugs.

Fig. 2 is a running example. We will locate the bugs of component A. The bugs can exist in component A or exist in other components that infect component A. In order to pinpoint all the suspicious parts among the components, we do unit testing for all components, and build the function call graph cover these components.

Thread 1 and thread 2 share variable x. func() in Thread 1 is supposed to be executed after func1() in Thread 2 reads x with correct value 1. However, a bug appears when Thread 1 runs faster and decreases x by 5 before func1 reads x. Set $x = 1$, $y \in U(0, 1)$, here U is a uniform distribution, and z is an integer evenly selected from [0, 9].

In this example, we perform unit testing for Thread 2 by considering those that may be affected by the surroundings to obtain the reference invariants. Then we do integrated testing for thread 2 and its real surroundings to obtain the online invariants.

The technique proposed in [23] can detect faulty data-access patterns among threads by monitoring the shared single variables. However, it cannot be applied to our running example. The reason is as follows. A faulty pattern $W_1 - R_2$ tells us that thread 1 modifies the shared variable x before thread 2 reads it. But this pattern could not tell us which parts of thread 1 and thread 2 incorrectly interact. Moreover, if the concurrency bugs relate to multiple variables, the technique from [23] may not work.

III. Bug Types and Program Invariants

In this section, we first describe the types of concurrency bugs to be detected in this paper, and then describe program invariants and show how to obtain them as well.

A. Concurrency Bugs

Park *et al.* [23] introduce their formal notations for key concurrency violations: atomicity violations and order violations. Our concurrency bugs are somewhat different from theirs. An *unexpected behavior problem* occurs if the invariants from a program testing differ from the reference invariants from a component testing. The concurrency bugs reflect that the program has unexpected behavior. We consider the following concurrency bug types.

1) Order Violation: Multiple sequential threads access to a shared object or variable in a wrong order, such that an unexpected behavior of interleaving appears. Our running example is an example of order violation.

2) Atomicity Violation: An atomicity violation occurs when an atomic method or a code block from a thread is interrupted during its execution and encounters an unexpected behavior problem while it is expected to be executed without being interrupted by other threads. An atomicity violation is somewhat like an order violation. But atomicity violations require atomic-access assumptions while order violations do not need them.

3) Deadlock Problem: A deadlock is a situation where two or more threads compete to access to shared resources but each block has to wait for the others to release the required resources. We focus on deadlock problems that trigger an unexpected behavior problem.

4) Bad Composition: A method is not well written for extensions and is used by multiple threads. If the number of threads is small, we may not experience incorrect behaviors. As more threads are involved, an unexpected behavior may happen due to the incorrect method.

B. Program Invariants

An invariant is a property that holds at a certain point or points in a program; these are often used in assert statements, documentation, and formal specifications. Examples include being constant ($x = a$), nonzero ($x \neq 0$), being in a range ($a \leq x \leq b$), linear relationships ($y = ax + b$), ordering ($x \leq y$), functions from a library ($x = fn(y)$), containment ($x \in y$), sortedness (x is sorted), and many more.

As Hangal and Lam [9] mentioned, most programs obey many invariants, many of which are not documented anywhere, and in fact, may not be known even to the original writers of the code. In this paper, we assume that we have a training mechanism to determine correct invariants. There are tools developed for finding dynamic invariants, such as *Daikon* by Ernst *et al.* [5], and DIDUCE by Hangle and Lam [9]. Also, Hangal *et al.* [10] developed a tool called IODINE to automatically extract dynamic invariants such as request-acknowledge pairs and mutual exclusion between signals for hardware designs. In this paper, we used *Daikon* as the invariant detector.

Daikon is a highly successful invariant generation tool for sequential programs written in languages such as Java and C++.

First it instruments a program by the front end that converts data from other format into Daikon's input format; Then, executes the program with an existing test suite, and gets a data trace (.dtrace) file; Finally, it finds patterns by learning the trace file to obtain likely invariants. Here data can come from any source, but Daikon is typically used to find invariants over variable values in running programs.

The following steps are needed to generate invariants.

1) Generate a test suite. Test cases are generated from software requirements. The selection of test cases must satisfy some coverage criteria, such as statement coverage, condition coverage, path coverage, and branch coverage. Path coverage covers all decisions and thus can generate accurate program invariants. However, it is hard to do so if a program is big. Therefore, we choose branch coverage which ensures that every judgement statement, e.g., if/else, will be tested. For the running example, the refined test suite contains 20 test cases covering all the branch of the code.
2) Enable the invariants in *Daikon*. By default, most of the invariants are enabled. If more invariants are needed, we just switch the invariants to enable.
3) Run the program with the test cases and get a trace file. The trace file contains six pairs of Enter/Exit.
4) Obtain the invariants by running *Daikon* invariant detector.

For the running example, Table I shows the reference invariants for Program A from unit testing.

For example, look at the invariant $\mathrm{this.xx} - \mathrm{this.z} - 1 == 0$ at SA.func1() ::: EXIT. When we run a test case through the front end Chicory, we will get a trace file. For example, we have a trace file that records the monitored variables at SA.func1():::EXIT, where: $\mathrm{this.y}$ is around 0.4, $\mathrm{this.xx} == 9$, and $\mathrm{this.z} == 8$. Expression $\mathrm{this.xx} - \mathrm{this.z} - 1 == 0$ is held for this trace file. After we run all 20 test cases, we obtain

TABLE I
REFERENCE INVARIANTS FOR PROGRAM A IN THE RUNNING EXAMPLE

Functions	Reference Invariants
SA.run():::EXIT	this.xx > orig(this.xx)
SA.func1():::EXIT	this.xx > orig(this.xx)
	this.xx - this.z - 1 == 0
SA.func2():::ENTER	this.xx - this.z - 1 == 0
SA.func2():::EXIT	this.xx - orig(this.z) - 1 == 0
SA.func3():::ENTER	this.xx - this.z - 1 == 0
SA.func3():::EXIT	this.xx == this.z
	this.xx - orig(this.z) - 1 == 0
SA.func4():::ENTER	this.xx - this.z - 1 == 0
SA.func4():::EXIT	this.xx - this.z - 1 == 0

TABLE II
INVARIANTS FOR PROGRAM A IN THE RUNNING EXAMPLE

Functions	Invariants from unit testing	Invariants from integration testing
SA.run():::EXIT	this.xx > orig(this.xx)	-
SA.func1():::EXIT	this.xx > orig(this.xx)	-
	-	this.xx != this.z
	this.xx - this.z - 1 == 0	-
SA.func2():::ENTER	-	this.xx != this.z
	this.xx - this.z - 1 == 0	-
SA.func2():::EXIT	-	this.xx != orig(this.z)
	this.xx - orig(this.z) - 1 == 0	-
SA.func3():::ENTER	-	this.xx != this.z
	this.xx - this.z - 1 == 0	-
SA.func3():::EXIT	this.xx == this.z	-
	-	this.xx <= this.z
	-	this.xx != orig(this.z)
	this.xx - orig(this.z) - 1 == 0	-
	-	this.z - orig(this.z) - 1 == 0
SA.func4():::ENTER	-	this.xx != this.z
	this.xx - this.z - 1 == 0	-
SA.func4():::EXIT	-	this.xx != this.z
	this.xx - this.z - 1 == 0	-

20 trace files. Then we use Daikon to analyze all the trace files, and find that $\text{this.xx} - \text{this.z} - 1 == 0$ is held for all of them. Thus we have an $\text{this.xx} - \text{this.z} - 1 == 0$.

IV. FAILURE DETECTION USING INVARIANTS

A. Generating Failure Invariants

A program is faulty if itself has bugs or it is affected by other faulty programs. For its correct version, we will get reference invariants with some test inputs. The failure invariants are the differences between reference invariants and the invariants from faulty version with the similar test inputs.

Table II shows the difference between the reference invariants and the online invariants for Program A, which implies that there is an order violation. The column *Invariant from unit testing* lists reference invariants from unit testing, and the column *Invariant from integration testing* lists online invariants from integrated testing when all threads interact. *If an invariant is not in the correct invariant set, then it is considered as a failure invariant*.

Note that the invariant: $\text{this.xx} - \text{this.z} - 1 == 0$ in $SA.\text{func1}()$ appears in unit testing but not in integration testing, while invariant: $\text{this.xx}! = \text{this.z}$ appears in integration testing but not in unit testing. Although they are different, but the latter includes the former.

Remark IV.1: It is possible to get "false positive" invariants, i.e., the invariants that are violated on valid behaviors (i.e., the correct input/outputs) which is not captured in the training correct invariant set. Later, we develop a way to remove such false positive invariants.

Since the faults of software are reflected in variables, thus one may consider to monitor variables. However, it will be overhead if we monitor all variables. In fact, monitoring every variable may not be required, as only a subset of variables, known as collar variables [19], truly affect the outcome of a system in a meaningful way. Therefore, choosing the most critical variables becomes the main issue. In this paper, instead of monitor variables, we choose to monitor several types of invariants generated by Daikon, e.g., linear relations like invariant $\text{this.xx} - \text{this.z} - 1 == 0$ for the running example.

B. Filtering Out Unjustified Failure Invariants

We use Daikon to compare two sets of invariants, one from unit testing and the other from integration testing. The Daikon will print out the difference like $\langle \text{inv1}\{\text{conf1}\}, \text{inv2}\{\text{conf2}\}\rangle(\text{type}, \text{relationship})$, where inv1 is from unit testing with confidence level conf1, and inv2 is from integration testing with confidence level conf2, type is the type of both invariants and relationship is the relationship between the two invariants.

type has total six values: UInt, U!Int, NInt, N!Int, Bin, and Ter, where U represents unary, Int represents interesting, !Int represents uninteresting, N represents nullary, Bin represents binary, and Ter represents Ternary. relationship has total 12 values: SJU, SUJ, SJJ, SUU, DJJ, DJU, DUJ, DUU, MJ, JM, MU, and UM, where S represents same but with different confidence level, U represents statically unjustified, J represents statically justified, D represents different, and M represents missing.

We filter out all the same or unjustified or uninteresting invariants by removing all U related terms. Now, the failure invariants have the form $\langle \text{inv1}, \text{inv2}\rangle(\text{type}, \text{relationship})$, where $\text{type} \in \{\text{Bin}, \text{Ter}, \text{UInt}, N\text{Int}\}$, $\text{relationship} \in \{DJJ, MJ, JM\}$. Note that for confidence level, we use the default value 0.99.

For convenience, we give the following definition.

1) Passing Failure Invariants: Passing failure invariants are the failure invariants appearing at the entry or the exit of a certain procedure which are passed from the previous procedures that generate them.

Later, we will discuss how to filter passing failure invariants to enhance the efficiency of our fault locating technique.

C. Building Static Call Graph

While generating invariants, we can build a *static call graph* from traces to describe the call relations among functions. A static call graph is a tuple $S = (s, E, T)$, where s is a node that has outputs but has no inputs, E is a set of nodes each of which

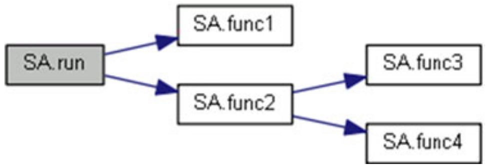
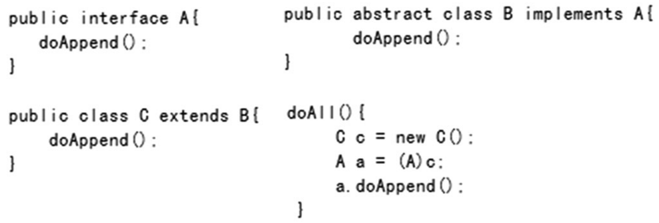


Fig. 3. Static call graph for running example.



```
public interface A{             public abstract class B implements A{
    doAppend();                         doAppend();
}                               }

public class C extends B{       doAll(){
     doAppend();                      C c = new C();
}                                     A a = (A)c;
                                      a.doAppend();
                                  }
```

Fig. 4. Example for improving static call graph.

has inputs but has no outputs. T is a set of arcs that connects nodes. For example, if functions f_j and f_i are two nodes in a static graph and f_j calls function f_i, then we have $f_j \rightarrow f_i$. We can use tools, e.g., *Doxygen* and *graphviz*, to generate static call graphs in a static way. However, the static call graphs generated by *Doxygen* usually cannot contain all the invoked functions from the main function in a concurrent program. We may miss some program points on this static call graph that might lead to enlarging the scope of suspicious code block. Static call graph will be used later to reduce extra invariants. Fig. 3 shows the static call graph for the running example.

Similar to the static call graph, if function f_i is always invoked after function f_j is executed, we denote this order by $f_j \rightarrow f_i$.

We found that a static function call graph is not accurate in actual execution and is not consistent with program points generated by Daikon, thus, we make an improvement for the static call graph. We use an example in Fig. 4 to show why we need the improvement.

In Fig. 4, abstract class B implements interface A, class C extends B and overrides method doAppend(), and the main function calls function doAll(). The problem is that Doxygen will simply draw an arrow between doAll() and A.doAppend() like doAll() $\rightarrow$ A.doAppend(). However, in Java programs, when a subclass overrides the method of its super class or interface, the substance of subclass will always invoke its own method no matter it is casted into super class or interface. So we expect to have something like doAll() $\rightarrow$ C.doAppend(), where we replace A.doAppend() with C.doAppend() from initial static call graph. Both Daikon and Aspectj record the program points that are actually executed, which implies that we need to have the actual function call graph. So, an improvement is required as Fig. 5 shows.

In Fig. 5, two graphs on the left of the arrow are generated by Doxygen. The red node represents the function like A.doAppend() that is not the actual executed function in programs. And the green node represents the function like C.doAppend() that would actually be executed in programs when the red one is invoked accordingly. Then we replace the red node with the green node as shown in the graph on the right of the arrow.

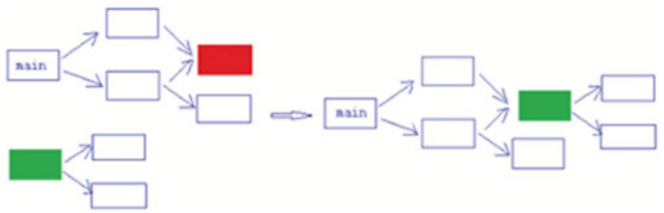


Fig. 5. Improving static call graph.

The replacement procedure is as follows: Use depth-first search to traverse on the old static call graph. When visiting a node, if it does not appear in any call tables, then replace it with the corresponding function appearing in the call tables. Therefore, we obtain a more accurate function call graph.

Remark IV.2: Our technique depends on the tools to generate invariants. If no instrumentation is used for invisible concurrency problems, one possible solution is to build an interaction graph among components and apply spectrum-based fault localization (SFL) [28] approach to the nodes of the graph.

D. Computation for Call Order and Depth

We use *Aspectj* to record function call orders in call tables. Since the concurrent program behaviors are not determined and some functions can be invoked for many times, we execute the program for n times and record all the call orders for functions, then compute the average call order for each function. In addition, we regard the depth of a position on a static call graph as the distance started from the root node when first visiting this position on the static call graph. The followings are the formulas to compute call orders for the positions:

$$\text{order}(f) = \left(\sum_{i=1}^{n}\sum_{j=1}^{m_i} o_{ij}\right) \Bigg/ \left(\sum_{i=1}^{n} m_i\right), \tag{1}$$

$$\text{order}(f_{ji}) = (\text{order}(f_i) + \text{order}(f_j))/2 \tag{2}$$

where f is a function, $f_j \rightarrow f_i$, f_{ji} is the position between function f_j ::: ENTER and f_i ::: EXIT marked as (f_j, f_i), o_{ij} is the call order of this function's *j*th call in the *i*th execution, m_i is the number of call times in the *i*th execution. The following is a formula to compute depths for positions:

$$\text{depth}(f_{ji}) = (\text{depth}(f_i) + \text{depth}(f_j))/2. \tag{3}$$

Sometimes, we can easily recognize the function call order in a program, so we simply mark the call order without using the formula above. Table III shows the function call orders for the running example.

E. Program Invariants Reduction

Since our method to detect program bugs is based on program invariants, we should remove *false positive invariants* to make the prediction more accurate. All the program points that have

TABLE III
RELATIVE CALL ORDER FOR FUNCTIONS IN THE RUNNING EXAMPLE

Functions	Relative call order
SA.run()	0
SA.func1()	1
SA.func2()	2
SA.func3()	3
SA.func4()	3

failure invariants are skeptical. For the running example, all functions have failure invariants, but we do not know which one is the most suspicious one. Hopefully we have an reduction technique that can reduce the influence of propagations of some invariants and thus to eliminate some skeptical functions. We propose such technique as follows.

The idea is to reduce the redundant invariants for each function on a static call graph. Consider two categories of redundant failure invariants:

1) Failure invariants appear at *Exit* points of a function and its parent function at the same time. We can remove these failure invariants from parent function but keep them in its children function. The reason is that these failure invariants from parent function could be caused by the child function.
2) Failure invariants appear at *Enter* and *Exit* points of a function at the same time. Functions like SA.func4() from the running example may not generate these failure invariants, which are generally transmitted from the parent function.

Next, we show how to remove redundant failure invariants. Assume that program invariants are obtained in the block from $\mathrm{function}$ ::: ENTER to $\mathrm{function}$ ::: EXIT. Consider any function f_i, and its invariants between f_i ::: ENTER and f_i ::: EXIT, $i = 1, \ldots, n$. Let

En_{f_i} be the set of the invariants of f_i ::: ENTER,

Ex_{f_i} be the set of the invariants of f_i ::: EXIT.

Our technique consists of the following operations:

$$En_{f_i} = En_{f_i} \setminus Ex_{f_i} \tag{4}$$

$$Ex_{f_i} = Ex_{f_i} \setminus En_{f_i} \tag{5}$$

$$Ex_{f_j} = Ex_{f_j} \setminus \cup_i \{Ex_{f_i} : f_j \rightarrow f_i\} \tag{6}$$

where symbol $f_j \rightarrow f_i$ means that function f_j calls function f_i or function f_i is always called after function f_j is executed. It is determined by the static call graph.

We use (4) and (5) to filter out the same failure invariants at both f_i ::: ENTER and f_i ::: EXIT. These failure invariants are not originally caused by f_i ::: ENTER or f_i ::: EXIT, and probably are the passing failure invariants transported from the parent functions of f_i. We use (6) to filter out the same repeated failure invariants at f_j ::: EXIT which appear in the block from f_j ::: EXIT to f_i ::: EXIT. Note that f_j calls f_i. There are redundant failure invariants at f_j ::: EXIT and f_i ::: EXIT, thus we only keep the same failure invariants at *EXIT* point of latest called function.

We add two additional rules to the reduction process. In addition to exclude the same failure invariants, we also exclude the following failure invariants.

1) If we have failure invariant at the entry place of a given program point as

$\langle y = f(x), \mathrm{null}\rangle$(or $\langle \mathrm{null}, y = f(x)\rangle$)

where *f* is a polynomial, x and y satisfy this formula; and at the same time, we have this failure invariant at the exit place of the same program point as

$\langle y = f(\mathrm{orig}(x)), \mathrm{null}\rangle$(or $\langle \mathrm{null}, y = f(\mathrm{orig}(x))\rangle$)

where $\mathrm{orig}(x)$ is the x upon the entry of the same program point; Then, we regard that the two failure invariants as the passing failure invariants and exclude them from program points.

Here is the explanation. If $\mathrm{orig(x)}$ appears at the exit of procedure, this procedure must have modified x in its body. Before the procedure executes, both x and y at entry point have already had the failure relationship as $y = f(x)$. After the procedure executes, y at the exit and x upon the entry remain the same relationship as $y = f(\mathrm{orig(x)})$. We see that y at the entry and y at the exit are the same value because x at the entry is the same as $\mathrm{orig(x)}$. It can be inferred that this failure relationship of exit might be simply infected by the failure relationship of the entry, and the relationship of entry might be simply passed from the previous false procedures. Thus, we infer that these two failure invariants are only passing failure invariants, and should be removed.

2) If we have the failure invariants at the entry of a given program point as $\langle y\triangle x, \mathrm{null}\rangle$(or $\langle \mathrm{null}, y\triangle x\rangle$), where $\triangle \in \{==, !=, >=<=, >, <\}$. At the same time, we have this failure invariant at the exit of the same program point as

$\langle y\triangle\mathrm{orig(x)}, \mathrm{null}\rangle$(or $\langle \mathrm{null}, \mathrm{y}\triangle\mathrm{orig(x)}\rangle$), where $\mathrm{orig(x)}$ is the x upon the entry of this same program point.

Then, we regard these two failure invariants as the passing failure invariants and remove them from program points. First, x at the entry and $\mathrm{orig(x)}$ at the exit are the same value. Whether this procedure decreases y or increases y or just does nothing to y, it is always the failure relationship at both entry and exit of the program point. So the two failure invariants might be only passing failure invariants, and should be removed.

To illustrate our technique, we apply the reduction procedure to the invariants of the functions in the running example as follows. We use func2() as the example to compute (4) and (5):

$$\begin{aligned} En_{\mathrm{SA.func2()}} &= \{\langle \mathrm{null}, \mathrm{this.xx!} = \mathrm{this.z}\rangle, \\ &\quad \langle \mathrm{this.xx} - \mathrm{this.z} - 1 == 0, \mathrm{null}\rangle\} \\ Ex_{\mathrm{SA.func2()}} &= \{\langle \mathrm{null}, \mathrm{this.xx!} = \mathrm{orig(this.z)}\rangle, \\ &\quad \langle \mathrm{this.xx} - \mathrm{orig(this.z)} - 1 == 0, \mathrm{null}\rangle\} \end{aligned}$$

where two-tuple $\langle \mathrm{inv1}, \mathrm{inv2}\rangle$ represents two invariants, inv1 from $En_{\mathrm{SA.func2()}}$ and inv2 from $Ex_{\mathrm{SA.func2()}}$.

In *Daikon*, orig(x) represents the value of variable x when entering procedure f_i ::: ENTER, and the value of x may be modified when appearing in procedure f_i ::: EXIT. We see that $\langle \mathrm{null}, \mathrm{this.xx!} = \mathrm{orig(this.z)}\rangle$ appears in $Ex_{\mathrm{SA.func2()}}$ and

$\langle \text{null}, \text{this.xx}! = \text{this.z}\rangle$ appears in $En_{\text{SA.func2()}}$. Since this.z at entry is the same as $\text{orig}(\text{this.z})$ at exit, SA.func2() cannot avoid to have such a failure relationship no matter it modifies this.xx or not. This kind of failure relationship that can be infected by previous procedures might be a passing failure invariant, so we perform reduction (4) and (5) for the two failure invariants. Thus

$$\begin{aligned}
&Ex_{\text{SA.func2()}} \cap En_{\text{SA.func2()}} \\
&= \{< \text{null}, \text{this.xx}! = \text{this.z} >, \\
&\quad \langle \text{this.xx} - \text{this.z} - 1 == 0, \text{null}\rangle\}. \\
&En_{\text{SA.func2()}} \\
&= En_{\text{SA.func2()}} \setminus (Ex_{\text{SA.func2()}} \cap En_{\text{SA.func2()}}) = \emptyset. \\
&Ex_{\text{SA.func2()}} \\
&= Ex_{\text{SA.func2()}} \setminus (Ex_{\text{SA.func2()}} \cap En_{\text{SA.func2()}}) = \emptyset.
\end{aligned}$$

Conclusion 1: At the beginning, $En_{\text{SA.func2()}} \neq \emptyset$ and $Ex_{\text{SA.func2()}} \neq \emptyset$, however, after reduction, $En_{\text{SA.func2()}} = Ex_{\text{SA.func2()}} = \emptyset$. This means that $\text{func2}()$ is not a suspicious function, we need to check other functions called by $\text{func2}()$. If we do not apply the reduction mechanism, we will stop at $\text{func2}()$ and move to its parent node and search others.

Similarly, we can compute (4) and (5) for all functions f_i for the example and obtain the refreshed En_{f_i} and Ex_{f_i}.

Next, we use $\text{run}()$ as the example to compute (3). For $\text{SA.run}()$, we have $\text{SA.run}() \rightarrow \text{SA.func1}()$ and $\text{SA.run}() \rightarrow \text{SA.func2}()$:

$$\begin{aligned}
Ex_{\text{SA.run()}} &= \{\langle \text{this.xx}\rangle \text{orig}(\text{this.xx}), \text{null}\rangle\}. \\
Ex_{\text{SA.func1()}} &= \{\langle \text{this.xx}\rangle \text{orig}(\text{this.xx}), \text{null}\rangle, \\
&\quad \langle \text{null}, \text{this.xx}! = \text{this.z}\rangle, \\
&\quad \langle \text{this.xx} - \text{this.z} - 1 == 0, \text{null}\rangle\}. \\
Ex_{\text{SA.func2()}} &= \emptyset. \\
&\quad Ex_{\text{SA.run()}} \cap Ex_{\text{SA.func1()}} \\
&= \{\langle \text{this.xx}\rangle \text{orig}(\text{this.xx}), \text{null}\rangle\}. \\
&\quad Ex_{\text{SA.run()}} \cap Ex_{\text{SA.func2()}} \\
&= \emptyset. \\
&\quad Ex_{\text{SA.run()}} \\
&= Ex_{\text{SA.run()}} \setminus \{(Ex_{\text{SA.run()}} \cap Ex_{\text{SA.func1()}}) \\
&\quad \cup (Ex_{\text{SA.run()}} \cap Ex_{\text{SA.func2()}})\} = \emptyset.
\end{aligned}$$

Conclusion 2: At the beginning, $Ex_{\text{SA.run()}} \neq \emptyset$. After reduction, $En_{\text{SA.run()}} = \emptyset$. This means that we need to continue to check other functions called by $\text{SA.run}()$. However, if we do not apply reduction mechanism, we will stop at $\text{SA.run}()$. This means that we obtain nothing since it is the root function.

Similarly, we compute (6) for all functions f_i and obtain the updated En_{f_i} and Ex_{f_i}. Finally, we obtain the reduced invariants as shown in Table IV.

TABLE IV
INVARIANTS AFTER REDUCTION TECHNIQUE FOR THE RUNNING EXAMPLE

Functions	Invariants Before	Invariants After
SA.func1():::EXIT	this.xx > orig(this.xx)	-
	-	this.xx != this.z
	this.xx - this.z - 1 == 0	-
SA.func3():::EXIT	this.xx == this.z	-
	-	this.xx <= this.z
	this.xx - orig(this.z) - 1 == 0	-
	-	this.z - orig(this.z) - 1 == 0

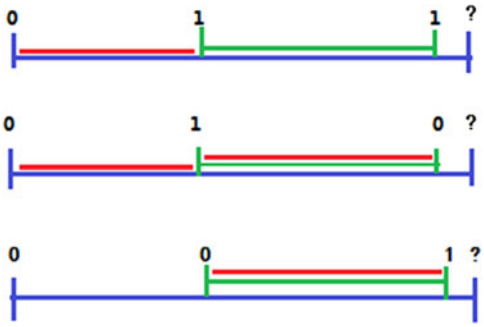


Fig. 6. Record from the static call graph. The green and blue segments represent functions and the blue calls the green. The left point of a segment is an entry, and the right point of the segment is the corresponding exit. Numbers 0 or 1 over the points of a segment are the failure flags. The red lines are the suspicious positions we should record.

F. Computing Possible Failure Functions

We will find possible failure functions and rank them based on the bad invariants obtained by comparing the reference invariants from unit testing with the invariants from integration testing.

For each function f_i, we use (EN_{f_i}, EX_{f_i}) as its failure flag, where

$$\begin{aligned}
EN_{f_i} &= 0, \text{if } \mathrm{En_{f_i}} = \emptyset;\ \text{otherwise}, \mathrm{EN_{f_i}} = 1 \\
EX_{f_i} &= 0, \text{if } \mathrm{Ex_{f_i}} = \emptyset;\ \text{otherwise}, \mathrm{EX_{f_i}} = 1.
\end{aligned}$$

Note, EN and EX are single failure flags with value 0 or 1, while En and Ex are sets of failure invariants. Then we apply depth-first search algorithm to the static call graph to select the suspicious positions. Accordingly, we can locate the bugs in suspicious functions where the bugs are more likely to be initially triggered.

Fig. 6 shows what we record when visiting a node on the static call graph.

In our search strategy, when we visit a green node (except main function), we check it and look back at its parent node, the blue one, then we record the red line as suspicious positions.

If $f_j \rightarrow f_i$, we have $(EN_{f_j}, EX_{f_j}) \rightarrow (EN_{f_i}, EX_{f_i})$. Assume that f_j is the blue function and f_i is the green function. When we visit f_i and look back at its parent function f_j, we have the following three cases by considering $(EN_{f_j}, EX_{f_j}) \rightarrow (EN_{f_i}, EX_{f_i})$:

1) *Case 1:* $(0, ?) \rightarrow (1, 1)$. It shows that we do not have failure invariants at f_j ::: ENTER but have failure invariants at f_i ::: ENTER. It indicates that something changes

between f_j ::: ENTER and f_i ::: ENTER. Whether f_i is correct or not, we always have failure invariants at f_i ::: EXIT. So we only record the possible faults in the position between f_j ::: ENTER and f_i ::: ENTER. Thus, this case indicates that a fault may be found in the position between f_j ::: ENTER and function f_i ::: ENTER.

2) *Case 2:* $(0, ?) \rightarrow (1, 0)$. It shows that we have failure invariants at point f_i ::: ENTER but no failure invariants at f_i ::: EXIT. This situation rarely happens unless there is a block at f_i ::: ENTER because no failure invariants at f_i ::: ENTER are eliminated after f_i finishes. Thus, this case indicates a deadlock, which may happen at function f_i which should be recorded. Since there are no failure invariants at f_j ::: ENTER but at f_i ::: ENTER like the situation in case 1, we also record the position between f_j ::: ENTER and function f_i ::: ENTER.
3) *Case 3:* $(0, ?) \rightarrow (0,\ 1)$. It shows that there are no any failure invariants at the point f_i ::: EXIT after f_i finishes. So the behavior of f_i is different from that in unit testing. Thus, this case indicates that a fault may be in function f_i which should be recorded.

Besides the above cases, we have another case. In order to decrease false negatives, we add other suspicious function candidates to an extra candidate set. *Suspicious function candidates* are the ones that are called more than once in an integrated execution and have failure flag (1, 1). We regard them also as suspicious positions.

Suppose a function that is invoked more than once in an integrated execution has a bug. At first time when it is invoked, its failure flag is (0, 1). But in the next time it is invoked, the fault might already be existed, so its failure flag becomes (1, 1). It is unreasonable to omit such a suspicious function. However, these functions that are invoked more than once with no bugs can easily be infected by other functions, changing their failure flag to become (1, 1). So we should carefully treat them and set up a candidate set to include this type of functions.

Note ? takes 0 or 1. We develop a searching strategy, together with depth-first search, to search and locate the faulty functions. More details about visiting functions on the static call graph are as follows:

Start to search at the root function of the graph. If $EN_{f_{\text{root}}}$ is 0, then we always have (0,?) at f_{root}. Assume $f_{\text{root}} \rightarrow f_i$ and let $f_j = f_{\text{root}}$. Visit f_i, and look up at f_j. We see that $(EN_{f_j}, EX_{f_j}) \rightarrow (EN_{f_i}, EX_{f_i})$: $(0, ?) \rightarrow (?, ?)$. Consequently, we have total eight different permutations. For each permutation, we perform the following:

1) $(0, ?) \rightarrow (1, 0)$ or $(0, ?) \rightarrow (1, 1)$. Record the faulty position according to Case 1 or 2, then stop searching on the branch of f_i.
2) $(0, ?) \rightarrow (0,\ 1)$. Record the faulty position according to Case 3, then continue searching on the branch of f_i.
3) Others, $(0, 0) \rightarrow (0, 0)$ or $(0, 1) \rightarrow (0, 0)$. We record nothing on node f_i and continue to search on the branch of f_i.
4) Check on the static call graph and add all the functions with failure flag (1, 1) that are invoked more than once in the integrated testing into the candidate set.

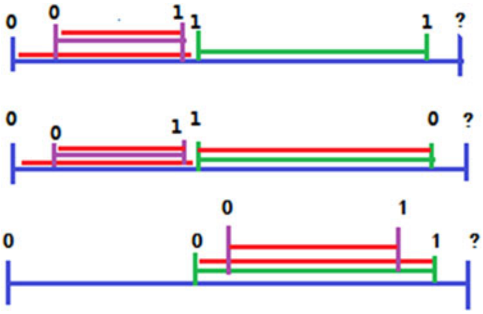


Fig. 7. Explanation of assumptions. The green, blue, and purple segments represent functions and the blue calls the green. The red lines represent the suspicious positions we should record.

Our approach to find suspicious positions is heuristic. After we find all the suspicious positions and the candidates of suspicious function, we give them the suspiciousness rates and rank them. The following assumptions are needed:

1) Different suspicious positions are executed at different orders. The earlier a suspicious position is executed, the more suspicious it will be.
2) The longer the distance of a suspicious position from the root on a function call graph, the more suspicious it will be.

Fig. 7 provides an explanation for the assumptions. We add another function, the purple one in the body of the blue function.

It is appropriate to infer that the deepest position overlapped by more red lines is more suspicious than the previous positions covered by less red lines. Besides, some suspicious positions are executed earlier than others and the faults in them can be transported and then infected in the later code block, so the bugs are more likely to be triggered in the earlier-executed skeptical code block. Thus, we regard the depth on the static call graph and the call order as two important factors in computing suspiciousness rates.

Based on the above analysis, we use the following formula for the computing:

$$S_{\text{rate}}(p) = \frac{\text{depth(p)}}{\text{order}(p)}$$

where p is a suspicious position or a candidate of suspicious function.

The following is an algorithm to compute the rates of suspicious positions. Its inputs are invariants, function call graph and function call orders.

In the algorithm, inv_Reduction() reduces redundant invariants for each function. DFS_Traverse() is a depth-first function to search for skeptical positions. $cal\_susp\_rate()$ calculates the suspiciousness rates of all skeptical positions.

Using the above proposed method, we can locate the suspicious positions and rank them. By further looking into the invariants of these positions, we can track the variables appearing in these invariants, and find out the causes.

Alogrithm: Detect Suspicious Positions.

```
Input: Invariants(inv), Function call graph(graph),
       function call order(order)
Output: Suspicious Position (position)
begin
    (EN, EX)=inv_Reduction(Inv, graph);
    For ($i \in graph$)
        position = DFS_Traverse(i, EN, EX);
        i++;
    rate=cal_susp_rate(position, order);
end
```

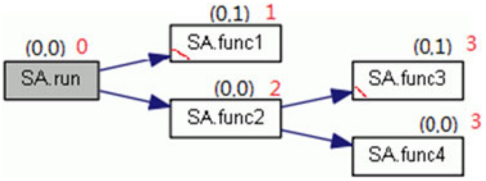


Fig. 8. Suspicious positions for the running example.

Now, we use the running example to compute the rates of suspiciousness functions.

In the static call graph for the running example, we mark each function f_j with (EN_{f_j}, EX_{f_j}), as shown in Fig. 8. For instance, mark SA.func1() with $(0, 1)$ meaning that SA.func1() ::: ENTER has no invariants while SA.func1() ::: EXIT has invariants. We also label the call order for each function besides the mark (EN_{f_j}, EX_{f_j}). For example, the order of SA.func1() is 1. Because SA.func3() and SA.func4() are conditionally called by SA.func2(), we label both functions with same order.

Let f_{root} be the root function, and $\text{depth}(\text{f}_\text{i})$ represents the shortest distance between f_i and root function f_{root}. In our running example

$$\begin{aligned} &f_{\text{root}} = \text{SA.run()},\\ &\text{depth}(\text{SA.run()}) = 0.0,\\ &\text{depth}(\text{SA.func1()}) = 1.0,\\ &\text{depth}(\text{SA.func3()}) = 2.0. \end{aligned}$$

Since

$$\text{order}(\text{SA.func1()}) = 1,\ \text{order}(\text{SA.func3()}) = 3$$

we have

$$\begin{aligned} S_{\text{rate}}(\text{SA.func1()}) &= \frac{\text{depth}(\text{SA.func1()})}{\text{order}(\text{SA.func1()})}\\ &= 1.00.\\ S_{\text{rate}}(\text{SA.func3()}) &= \frac{\text{depth}(\text{SA.func3()})}{\text{order}(\text{SA.func3()})}\\ &= 0.67. \end{aligned}$$

TABLE V
SUSPICIOUS POSITIONS FOR RUNNING EXAMPLE

Position name	invoked order	position	$Susp\_rate$	rank	fault spot
SA.func1()	1	1.00	1.00	1	-
SA.func3()	3	2.00	0.67	2	*

TABLE VI
INFORMATION OF THE SIX CONCURRENT PROGRAMS

Program	Lines of code	Bug description	Thread_N	TestCase_N
Producer-Consumer	80	bad composition	4	12
Dining Philosophers	125	Deadlock	5	18
reorder	44	Atomicity violation	4	7
account	66	Deadlock	5	10
log4j2	7951	Atomicity violation	2	851
log4j1	8570	Deadlock	2	982

Table V displays all suspicious functions for the running example.

From this table, we see that there are two suspicious positions for the running example. We label all the suspicious positions with red line in the static call graph. The suspiciousness rate of SA.func1() is 1.00, while that of SA.func3() is 0.67. So SA.func1() ranks the first. Actually, SA.func1() has a race condition bug.

G. Complexity Analysis

Suppose, we have n nodes on a function call graph, and each program point has m failure invariants on the average, where m is equal or bigger than 1. We use hashmap to store failure invariants for each program point, thus the asymptotic time complexity of reduction is $O(mn)$. Then, by checking the failure flags of each node, the asymptotic time complexity of our search strategy to find out suspicious positions as well as candidates of suspicious functions is about $O(n)$. So the total complexity is around $O(mn) + O(n) = O(mn)$ for our method. As for space complexity, the memory to store static call graph and failure invariants requires about $O(mn)$.

Every time a component is integrated with other components, we need to check the program points. So, if a system has n components, we need to check n times to get the failure invariants, which could be an overhead of our method.

V. EXPERIMENTS

We have total six concurrent programs in our experiment, as shown in Table VI.

Here in Table VI, Thread_N represents the number of threads for each program, and TestCase_N represents the number of test cases used to generate invariants. Test case can be randomly executed many times. The above concurrent programs have three types of concurrency bugs: deadlock, atomicity violation, and bad composition, which are invisible concurrency bugs that cannot be thrown out at runtime. Also, the components from each program have no errors in unit testing.

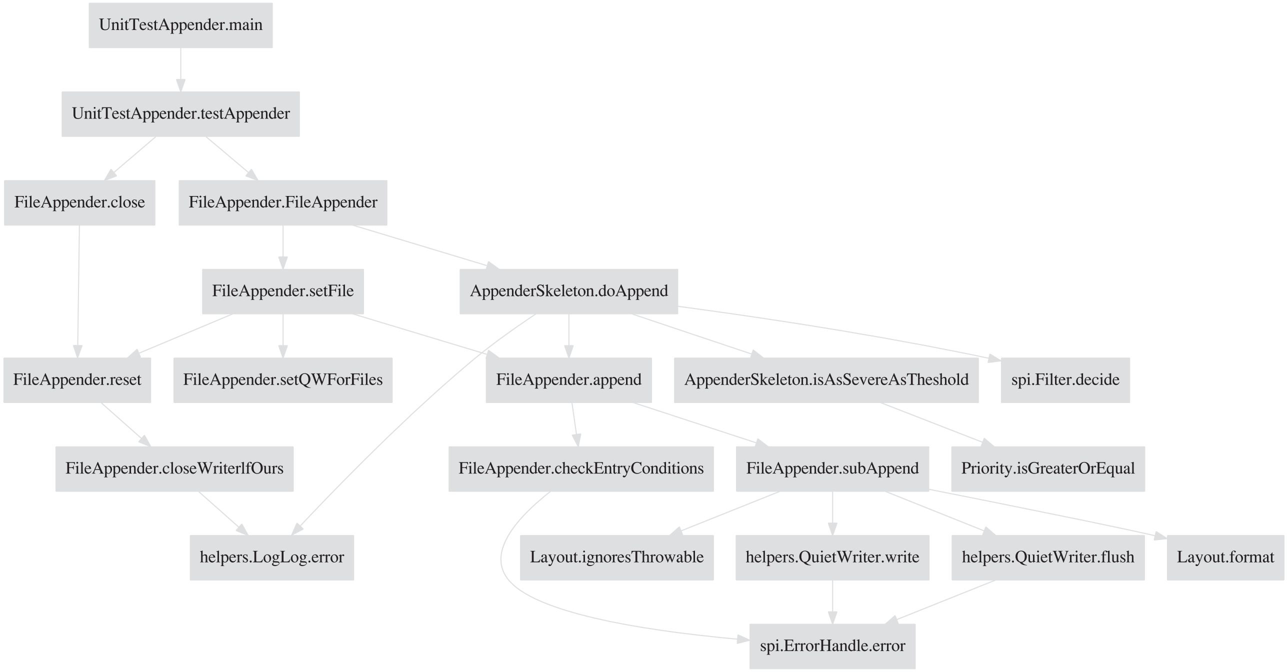

UnitTestAppender.main
UnitTestAppender.testAppender
FileAppender.close
FileAppender.FileAppender
FileAppender.setFile
AppenderSkeleton.doAppend
FileAppender.reset
FileAppender.setQWForFiles
FileAppender.append
AppenderSkeleton.isAsSevereAsTheshold
spi.Filter.decide
FileAppender.closeWriterIfOurs
FileAppender.checkEntryConditions
FileAppender.subAppend
Priority.isGreaterOrEqual
helpers.LogLog.error
Layout.ignoresThrowable
helpers.QuietWriter.write
helpers.QuietWriter.flush
Layout.format
spi.ErrorHandle.error

Fig. 9. Function call graph for log4j2.

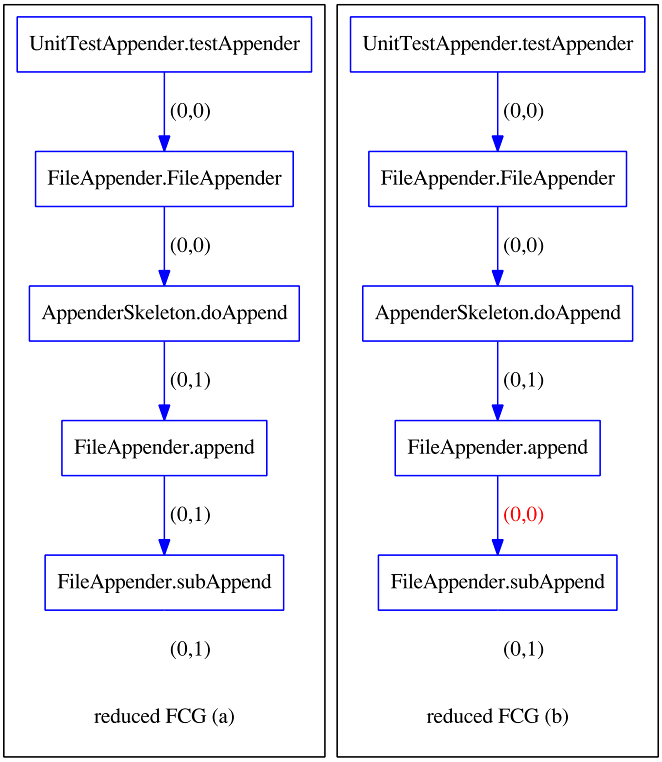


Fig. 10. Reduced FCG for log4j2.

TABLE VII
INVARIANTS FOR PROGRAM LOG4J2

Functions	Failure invariants <Before, After>
AppenderSkeleton.doAppend:::EXIT	$< this.closed == orig(this.closed), null >$
FileAppender.append:::EXIT	$< this.qw == orig(this.qw), null >$
	$< this.tp == orig(this.tp), null >$
	$< this.qw! = null, null >$
	$< this.tp! = null, null >$
FileAppender.subAppend:::EXIT	$< this.qw == orig(this.qw), null >$
	$< this.tp == orig(this.tp), null >$
	$< this.qw! = null, null >$
	$< this.tp! = null, null >$

A. *log4j2: Atomicity Violation Bug*

We select a program *log4j2* from software-artifact infrastructure repository (SIR).[5] This repository provides Java, C, C++, and C# programs for use in experimentation with testing and analysis techniques, and materials facilitating that use. SIR project is being supported by the National Science Foundation. Program *log4j2* with some real faults contains a Java program Apache *log4j* that is used to insert log statements into the code for debugging. It has 7951 lines of code and 117 Java class files. We need to check if FileAppender.append() is atomic.

A Java class FileAppender has been chosen as a component. First, we make a unit test for a FileAppender instance to call the public method ..doAppend() which is directly inherited from parent class AppenderSkeleton, and then call ..close(). From the unit testing, we obtain the reference invariants. Then, we do integration testing by adding a thread to call ..FileAppender.close() in a random way to disturb ..doAppend(). Fig. 9 shows the main function call graph in the unit testing.

By comparing reference invariants with online invariants, we filter out some of them for the unjustified failure invariants, from where we obtain total three program points with total nine failure invariants, as shown in Table VII.

Next, we apply reduction technique for invariants. By applying reduction (6), we remove 1 program point, i.e., append() ::: EXIT, since it has the same failure invariants as those of subappend() ::: EXIT in Table VII. Table VIII shows final five failure invariants after reduction.

[5] http://sir.unl.edu/portal/index.php

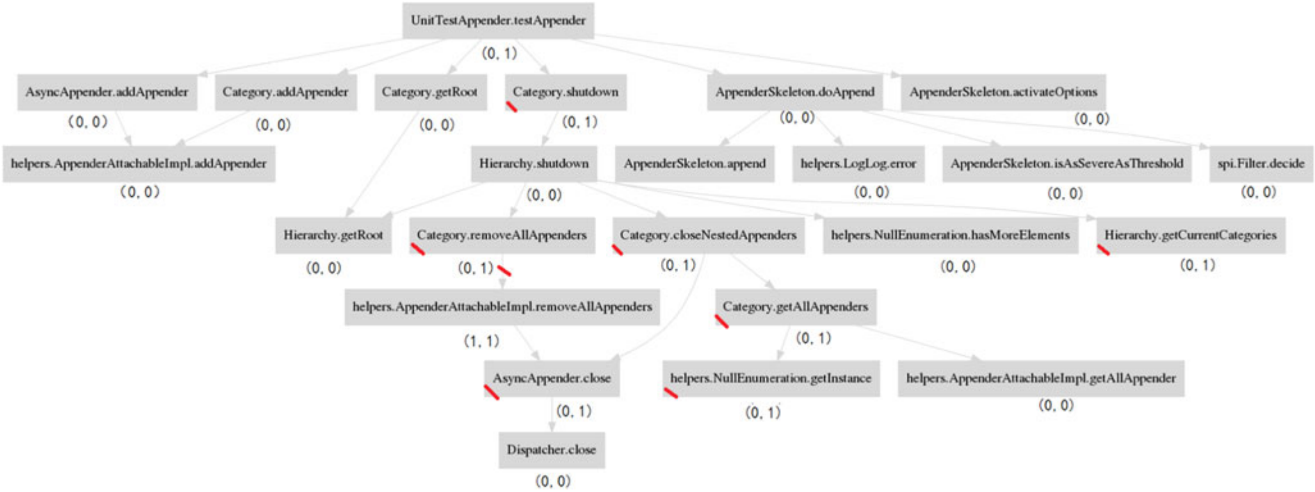

UnitTestAppender.testAppender
(0, 1)
AsyncAppender.addAppender
(0, 0)
Category.addAppender
(0, 0)
Category.getRoot
(0, 0)
Category.shutdown
(0, 1)
AppenderSkeleton.doAppend
(0, 0)
AppenderSkeleton.activateOptions
(0, 0)
helpers.AppenderAttachableImpl.addAppender
(0, 0)
Hierarchy.shutdown
(0, 0)
AppenderSkeleton.append
helpers.LogLog.error
(0, 0)
AppenderSkeleton.isAsSevereAsThreshold
(0, 0)
spi.Filter.decide
(0, 0)
Hierarchy.getRoot
(0, 0)
Category.removeAllAppenders
(0, 1)
Category.closeNestedAppenders
(0, 1)
helpers.NullEnumeration.hasMoreElements
(0, 0)
Hierarchy.getCurrentCategories
(0, 1)
helpers.AppenderAttachableImpl.removeAllAppenders
(1, 1)
Category.getAllAppenders
(0, 1)
AsyncAppender.close
(0, 1)
helpers.NullEnumeration.getInstance
(0, 1)
helpers.AppenderAttachableImpl.getAllAppender
(0, 0)
Dispatcher.close
(0, 0)

Fig. 11. Function call graph for log4j1.

TABLE VIII
INVARIANTS FOR PROGRAM LOG4J2 AFTER REDUCTION

Functions	Failure invariants <Before,After>
AppenderSkeleton.doAppend:::EXIT	$< this.closed == orig(this.closed), null >$
FileAppender.subAppend:::EXIT	$< this.qw == orig(this.qw), null >$
	$< this.tp == orig(this.tp), null >$
	$< this.qw! = null, null >$
	$< this.tp! = null, null >$

TABLE IX
SUSPICIOUS POSITIONS FOR LOG4J2

Posit_name	call order	position	$Susp\_rate$	rank	fault spot
FileAppender.subAppend	33.0	4.00	0.121	1	*
AppenderSkeleton.doAppend	28.0	2.00	0.071	2	-

Since there are no failure invariants in other functions in Fig. 9, we simply remove those unrelated functions from the static call graph. Fig. 10 shows the simplified static call graph for five primary functions which are relevant to the failure invariants. FCG(a) is the reduced function call graph before applying reduction (6), while FCG(b) is the reduced function call graph after applying reduction (6).

After computing, we obtain two suspicious positions, which are displayed in Table IX.

Now, we try to find the places that have bugs. Combining this table with the reduced static call graph, we see that a suspicious segment of path starting from $..\mathrm{doAppend}()$ to $..\mathrm{subappend}()$: $(0,1) \rightarrow (0,0) \rightarrow (0,\ 1)$ in FCG(a) after comparing with $(0,1) \rightarrow (0,1) \rightarrow (0,\ 1)$ in FCG(b). Note that both FCG (a) and FCG (b) display the failure flags of the functions, and FCG (a) becomes FCG (b) after reduction. Thus, there must be some faults in $..\mathrm{doAppend}()$ and $..\mathrm{subappend}()$, which do not affect other functions since there are no failure invariants in any other functions in Fig. 9.

Because the suspiciousness rate of $..\mathrm{subappend}()$ is higher than that of $..\mathrm{doAppend}()$, $..\mathrm{subappend}()$ ranks first. Thus, the bugs are very likely to be in function $..\mathrm{subappend}()$. This conclusion is consistent with the result from SIR project.

B. log4j1: Deadlock Problem

We select another program *log4j1* from SIR. It has 8570 lines of code and 111 Java class files. We design unit testing for some methods and then integrate them to find out concurrency bugs.

Choose component AsyncAppender and design a unit testing for it. First, we obtain reference invariants from the unit testing. Then, we design an integration testing and obtain online invariants. Fig. 11 shows the main function call graph in our unit testing. According to our replacement algorithm mentioned before, we have replaced Appender.close with AsyncAppender.close, since the latter is actually executed.

After comparing reference invariants with online invariants, we filter out some failure invariants for the target failure invariants. There are eight suspicious positions marked with red lines in Fig. 11. All the suspicious positions are shown in Table X.

In this case, the two most suspicious positions are Category.getAllAppenders and AsyncAppender.close, with suspiciousness rates 0.091 and 0.083, respectively. The real bug is between AsyncAppender.close and Dispatcher.close, very close to AsyncAppender.close. So our method is helpful to find out the bug.

C. Dining Philosophers: Deadlock Problem

This example is from the source:
http://my.oschina.net/sharkbobo/blog/270238.

This is a famous philosopher dining program. The scenario is as the following. There are five philosophers. Each philosopher picks up left chopstick, picks up right chopstick, eats and puts down all chopsticks. There is a deadlock bug in this program. The possibility of occurrence of deadlock is much lower than normal behavior. We first generate the right reference invariants from normal-running circumstance. When the program hits deadlock, we can capture the failure invariants by comparing invariants.

The function call graph is displayed in Fig. 12:

After computing, the suspicious positions are shown in Table XI. Note that Chopstick.pickdown() is a candidate function

TABLE X
SUSPICIOUS POSITIONS FOR LOG4J1

Posit_name	call order	position	$Susp\_rate$	rank	fault spot
Category.getAllAppenders	44.0	4.0	0.091	1	-
AsyncAppender.close	48.0	4.0	0.083	2	*
helpers.NullEnumeration.getInstance	61.0	5.0	0.082	3	-
Category.closeNestedAppenders	42.0	3.0	0.071	4	-
(Category.removeAllAppenders, helpers. AppenderAttachableImpl.removeAllAppenders)	63.5	3.5	0.055	5	-
Hierarchy.getCurrentCategories	60.0	3.0	0.050	6	-
Category.removeAllAppenders	63.0	3.0	0.048	7	-
Category.shutdown	38.0	1.0	0.026	8	-

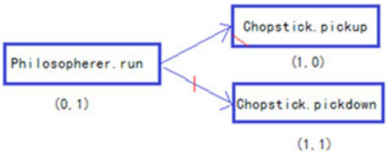


Fig. 12. Function call graph for Dining philosophers.

TABLE XI
SUSPICIOUS POSITIONS FOR DINING PHILOSOPHERS2

Posit_name	call order	position	$Susp\_rate$	rank	fault spot
Chopstick.pickup	1	1.00	1.00	1	*
Chopstick.pickdown	1	1.00	1.00	1	*
(Philosopherer.run, Chopstick.pickdown)	2	0.50	0.04	2	-

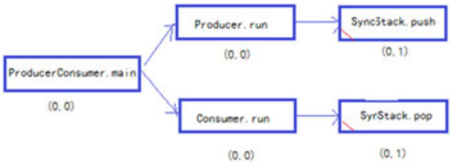


Fig. 13. Function call graph for Producer–Consumer.

because it has failure flag (1, 1) and is invoked more than once in an execution.

When all the philosophers pick up the right chopsticks, the program will encounter a deadlock. This bug can affect the invariants from the normal-running circumstance. In fact, the program encounters the deadlock problem at Chopstick.pickup() and Chopstick.pickdown(), where all the five philosophers pick up the chopsticks.

D. Producer–Consumer: Bad Composition

This example comes from the source:
http://down.51cto.com/data/670251.

It is a traditional Producer–Consumer program. The producer puts WoTou on a stack, and the consumer consumes WoTou from the stack. There is a fixed-size buffer named SyncStack for producer and consumer. Suppose the size of buffer is Max. There is a hidden fault in this program: the producer might sometimes put more than Max WoTou into the buffer because of a bug in SyncStack class.

First, we generate reference invariants from the program where there is one producer and one consumer; later, we catch the online invariants from multiproducer and multiconsumer. Then, we apply our technique to deal with all the invariants.

The function call graph is displayed in Fig. 13.

TABLE XII
SUSPICIOUS POSITIONS FOR PRODUCER–CONSUMER

Posit_name	call order	position	$Susp\_rate$	rank	fault spot
SyncStack.push	2	2.00	1.00	1	*
SyncStack.pop	2	2.00	1.00	1	-

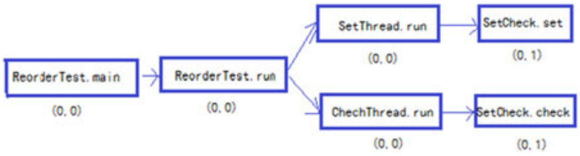


Fig. 14. Function call graph for reorder.

The suspicious positions are computed in Table XII.

Note that SyncStack.pop() and SyncStack.push(..) are synchronized functions. If there are more than one producers, then some producer may not recheck the stack when the stack full is notified, and continues to produce a WoTou into the stack. This bug is exactly in SyncStack.push(..).

We do find out the suspicious positions SyncStack.push() and SyncStack.pop(..), and both of them have the same suspiciousness rate 1.00. In the reality, SyncStack.push() do contains a bug. The following is the pseudocode containing a bug.

```
  push(WoTou wt){      // ought to be
    if(index == Max){ // while(index ==
Max)
  this.wait();
    }
     ...
  }
```

E. Reorder: Atomicity Violation Bug

We select a program *reorder* from SIR. The class SetCheck has two methods: set(), which is expected to be atomic; and check(), which checks the variable a and b. The pseudocode of class SetCheck is as the following:

```
  class SetCheck {
      set() {
     a = 1;
     b = -1;
      }
      check(){
        return ((a==0 && b==0)||(a==1 &&
        b==-1));
      }
  }
```

The function call graph is displayed in Fig. 14.

We attempt to examine the two methods of class SetCheck and design a situation where some threads separately call set() and check(). First, we obtain reference invariants from unit testing, then compare the reference invariants with the online invariants from integration testing. We obtain two functions with failure invariants, SetCheck.set and SetCheck.check. The suspicious positions are computed in Table XIII.

Although both of the suspicious positions have the same suspiciousness rate, it is more likely that set() is more skeptical

TABLE XIII
SUSPICIOUS POSITIONS FOR REORDER

Posit_name	call order	position	$Susp\_rate$	rank	fault spot
SetCheck.set	3	3.00	1.00	1	*
SetCheck.check	3	3.00	1.00	1	-

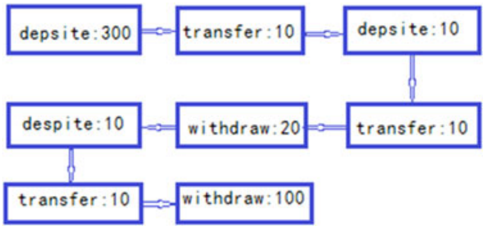


Fig. 15. Thing that each account does.

TABLE XIV
SUSPICIOUS POSITIONS FOR REORDER

Posit_name	call order	position	$Susp\_rate$	rank	fault spot
Account.transfer	4	2.00	0.50	1	*

than check() because check() is just an observed method. So we know that there might be something wrong in set() from our test. In fact set() has an atomicity violation.

F. Account: Deadlock Problem

We select a program *account* from SIR. The class Account has three methods: depsite(), withdraw(), and transfer().

We attempt to examine all three methods by designing a situation that operations are applied to an account with initially deposited money 100, as displayed in Fig. 15.

For unit testing, we have two accounts to do this work concurrently to get reference invariants. Then, we let more accounts joining together to do this work to get the online invariants. The suspicious position is computed in Table XIV. The call order of transfer() takes the average of call orders 2, 4, and 6. The depth of transfer() is the depth when it is first visited.

We obtain only one suspicious position Account.transfer. Indeed it causes deadlock problem since it locks the object when money 10 is transfer to this object.

G. Elevator System: No Bug Problem

This example (https://github.com/dylangsjackson/Elevator) is used to illustrate the shortcomings of our technique.

This program simulates an elevator system in a building with M elevators, N floors, and P riders. Each elevator runs on its own thread, so does each rider. Each rider is given a list of floors to visit in sequence, relying on elevator threads to take them to their desired floors.

Suppose, we have only one elevator (E) in a building and it works well. Then we add more elevators to this building. We find out that E works well in multi-elevator environment. This means that the program has no bugs.

Now, we apply our technique to this system. We first generate reference invariants of E by running a single elevator together with other components, and generate online invariants of E by running multiple elevators together with other components. We expect to see that the online invariants are the same as the reference invariants. However, we still find that there exit differences. This means that the program should have bugs based on our technique, but it does not!

This experiment illustrates that the interleaving of E in multi-elevator environment is not the same as that in single-elevator environment, even though we can generate "good enough" reference invariants of E. This means that our technique cannot cover such case.

VI. THREATS TO VALIDITY

External (projects and tools): Our results may not be used well for those programs where components have very complex interactions. We only test our technique to the existing bug types found in the literature. Our method can be applied to the invisible concurrency bugs such that there are no errors to be thrown out during the integrated testing and unit testing. In addition, this approach is used to find out the concurrency bugs triggered in integrated executions. The method has a limitation to detect all types of concurrency bugs since Daikon has its own limits. This method cannot work well if a concurrency bug does not reveal itself on failure invariants. A possible solution to this issue is to slice some functions by adding extra program points and draw a new static call graph for them and then do the test by monitoring these extra program points.

Internal (correctness of our implementation): We have used Daikon to generate invariants. We assume that it is correctly implemented. We have implemented the reduction algorithm by ourselves. To ensure the correctness of our implementation, we adopt a metamorphic testing (MT) strategy. MT is effective in alleviating the test oracle problems and has been applied to many problems. It tells whether the program is faulty or not by checking the violation of some properties each of which is known as a metamorphic relation (MR) of the given program instead of checking the single output of one given test input. Another threat is that the static call graph is not very consistent with the real complicated interleavings from undermined programs.

Construct (metrics and versions): To obtain reference invariants from unit testing, we hope that the test cases cover as many interactions as possible. Thus, we are unable to design a more systematic evaluation with many more programs in this paper. Another threat comes from *Daikon*. Since it implements a statistical learning method to obtain the easily comprehensible invariants, the quality of program invariants highly relies on the design of the test suite. Our approach mainly concentrates on whether program points have failure invariants or not. If the quality of the test suite is poor, we may have false failure invariants at the points of earlier invoked functions and incorrect fault positions may be located.

VII. RELATED WORK

Many fault localization tools exist, for example, [4], [11], [16], [17], and [25]. A typical one is the work by Abreu *et al.* [1]. They studied the utility of low cost, generic invariants (screeners) in their capacity of error detectors within a SFL [28] approach aimed to diagnose program defects in the operational phase. The screeners considered are simple bit-mask and range invariants that screen every load/store and function argument/return program point. But these work are all for sequential programs.

For concurrent programs, when multiple threads perform unsynchronized access to a shared memory location, there are some work on detecting data races, such as static techniques based on type systems [7], model checking [20], and general program analysis [21]. The main drawback of data race detectors is that some races like those used in barriers, flag synchronization, and producer–consumer queues are common parallel patterns that rely on deliberate but benign races [18]. Programmers are left to sort out benign and problematic cases on their own. In our method, we provide additional information (suspiciousness rate) to help pinpoint a fault's root cause.

Kusano *et al.* [14] proposed a new method for dynamically generating likely invariants from multithreaded programs. Their method takes a multithreaded C/C++ program as input and returns likely invariants as output. First, it instruments the code using a new LLVM-based front end to add monitoring capabilities for dynamic analysis. Then, it executes the program under the control of a systematic interleaving explorer. The generated traces are fed to a classifier that separates the passing traces from the failing traces. Finally, it feeds a subset of the traces to a customized Daikon invariant inference engine which returns the likely invariants as output. But it still has well-known interleaving explosion problem even though their use of selective exploration strategies. Moreover, some generated invariants may not exactly represent concurrency related program behaviors.

Lu *et al.* [18] proposed an invariant-based approach called AVIO to detect atomicity violations. The idea is based on the observation called access interleaving invariant, which is an indication of programmers' assumptions about the atomicity of certain code regions. By automatically extracting such invariants and detecting violations of these invariants at runtime, AVIO can detect a variety of atomicity violations. Comparing to this method, ours can detect many types of concurrency bugs.

Park *et al.* [23] proposed a pattern-based dynamic analysis technique for fault localization in concurrent programs that combines pattern identification with statistical rankings of suspiciousness of those patterns. They apply the technique to both atomicity and order violations. During testing, the technique detects access patterns from actual program executions, which either pass or fail. For each pattern, the technique uses the pass/fail statistics to compute a measure of suspiciousness that is used to rank all occurring patterns, in the spirit of Tarantula in the sequential case. However, it may also have large number of interleavings. Moreover, it cannot be used online since it needs an oracle for testing.

Koca *et al.* [13] proposed a SFL technique for localizing faulty code blocks and designed an implement tool called SCURF. First, they instrument the code to force the program to run in different combinations of thread interleavings, at the meantime, they collect the information for each block in each test. Then they employ SFL to correlate detected errors with the concurrently executing code blocks. However, this technique requires an oracle.

The Delftgrind error detector is implemented in C as a Valgrind [22] tool. Delftgrind is a three stage pipeline architecture. The first stage of the pipeline is the Negi frontend that parses the test case info and passes each one to Valgrind. Valgrind does all the needed transformations. Every interesting event, such as memory loads and stores, is then reported to the Negi backend that contains an event dispatcher, which then hands it down to the invariant checkers. The invariant checkers are in charge of selecting what events they want to monitor, as well as constructing, training, and enforcing invariants. Ninjin [27] is another error detector, whose architecture is very similar to the architecture used in Delftgrind. An experiment launcher is created reusing most of the code from Delftgrind, and the invariant screener code was extracted to a standalone program that parsed the logging output of the instrumented program. Unlike Delftgrind, Ninjin performs the instrumentation of the code at compile time instead of at runtime. The target program is instrumented and linked with the libninjin runtime library, which contains all the logging calls that produce the output that Ninjin will parse. Currently, only two invariant checkers have been developed, bitmask invariants and range invariants, which are most suited for embedded systems.

Campos *et al.* [3] added a new feature to GZoltar toolset to monitor variables. They proposed the use of two created algorithms to detect the system's so-called collar variables, and only monitor these variables, which could help reduce the performance loss without any significant reduction in error detection quality. By using fault screeners only on collar variables would enable a much lighter execution of the applications. However, current collar variable detection algorithms at the moment require multiple executions.

VIII. Conclusion

This paper has presented an invariant-based approach to detect concurrency bugs. By using reference invariants from unit testing, we can detect concurrency bugs caused by incorrect interleaving between components. Moreover, we can locate the places that cause the unexpected behaviors in the components.

The proposed method has several merits. It can check many types of invariants rather than bit-mask or range invariants, thus it has stronger ability to detect errors based on program invariants. It can detect many types of bugs such as deadlock, atomicity violation, and bad composition. It does not need an Oracle in unit testing since we only care about the properties of the component to be tested. Instead of monitoring variables, we monitor the program points. The bad things are that we do need experienced tester in unit testing, and false positive/negatives invariants may occur.

In the future, we will develop strategy to select as few effective invariants as possible to trace root causes. We also improve the searching strategy and the formula to compute the suspiciousness rate. We will study the relations among faults and failure invariants, and the dependence of failure invariants among the program points. We hope to generate invariants by trying other tools besides Daikon. Moreover, we need to solve the following issues to further improve our method:

1) Near complete set of invariants. It is hard for invariant detectors to discover a complete set of invariants since the problem of determining all invariants is undecidable. Hence, we need to develop a way to find near complete set of invariants.
2) Correct invariants in the computing. Not all reported invariants will be true for every possible execution of a program since the results of dynamic invariant detection depend on the particular test suite.

REFERENCES

[1] R. Abreu, A. González, P. Zoeteweij, and A. J. C. van Gemund, "Automatic software fault localization using generic program invariants," in *Proc. ACM Symp. Appl. Comput.*, 2008, pp. 712–717.

[2] R. Abreu, A. González, P. Zoeteweij, and A. J. C. van Gemund, "Using fault screeners for software error detection," *Eval. Novel Approaches Softw. Eng. Commun. Comput. Inf. Sci.*, vol. 69, pp. 60–74, 2010.

[3] J. Campos, A. Riboira, A. Perez, and R. Abreu, "GZoltar: An eclipse plug-in for testing and debugging," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Essen, Germany, Sep. 3–7, 2012, pp. 378–381.

[4] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 342–351.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001.

[6] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 237–252, Oct. 2003.

[7] C. Flanagan and S. N. Freund, "Type-based race detection for java," in *Proc. ACM SIGPLAN 2000 Conf. Program. Lang. Des. Implement.*, Jun. 2000, pp. 219–232.

[8] A. G. Sänchez, "Automatic error detection techniques based on dynamic invariants," Master's thesis, Dept. Softw. Technol., Delft Univ. Technol., Delft, The Netherlands, 2007.

[9] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proc. 24th Int. Conf. Softw. Eng.*, 2002, pp. 291–301.

[10] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty, "IODINE: A tool to automatically infer dynamic invariants for hardware designs," in *Proc. 42nd Des. Autom. Conf.*, San Diego, CA, USA, 2005, pp. 775–778.

[11] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. 20th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, New York, NY, USA, 2005, pp. 273–282.

[12] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[13] F. Koca, H. Sözer, and R. Abreu, "Spectrum-based fault localization for diagnosing concurrency faults," in *Proc. IFIP Int. Conf. Testing Softw. Syst.*, 2013, pp 239–254.

[14] M. Kusano, A. Chattopadhyay, and C. Wang, "Dynamic generation of likely invariants for multithreaded programs," *ICSE* (1), pp. 835–846, 2015.

[15] N. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," *IEEE Computer*, vol. 26, no. 27, pp. 18–41, Jul. 1993.

[16] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proc. 2005 ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, 2005, pp. 15–26.

[17] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 10, pp. 831–848, Oct. 2006.

[18] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting atomicity violations via access interleaving invariants," in *Proc. 12th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, San Jose, CA, USA, Oct. 21–25, 2006, pp. 37–48.

[19] T. Menzies, D. Owen, and J. Richardson, "The strangest thing about software," *IEEE Computer*, vol. 40, no. 1, pp. 54–60, Jan. 2007.

[20] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, Jun. 2007, pp. 446–455.

[21] M. Naik and A. Aiken, "Conditional must not aliasing for static race detection," in *Proc. 34th Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 2007, pp. 327–338.

[22] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, Jun. 2007, pp. 89–100.

[23] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, Cape Town, South Africa, May 2–8, 2010, pp. 245–254.

[24] M. Prvulovic and J. Torrellas, "ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes," in *Proc. 30th Annu. Int. Symp. Comput. Archit.*, Jun. 2003, pp. 110–121.

[25] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proc. IEEE Int. Conf. Softw. Eng.*, Montreal, QC, Canada, Oct. 2003, pp. 30–39.

[26] S. Savage *et al.*, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[27] J. Santos, "Automatic error detection using program invariants for fault localization," Master's thesis, Mestrado Integrado em Engenharia Informática e Computação, Univ. Porto, Porto, Portugal, 2012.

[28] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

Authors' photographs and biographies not available at the time of publication.