SPDebugger: A Fine-Grained Deterministic Debugger for Concurrency Code

Ziyi LIN†(a), Yilei ZHOU(b), Hao ZHONG†(c), Yuting CHEN†(d), Haibo YU†(e), Nonmembers, and Jianjun ZHAO†††(f), Member

SUMMARY When debugging bugs, programmers often prepare test cases to reproduce buggy behaviours. However, for concurrent programs, test cases alone are typically insufficient to reproduce buggy behaviours, due to the nondeterminism of multi-threaded executions. In literature, various approaches have been proposed to reproduce buggy behaviours for concurrency bugs deterministically, but to the best of our knowledge, they are still limited. In particular, we have recognized three debugging scenarios from programming practice, but existing approaches can handle only one of the scenarios. In this paper, we propose a novel approach, called SPDebugger, that provides finer-grained thread controlling over test cases, programs under test, and even third party library code, to reproduce the pre-designed thread execution schedule. The evaluation shows that SPDebugger handles more debugging scenarios than the state-of-the-art tool, called IMUnit, with similar human effort.

1. Introduction

Concurrent programming has been widely used in practical Java projects. According to a recent survey [1], 75% of the surveyed 2,227 real-world projects use concurrent mechanisms directly or indirectly. Along with the benefits such as improving program performance and providing better user experience, concurrent programming also brings various bugs [2]. In practice, programmers often prepare test cases to reproduce buggy behaviours, so that they can have a close understanding on the buggy code. However, it is more challenging to reproduce buggy behaviours for concurrent programs, since they usually execute nondeterministically and anti-intuitively [3]–[5].

We notice that programmers can take several typical naive approaches to debug the concurrent programs. The first is stress testing, where a test case that can trigger a concurrency bug by executing the buggy program for many times. Such testing is time consuming and ineffective. The second is to instrument sleeping statements which can roughly control threads interleavings by postponing thread executions. However, this approach cannot guarantee expected execution sequences. The third is to add break points at suspicious locations, and manually control thread interleavings to reproduce buggy behaviours. Despite of that it is more interactive, it is still tedious and cannot handle complicated bugs, since programmers can become confused in thread switches.

In literature, various approaches [6]–[10] have been proposed to enforce thread interleavings for unit testing. A typical such approach is IMUnit [7]. It advocates an event-based and partial-order methodology to define and control thread interleavings. More specifically, programmers declare events among testing program, and define a schedule which is expecting to execute at the beginning of the test. Figure 1 shows an example of IMUnit. In this test case, two elements are added into a blocking queue whose capability is one in one thread, and two elements are taken out from the queue in another thread. 4 events are declared (lines 10, 11, 15 and 18), and one schedule is defined (line 2). The testing is deterministically executed according to the schedule, resulting the assertions (line 17 and 20) passed constantly on each run of the test.

However, manifesting thread interleavings within test cases in a line-based style is insufficient to reproduce buggy behaviours in many cases. As we all know, JVM executes compiled instructions, but not source code. One line of source code can be compiled into several instructions. In our three recognized debugging scenarios (see Sect. 2 for details), IMUnit and the other approaches can handle only the line-based scenario which makes two assumptions: (1) the interleaving of threads only happen between lines of source code and (2) no need to check iteration state. But such assumptions are not satisfied in both theoretical and practical situations. For example, a bug collected in our previous work JaConTeBe [11], JDK4779253 (shown in Fig. 1 b), is triggered when the two underlined condition checks (statement 1 and 3) are preempted in between by a null reference assignment (statement 2). IMUnit cannot control threads to interleave inside a line of code. One alternate way to enforce such controlling is to break the original two condition checks connected by “&&” into two “if” statements, and instrument an event declaration between them. However, it is
discouraged to modify original programs for 3 reasons: (1) the modification may introduce new bugs; (2) it has to prove the bug reproduced from the altered program is the identical one as the original one; and (3) it is infeasible to modify programs from third party libraries. In addition, IMUnit does not support controlling program under test.

In this paper, we propose a novel approach, called SPDebugger, that provides finer-grained thread controlling into the scope of program under test and even third party libraries. To schedule thread execution sequences, SPDebugger uses a global clock that is advanced by event firings. We design a DSL to describe its working mechanism. On one hand, the DSL is implemented in a XML schema, therefore programmers can write DSL specifications in a separate XML file without modifying the program under test. On the other hand, the thread controlling work flow is implemented on a testing framework based on JDI (Java Debugging Interface). SPDebugger takes test cases and XML specification as its input to deterministically reproduce the expected execution schedule. This paper makes three contributions:

- Categorizing debugging scenarios. We identify three types of concurrency debugging scenarios, and two of them are not well supported by existing approaches.
- Designing a DSL for threads controlling. We design a DSL which combines a global clock and events for more concise and precise threads controlling.
- Implementing and evaluating. We have implemented a debugging tool, called SPDebugger, which combines a user-defined XML specification and JDI-based framework together to manipulate thread interleavings without altering source code. We have evaluated the tool on real-world methods. The results show that comparing with the state-of-the-art tool, IMUnit, SPDebugger can handle more debugging scenarios without increasing user efforts.

The rest of this paper is organized as follows: Section 2 introduces the recognized debugging scenarios. Section 3 introduces the methodology of our approach. Section 4 introduces the implementation details, and Sect. 5 evaluates our approach. Section 6 discusses the related works and Sect. 7 concludes.

2. Debugging Scenarios

Based on our programming experience, we identify three types of concurrency debugging scenarios as follows:

2.1 Line-Based Debugging

In the line-based debugging, thread preemptions happen between lines, but not within a line. In this scenario, programmers can determine thread executions and corresponding code by instrumenting controlling statements (e.g., sleep statements, IMUnit events declarations) between lines. Threads interleave based on instructions. As one line of source code can be compiled into one or more instructions, line-based interleaving can be considered as a special case for all possible interleavings.

A typical example of line-based debugging is shown in Fig. 1 a. In particular, the two threads, `main` (the default thread) and `addThread`, are declared separately to execute different tasks. Programmers can add events before or after lines to enforce thread interleavings.

2.2 In-Line Debugging

Although threads interleave on instructions but not lines, in practise thread preemptions often happen between lines. But when a line of code is composed by complicated expressions, it is very likely for threads to preempt in line. In this scenario, programmers have to consider how to implement in-line preemption when they try to reproduce such bugs.

For example, in Fig. 1 b, a line of code is composed by two condition expressions that are connected with an and operator. Other threads can preempt a thread between the two condition expressions. As another example, in Fig. 2 a, the line in `foo` method is composed by method
invocation chains. It is possible that statement 2 is executed between statement 1 and 3, leading to an unexpected NullPointerException.

2.3 Loop Debugging

Loop debugging involves with loop statements. A loop statement can initiate many threads to execute the same piece of code concurrently. In addition, a thread can have loop statements. In both cases, the buggy behaviours can happen only in specific iterations. As a result, programmers need to consider iteration states during debugging.

For example, code in Fig. 2 b is simplified from a server code that deals with requests from clients, and a loop statement creates threads. The programmer may want to schedule only one of the threads, instead of all of them. As another example, in Fig. 2 c, a loop statement is executed inside a thread, but the programmer may want to schedule a preemption in the loop only when a specific condition is satisfied. Existing IDEs such as Eclipse\footnote{https://www.eclipse.org/} and XCode\footnote{https://developer.apple.com/xcode/} allow adding conditional breakpoints, but such breakpoints do not support debugging concurrency bugs.

3. Approach

To the best of our knowledge, existing approaches cannot effectively deal with in-line and loop debugging scenarios for concurrency code. In this paper, we propose a novel approach, called SPDebugger, that supports all the three debugging scenarios. In particular, SPDebugger deterministically executes the concurrent program by controlling a few key preemptions which are predefined in a separate schedule file. In the schedule, such preemptions are specified by **Sequence Point (SP)** as the primitive controlling unit. This section introduces the key components of SP.

3.1 Sequence Point

**Definition 1** (Sequence Point): \( sp = (t, l, s, c) \), where

- \( t \): the thread that binds to this SP.
- \( l \): the location that places this SP in the code of the original program. It contains a file name (f), a line number (ln) and a column number (col).
- \( s \): the ID of the execution sequence, indicating the global execution sequence of this SP. It is a unique natural number starting from 1.
- \( c \): the constraints for SP.

SP controls the preemption(\( s \)) of a specified thread(\( t \)) at a certain location(\( l \)) of a program with the default or other constraints(\( c \)). An SP’s constraints are checked when it is reached in an execution of the specified thread, and the thread is allowed to continue when the check passes; otherwise, the thread is suspended until the check passes.

A piece of code can be executed by more than one thread, but an SP placed in such code may not be required to be hit and checked by each thread. To prevent unnecessary hits and checks, SP is bound to the target thread by specifying \( t \). It is by default set to the current executing thread when there is no ambiguity, otherwise, it can be set by a thread name.

\( l \) indicates the location to place SP in the source code of a program. With a qualified file name, a line number and a column number, an SP can be placed anywhere in the program, which empowers SPDebugger the ability to manipulate thread interleavings within a line.

\( s \) the unique sequence number indicating the global execution sequence of SP, and its ID number as well. Therefore, programmers can define SP and plan thread schedule at the same time, and it is intuitive for programmers.

\( c \) the constraints that are checked to determine what to do next. We define three types of constraints such as binding, activating and scheduling. The binding constraint (Constraint 1) checks whether the current executing thread is the thread that is bound to the SP. If it is, the activating constraint (Constraint 2) is checked to determine whether the SP should be activated or ignored. If it should be ignored, the thread continues as the SP does not exist. If it is activated, the scheduling constraint (Constraint 3) is checked as the last step to determine should the thread suspend or continue. It is feasible to combine the constraints for com-
plicated cases.

**Constraint 1** (The Binding Constraint): this refers the current SP and thread is the given thread ID.

\[ this.t == \text{thread} \] (1)

The simple iteration constraint is a typical activating constraint. When a programmer needs to manifest thread interleaving inside a loop, he/she may want to enforce the interleaving only limited times under certain conditions, instead of controlling the interleaving in every iteration. The simple iteration constraint can be used to deal with such situation by setting the intended iteration state.

**Constraint 2** (The Iteration Constraint): guard is a preset value specifies which iteration of a loop to bind. iteration_counter is a counter maintained to automatically count iterations.

\[ \text{guard} == \text{iteration}\_\text{counter} \] (2)

**Constraint 3** (The Scheduling Constraint): this refers the current SP, and global_sequence is a global counter that maintains the expected sequence ID of the schedule.

\[ this.s == \text{global}\_\text{sequence} \] (3)

It requires at least four SPs to enforce a preemption. Two of them define inserting boundaries, and the other two SPs define the minimum range of what code to execute as a preemption. For example, in JDK4779253, as shown in Fig. 3, SP1 and SP4 set the inserting location, and SP2 and SP3 give the range of code snippets to insert. When SP2 is reached before SP1, thread t2 is blocked until SP1 is executed, therefore, statement 1 is guaranteed to perform ahead of statement 2. Similarly, only when SP3 is executed, thread t1 can pass SP4 to execute statement 3, therefore, the NullPointerException is thrown and the buggy behaviour is reproduced.

### 3.2 DSL Specifications

We design a DSL to describe the SP-based scheduling. The DSL has the following elements:

**Action:**

- \( \text{checkConstraints(thread, sp)} \) checks the binding constraints of SP, and if they are satisfied, it checks the scheduling constraints of SP.
- \( \text{increaseCounter} \) increases the value of the global sequence counter by 1.
- \( \text{runThread(thread)} \) continues to run the specified thread.

- \( \text{suspendThread(thread)} \) suspends the specified thread from executing.
- \( \text{resumeThread(thread)} \) resumes the specified thread to get ready for checking constraint.

**Event:**

- \( \text{unbound} \): It denotes that the binding constraints of SP are not satisfied.
- \( \text{satisfied} \): It denotes that the scheduling constraints of SP are satisfied.
- \( \text{unsatisfied} \): It denotes that the scheduling constraints of SP are unsatisfied.
- \( \text{hitSP} \): It denotes that an SP is reached.
- \( \text{counterChanged} \): It denotes that the value of global sequence counter is changed.

**State::Uncontrolled**

- Available action: none
- State transition: Uncontrolled→Ready

**State::Ready**

- Available actions: checkConstraints, runThread, suspendThread
- State transitions: Ready→Running, Ready→Suspended, Ready→Uncontrolled

**State::Running**

- Available action: increaseCounter
- State transition: Running→Ready

**State::Suspended**

- Available action: resumeThread
- State transition: Suspended→Ready

Each bound thread is controlled according to the work flow shown in Fig. 4. The rectangles are states, contents below the bar in a state are available actions, arrows between states are state transitions, and the words beside arrows are the events fired for transition. The stars marking on actions indicate they are performed immediately when the thread enters the state. There are 3 principles for the work flow:

- The thread enters Ready to check all constraints whenever an SP is hit.
- If the scheduling constraints are satisfied, the thread
can continue executing, otherwise blocks.

- Action `resumeThread` can only be performed when the thread in `Suspended` is passively notified by the updating of global sequence counter (GSC).

We next explain the work flow of thread controlling with the example in Fig. 3. Suppose that in the first line, SP1 and SP4 bound to thread `t1`; and in the second line, SP2 and SP3 bound to thread `t2`. The GSC is 1.

`t1` and `t2` are both `Uncontrolled` when they start. When `t2` hits SP2, it fires the `hitSP` event and transits to `Ready`. The `checkConstraints` action is executed immediately to check the binding constraints and the scheduling constraint. Here, the scheduling constraint is unsatisfied, since SP2.s (2) does not equal to the current GSC (1), so `t2` performs the `suspendThread` action and fires unsatisfied event to transit to `Suspended`.

`t2` then executes and reaches SP1. Similarly, it transits to `Ready` state, and passes binding and scheduling constraints checking. So it performs `runThread` action, firing the `satisfied` event to transit to `Running`. `t2` performs the `increaseCounter` action to update the GSC to 2 as soon as the state changed, and then continues to execute the rest part of its program.

On the other hand, when GSC is updated, the `counter-Changed` event is fired (by system) to notify all suspended threads to perform `resumeThread` to get to `Ready`. Therefore, `t2` transits to `Ready`. The constraints in SP2.c are checked again and passed this time, so `t2` executes `runThread` action and transits to `Running`. As a result, the GSC is updated to 3.

At this moment, both `t1` and `t2` are `Running` and eligible to execute. The choice is made by the JVM scheduler. Suppose that `t1` is chosen. It cannot pass SP4 when the GSC is 3, and it transits to `Suspended`. When `t2` is chosen, it continues the rest of the second line and hits SP3 eventually. When the GSC is updated to 4, `t1` is allowed to execute.

Following the work flow described above, threads are executed strictly by the predefined schedule: the assignment statement between SP2 and SP3 preempts during the execution of thread `t1` at the specified location, which is between SP1 and SP4.

### 3.3 XML Schema

SPDebugger takes XML file as an input for DSL specifications for two reasons: (1) It is tedious and effort costing for programmers to learn and write DSL specifications. But XML is familiar to everyone. (2) Separate XML file can eliminate coupling between schedule and program, so programmers does not need to modify the program for setting SPs and schedule. In this section, we represent the design of the XML schedule file, which is known as XML schema.

The root element of the XML file is named as `schedule`, and it has two main types of elements, such as thread-declaration elements and SP-scheduling elements.

To handle the loop situation, we need to identify the thread created inside loops. We say that a thread is intended to control as the target thread. Element `Thread` is designed to mark the target thread, as shown in Fig. 5a. A `Thread` element has an attribute, `id`, and two sub-elements:

- `id`, the identification of the thread. It can be considered as the primary key of a `Thread` entity.
- `location`, the location where a thread is declared in the program.

`consExpr`, the constraint expression for binding a target thread. The location alone cannot indicate a specified thread in loop. For example, when a location points to line 2 of Fig. 2b, there are ten threads created at the same location. A binding constraint is required to distinguish the target thread from the others. The programmer can specify a simple binding constraint such as “i==5” to tell the testing framework which thread to take as target thread.

It is unnecessary to declare `Thread` elements for all threads. When threads are created with names in the program, the testing framework can easily track them with the names. In line-based debugging scenarios, threads may be not explicitly named, but it is not difficult to automatically reason the corresponding thread at runtime as they are created separately, such as the example shown in Fig. 1a.

According to its definition (Definition 1), we design an SP element shown as in Fig. 5b. In particular, an SP element has two attributes such as `id` (denoting the `s` in Definition 1) and `thread` (denoting `t`), and two sub-elements such as `location` and `constraint` (denoting `l` and `c`, respectively).

- `thread` indicates that a target thread binds to this SP. It is empty if the binding has no ambiguities; it is the thread name if it has been declared in code; otherwise it is the `id` of the previously declared `thread` element.
- `location` denotes where to place an SP. It includes the file name, the line number and the column number, and an SP is instrumented right before a location. A location can have more than one SP, and an SP with smaller sequence `id` comes before the bigger ones.
• constraint has an attribute that specifies the constraint type (i.e., activating or scheduling), and an expr element for constraint expression. By default, each SP contains a binding constraint (i.e., Constraint 1) and a scheduling constraint (i.e., Constraint 3).

For example, in Fig. 2b, programmers can bind the fifth created thread to an SP right before invoking the bar() method at line 12 with the following code:

```
<schedule>
  <thread id="t-1"/>
  <location file="someFile" line="2"/>
  <consExpri==4</consExpr>
</thread>
<sp id="1" thread="t-1">
  <location file="someFile" line="12"/>
</sp>
...
</schedule>
```

The following XML code defines the reproducing schedule for the bug in Fig. 3:

```
<schedule>
  <sp id="1" thread="t1">
    <location file="someFile" line="10" column="15"/>
  </sp>
  <sp id="2" thread="t2">
    <location file="someFile" line="100" column="0"/>
  </sp>
  <sp id="3" thread="t2">
    <location file="someFile" line="100" column="17"/>
  </sp>
  <sp id="4" thread="t2">
    <location file="someFile" line="10" column="15"/>
  </sp>
</schedule>
```

The file has no thread element, since it has no ambiguities for the thread declaration. The file has no constraint elements either, since the default constraints are added by the testing framework automatically.

4. Implementation

We have implemented the SPDebugger with the supports of an open source instrumentation tool called ASM and a Java debugging framework called JDI.

Java platform debugger architecture (JPDA)

is an infrastructure for Java debugging applications. JDI is the top tier of JPDA. It provides Java level interfaces to obtain program execution states and to control executions of programs, (e.g., suspending and resuming threads). It has been widely adopted by many popular IDEs (e.g., Eclipse and NetBeans) for debugging.

Figure 6 shows the overview of our SPDebugger framework. It takes a buggy program and a programmer defined schedule XML file as its inputs, and produces expected buggy running of a program. SPDebugger maintains two JVMs: the debuggee JVM that runs the program under test and triggers controlling requests, and the debugger JVM responds the requests according to the specified DSL workflow and schedules the thread interleavings. Here, we implement the request-respond mechanism of SPDebugger based on JDI.

When the debuggee JVM starts, it reads the user-defined XML file and instruments the relevant binding and controlling code into the proper locations of the binary code. Notice the locations declared in the XML are locations in source code, SPDebugger maps them to binary code locations before instrumentation.

The thread elements are parsed to binding code, and SP elements are parsed to controlling code. They are instrumented at the specified locations by ASM. The instrumented code works in two steps. It firstly checks corresponding constraints: binding code for binding constraints; controlling code for activating constraints and scheduling constraints. It then performs corresponding actions when the check passes. The controlling is traditionally achieved by adding synchronizing statements (e.g., sleeping, waiting, or latches). However, it is error-prone, and can cause deadlocks. Therefore, instead of such direct controlling, the debuggee JVM fires requests to the debugger JVM, asking the debugger to leverage JDI to control threads.

The debugger JVM monitors executions of the program under test and maintains their states. When it receives a controlling request, it determines to suspend or run the thread according to program’s current state (e.g., thread states and GSC values) and the specifications defined in Sect. 3.2.

5. Evaluation

We have conducted evaluations on SPDebugger to answer two research questions:

**RQ1.** To what degree can SPDebugger reproduce pre-designed threading schedules deterministically?

**RQ2.** How much effort can cost with the support of SPDebugger?

We compare SPDebugger with IMUnit only for 3 reasons: (1) IMUnit has been published in one of the most reputable conference (FSE) as a tool of state-of-the-art for Java; (2) IMUnit has provided adequate resources for comparison, including both source code and testing programs; (3) IMUnit is very representable, many other more recent works improve certain aspects on IMUnit, but they do not break through IMUnit’s methodology and not handle more scenarios. Therefore, it is not necessary to compare with the similar works repeatedly. Our results show that SPDebugger supports more debugging scenarios with similar effort than the state-of-the-art tool.
5.1 Setup

We implement SPDebugger as an Eclipse plugin. The evaluations are conducted on a PC with Intel Core i5 1.6GHZ CPU, 4G RAM, Windows 8.1 and JDK1.6.0.45.

Subjects. We employ 40 test cases in Table 1 as our benchmarks. Column “Covered Scenario” lists the covered debugging scenarios as discussed in Sect. 2; Column “#Tests” lists the number of test cases; Column “Project” lists source projects; and Column “Size(LOC)” shows the scale of the project under evaluation.

All the selected projects are popular in both industrial and concurrency research areas. JBoss Cache and Apache Collections have been selected in the evaluations of IMUnit [7] and we reuse them for comparison in the line-based debugging scenario. The input of IMUnit is a test case that is instrumented with a controlling schedule annotation and events annotations, as shown is Fig. 1 a. To prepare the test case for SPDebugger, we remove these annotations, but keep the rest of the test case. We carefully design the schedules, so that both the tools are expected to reproduce the same thread interleavings. The sizes of these two subjects are quite small because only codes relevant to the tests are extracted from original projects and run with the tests.

The other projects are selected from our previous benchmark JaConTeBe [11] for another two scenarios. They are imported in the form of Java archive files to evaluate the effectiveness of SPDebugger on binary code. We do not extract the testing relevant code but employ the whole project to evaluate whether SPDebugger can handle big scale real-world applications. The evaluated methods are either from bugs in JaConTeBe, such as JDK4779253 for in-line scenario and Groovy4292 for loop scenario, or similar to the example programs in Fig. 1 b, Fig. 2 a, Fig. 2 b and Fig. 2 c. For example, JDK4779253 is a typical case of in-line scenario as discussed previously. Groovy4292 from JaConTeBe is a typical case of loop debugging scenario. Its reproduction is very complex, in briefly, it requires 4 thread context switches in 3 iterations to reproduce the bug.1 In order to reproduce this bug, developers have to either set break points and control threads interleavings manually, which is tedious and error-prone; or run the test for hundreds of times, hoping one of them could trigger the bug by accident. Test cases and SP XML schedule files are designed for evaluating subjects, one schedule for one test. We cannot prepare corresponding IMUnit schedule and events annotations for the test cases of these methods, because we need to insert events annotations into the binary code under test for the same scheduling, but it is not supported by IMUnit.

Table 1

| Covered Scenario | #Tests | Project       | Size(LOC) |
|------------------|--------|---------------|-----------|
| Line-based       | 8      | JBoss Cache   | 628K      |
|                  | 12     | Apache Coll.  | 738K      |
| In-Line          | 5      | JDK           | 1307K     |
|                  | 5      | Derby         | 568K      |
| Loops            | 5      | Tomcat        | 219K      |
|                  | 5      | Groovy        | 120K      |
| Total            | 40     |               | 2013K     |

Table 2

| Project          | #Tests | #Passed |
|------------------|--------|---------|
| JBoss Cache      | 8      | 8       |
| Apache Collections | 12    | 12      |
| JDK              | 5      | -       |
| Derby            | 5      | -       |
| Tomcat           | 5      | -       |
| Groovy           | 5      | -       |
| Total            | 40     | 50%     | 100%     |

5.2 RQ1. Scheduling Ability

This section answers the RQ1 by evaluating SPDebugger and IMUnit with various test cases. Metrics. The design of test cases follows IMUnit’s style. Tasks are concurrently executed in a test. The assertion(s) at the end of the test is (are) true only when the threads exactly run along with predefined specific schedules. As shown Fig. 1 a, the assertions at line 16 and 20 are true only if the threads run according to the schedule defined at line 2.

It is possible that the threads can execute the schedule “naturally”, therefore, we run each line-based test case with each tool for 50 times. If the test can always pass the assertion, we know the thread schedules are deterministically manifested. Otherwise, if the test fails to pass the assertions once, the schedule is not deterministically reproduced. For tests of in-line scenario, SPs are set inside a line of source code to force thread preempting in line. For tests of loop scenario, SPs’ iteration constraints are additional set to match certain iteration states. We run each of these tests only with SPDebugger for 50 times, as IMUnit does not support such scenarios. Only when the specified schedule is reproduced for all 50 times, the test is considered as passed.

Result. Table 2 shows the results of reproducing expected schedules. Column “Project” lists the projects of test cases. Column “#Tests” lists the number of test cases. Column “#Passed” lists the number of passed test cases. Here, “-” denotes that the corresponding tool does not support the debugging and there is no result.

Both tools passes all the test cases of the top two rows. However, for the other test cases, IMUnit does not pass for two reasons: 1) They require to step into the binary code to enforce thread interleavings, but IMUnit cannot instrument binary code. 2) IMUnit does not support finer controls than lines. In total, IMUnit passes half of the test cases. SPDebugger successfully reproduces all expected thread schedules, and passed all test cases.

For RQ1, our results show that SPDebugger is as effective as IMUnit for the line-based debugging scenario.

1Details of this bug can be found at http://stap.sjtu.edu.cn/index.php?title=Groovy4292.
Furthermore, SPDebugger handles the other two debugging scenarios, and is able to handle large scale binary code from real-world applications.

5.3 RQ2. Scheduling Effort

This section answers RQ2 by comparing efforts spent on SPDebugger and IMUnit to achieve the same threading schedules.

**Metrics.** For RQ2, we measure the effort of IMUnit with the number of declared events, and measure the effort of SPDebugger with the number of declared SPs. We consider the effort of declaring an event equals to the effort of declaring an SP, since they both require locating the proper place to write declaration statements.

**Result.** Table 3 shows results for RQ2. Column “Project” lists project names. Column “LOC” lists lines of code of test cases. Column “Efforts” lists number of declared events or SPs. Sub-column “IMUnit” lists number of declared events, and sub-column “SPDebugger” lists number of declared SPs.

Only test cases in JBoss Cache and Apache Collections are compared in RQ2, since both tools can work on these test cases. For all the 20 test cases, each one has a schedule defined by IMUnit and a schedule defined by SPDebugger. Both schedules define the same threads execution interleavings. On average, IMUnit requires 4 (80/20) events to define a schedule, while SPDebugger requires 4.35 (87/20) SPs. The result indicates it needs slightly more effort to define a schedule with SPDebugger.

SPDebugger needs more effort, since IMUnit provides a more meaningful syntax which can replace the combination of several primitive tokens. For example, an IMUnit schedule description is:

```
[beforeGet:afterGet]@readThread1 → beforeAdd@main
```

The description denotes that the `beforeGet` and `afterGet` events in the `readThread1` thread happen before the `beforeAdd` event in the `main` thread. “[]” indicates there is no thread preemption between the `beforeGet` and `afterGet` events. SPDebugger needs more SPs to define such blocking schedule.

6. Related Work

In literature, there are two types are concurrency bug reproducing techniques. One is automatic reproduction [12]–[15], and the other is interactive reproduction [6]–[9].

Automatic approaches leverage runtime or postage information to generate test cases that can reproduce concurrency bugs. Leap [12] and Clap [13] are typical record-and-replay techniques. They monitor the shared memory states at runtime, and deterministically replay what has been recorded. The concurrency bug may not happen until a very long run of program, and it requires to record huge amount of data in this case. However, replaying all events can be expensive, so researchers explored replaying events only after a buggy behaviour is triggered. A typical such technique is called execution synthesis [14], [15]. It analyzes the thread dump information from deadlock to reconstruct a test case that triggers an identical bug.

Interactive approaches follow predefined intentions to schedule threads executions, and are useful for concurrency debugging. For example, MultithreadedTC [6] controls thread executions based on a global clock that is measured by logical ticks. A thread is held, when its ticking time is not arrived. The clock advances when all threads hold. As another example, IMUnit [7] enforces thread interleavings based on events. A schedule defines the partial order among events. They can enforce the threads executions in the test cases, but have no control over the program under test.

Concurrent Breakpoints [8] is a light-weight and programmatic way to reproduce a concurrency bug. A concurrent breakpoint represents the locations and program state predicates. A thread is suspended when concurrent breakpoint is triggered. Programmers can insert concurrent breakpoint code to a program under test to reproduce the bug. Concurrit [9] works on C/C++ concurrency bug reproduction. Besides enforcing threads executions with predefined schedule, it can explore all possible schedules under predefined constraints to trigger concurrency bugs. However, these approaches do not support fine-grained debugging.

SPDebugger combines a global clock, events and conditional breakpoints. It considers both source code and binary code of the program under test. Furthermore, it supports the in-line and loop debugging scenarios, complementing existing approaches.

7. Conclusion

Although various concurrency reproducing approaches are proposed, we argue that they have limited abilities to handle fine-grained debugging scenarios such as the in-line debugging and the loop debugging. In addition, most approaches need to modify source code at debugging time, which does not apply on binary code in third party libraries. To address the limitations, in this paper, we propose a novel approach, called SPDebugger, that supports finer debugging scenarios. SPDebugger allows defining thread execution schedules with an XML file, and uses instrumentation and JDI techniques to debug a program without modifying its source code. Our evaluation results show that SPDebugger handles more debugging scenarios without increasing much effort.

There are a few points worth keeping on study in the future:

- Java program interleaves at instruction level, but some
instructions cannot be located at source code level. For example, “i++” can be actually compiled into “iconst_0”, “istore_1”, “iinc 1,1”, “ilaod_1” and “ireturn”. But there is no way to set SPs among these instructions when only source codes are considered. More sophisticated approaches are required to handle such situation.

- There are certain patterns for thread interleavings, such as the blocking idiom designed in IMUnit. We plan to summarize the patterns and design corresponding DSL specifications, so that users can design schedules more easily and save efforts.
- We plan to implement more convenient GUI for users to design schedules inside the IDE.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive comments. This work is sponsored by the 973 Program in China (No. 2015CB352203), the National Nature Science Foundation of China (No. 61572312, No. 61572313, and No. 61272102), and the grant of Science and Technology Commission of Shanghai Municipality (No. 15DZ1100305). This work is also supported in part by Japan Society for the Promotion of Science, Grant-in-Aid for Research Activity Start-up (No. 16H07031).

References

[1] G. Pinto, W. Torres, B. Fernandes, F. Castor, and R.S.M. Barros, “A large-scale study on the usage of java’s concurrent programing constructs,” Proc. Journal of Systems and Software, vol.106, pp.59–81, 2015.

[2] Y. Eytani, K. Havelund, S.D. Stoller, and S. Ur, “Towards a framework and a benchmark for testing tools for multi-threaded programs,” Proc. Concurrency and Computation: Practice and Experience, vol.19, no.3, pp.267–279, 2007.

[3] S.M. Melo, S.R.S. Souza, R.A. Silva, and P.S.L. Souza, “Concurrent software testing in practice: A catalog of tools,” Proc. 6th A-TEST, New York, NY, USA, pp.31–40, Aug. 2015.

[4] K. Lu, X. Zhou, T. Bergan, and X. Wang, “Efficient deterministic multithreading without global barriers,” Proc. 19th PPoPP, New York, NY, USA, vol.49, no.8, pp.287–300, Feb. 2014.

[5] L. Wildman, B. Long, and P. Strooper, “Dealing with nondeterminism in testing concurrent java components,” Proc. 12th APSEC, pp.393–400, IEEE, Dec. 2005.

[6] W. Pugh and N. Ayewah, “Unit testing concurrent software,” Proc. 22nd ASE, New York, NY, USA, pp.513–516, Nov. 2007.

[7] V. Jagannath, M. Gilgoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov, “Improved multithreaded unit testing,” Proc. 19th FSE, Szeged, Hungary, pp.223–233, Sept. 2011.

[8] C.S. Park and K. Sen, “Concurrent breakpoints,” Proc. 17th PPoPP, New York, NY, USA, pp.331–332, March 2012.

[9] T. Elmas, J. Burnim, G. Necula, and K. Sen, “CONCURRIT: a domain specific language for reproducing concurrency bugs,” Proc. 34th PLDI, Seattle, WA, USA, pp.153–164, June 2013.

[10] E. Vainer and A. Yehudai, “Taming the concurrency: Controlling concurrent behavior while testing multi threaded software,” Proc. CoRR, vol.abs/1409.0982, 2014.

[11] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao, “Jaconitebe: A benchmark suite of real-world java concurrency bugs (T),” Proc. 30th ASE, Lincoln, NE, USA, pp.178–189, Nov. 2015.

[12] J. Huang, P. Liu, and C. Zhang, “Leap: Lightweight determinis- tic multi-processor replay of concurrent java programs,” Proc. 18th FSE, New York, NY, USA, pp.207–216, Nov. 2010.

[13] J. Huang, C. Zhang, and J. Dolby, “Clap: Recording local executions to reproduce concurrency failures,” Proc. 34th PLDI, New York, NY, USA, pp.141–152, June 2013.

[14] J. Röbler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea, “Reconstructing core dumps,” Proc. 6th ICST, Luxembourg, Luxembourg, pp.114–123, March 2013.

[15] C. Zamfir and G. Candea, “Execution synthesis: a technique for automated software debugging,” Proc. 5th EuroSys, Paris, France, pp.321–334, April 2010.
Haibo Yu received the Ph.D. degree in computer science from Kyushu University, Japan, in 2009. She joined the School of Software at Shanghai Jiao Tong University in March, 2010. Her research interests include software engineering, information retrieval and Web application systems.

Jianjun Zhao received the Ph.D. degree in Computer Science from Kyushu University (Japan) in 1997. He joined the Department of Computer Science and Engineering, Fukuoka Institute of Technology (Japan) as an Assistant Professor, and then was promoted to be an Associate Professor in 2000. Since November 2005, he has been with the School of Software, and then the Department of Computer Science and Engineering, Shanghai Jiao Tong University (China) as a Professor. From April, 2016, he joined the Department of Advanced Information Technology, Kyushu University (Japan) as a Professor. His research interests include program analysis for software engineering and compiler optimization.