Anomaly detection in concurrent Java programs using dynamic data flow analysis

Kassem Saleh\textsuperscript{a,}*\textsuperscript{,}, Abdel Aziz Boujarwh\textsuperscript{b}, Jehad Al-Dallal\textsuperscript{b}

\textsuperscript{a}Department of Computer Science, American University of Sharjah, Box 26666, Sharjah, United Arab Emirates
\textsuperscript{b}Department of Computer Engineering, Kuwait University, Kuwait City, Kuwait

Received 24 May 2000; revised 2 February 2001; accepted 10 September 2001

Abstract

Concurrency constructs are widely used when developing complex software such as real-time, networking and multithreaded client–server applications. Consequently, testing a program, which includes concurrency constructs, is a very elaborate and complex process. In this work, we first identify the different classes of synchronization anomalies that may occur in concurrent Java programs. We then consider testing concurrent Java programs against synchronization anomalies using dynamic data flow analysis techniques. Moreover, we show how the data flow analysis technique can be extended to detect such anomalies. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Dynamic data flow analysis; Instrumentation; Concurrency; Software testing; Synchronization anomaly

1. Introduction

Software testing is the process of executing a program with the intent of finding errors. It is a critical and important phase of the application software development life cycle. In addition, it is a time consuming and costly stage. In a typical programming project, approximately half of the effort is spent on testing activities (i.e. validation and verification), a large component of which is testing the program under development. The main goals for performing these life cycle activities are to increase the reliability and quality of the developed software.

A large portion of a high-level program consists of data declarations and manipulation statements. Data flow analysis (DDFA) is an effective approach to detect the improper use of data in the software under test (SUT) [10]. DDFA analyzes the sequence of actions affecting each data item in the SUT. Unreasonable things that may happen to a data item are referred to as data anomalies. DDFA of a SUT can be performed either statically or dynamically. In static DDFA, the source code is inspected and the sequences of uses of the data items in the SUT are tracked, without running the program. However, in the dynamic DDFA, the sequences of actions are tracked during execution of the program. Huang [1] introduced the idea of program instrumentation in which

the original source code is inserted with probes to indicate the data actions. Data flow anomalies are then dynamically detected by analyzing probe information generated during the program execution. Furthermore, data flow anomalies can be traced through state transitions instead of sequences of actions [1].

The dynamic data flow analysis (DDFA) technique is extended to test programs written in FORTRAN [2], Cobol [3], Pascal [4] and C [5,6]. In such programming languages, data is used in functions and manipulated by basic operators. In addition, the dynamic data flow analysis has been extended to test C++ [7] and Java [8] programs considering their object-oriented nature. In object-oriented programming languages, a simple assignment can lead to a sequence of actions since all the data, operators and functions of an object are encapsulated within the object itself.

All DDFA techniques are directed to analyze single-process programs. However, Cheung [9] and Taylor [10] introduce algorithms to detect anomalies in concurrent programs using static data flow analysis. In the concurrent software, data anomalies can occur in more subtle ways than in single-process programs. Moreover, concurrency creates an environment in which new types of data flow and synchronization errors such as deadlocks may occur. Furthermore, Naumovich et al. [14] presented an algorithm for obtaining information about which statements in a concurrent Java program may happen in parallel. This information can then be used to perform some program

* Corresponding author. Tel.: +971-6585-555; fax: +971-6585-8581.
E-mail address: ksaleh@aus.ac.ae (K. Saleh).
optimization, debugging and to detect synchronization anomalies. However, the complexity of their algorithm is cubic in the size of the SUT.

Java is a relatively new programming language developed by Sun Microsystems [11,12]. It is a general-purpose object-oriented programming language. After compiling a Java program, a machine-independent byte-code representation is obtained, which can then run anywhere either on the internet using Java-enabled web browsers or on a standalone personal computer or workstation. Concurrency is one of the advanced elements of Java’s object model [13]. Concurrency in Java allows us to write a program that can create one or more threads that can run completely independently or in synchronization. Multithreading is useful for writing animations, networking programs, server-based applications and real-time programs among other types of applications.

In an earlier work, we considered testing single-thread Java programs [8]. However, the aim of this paper is to extend the DDFA approach to test concurrent Java programs. We discuss different classes of data flow and synchronization anomalies that may occur in Java programs. Moreover, we show how to detect such anomalies.

The paper is organized as follows. In Section 2, we give an overview of concurrency and multithreading in Java. Section 3 discusses the classes of anomalies in concurrent Java programs. In Section 4, we describe the extensions of DDFA to test concurrent Java programs. In Section 5, we apply the extensions to an example of a Java program. Finally, in Section 6, we conclude the paper.

2. Concurrency and multithreading in Java

In Java, concurrency is implemented using the thread concept. Each thread is a single sequential flow of control. In multithreaded Java programs, a Java Virtual Machine can support multiple flows of execution in the same program. Moreover, the threads are executed in a shared memory. Thus, if a global variable is changed in one thread, all the other running threads will observe the change. On a multiprocessor machine, different threads may run concurrently or in a synchronized manner. However, on a single processor machine, threads are supported only by time-slicing the processor. Java supports threads at the language level instead of using helper library as in C.

Threads introduce a need for synchronization. Java uses a synchronization concept like a monitor to allow only one thread at a time to execute a critical part of the code. Monitors, in Java, are implemented using the synchronized keyword. Code within a synchronized block cannot be accessed by more than one thread at a time. Synchronization can be enforced at the variable, object, block, method and class levels as shown below.

```java
synchronized (object) {
    // critical statements
}
```

The synchronized statement states that only one thread can execute the critical statements on an object at a time. Synchronized keyword can be used also to declare that a method is synchronized.

```java
synchronized void critical_method()
{
    // critical statements
}
```

The above code means that only one thread can execute critical_method on any object at a time.

```java
void method()
{
    synchronized (this)
    {
        // critical statements
    }
}
```

Java implicitly associates a lock with each object. When a thread attempts to enter an object, the synchronized statement locks the object. All other threads attempting to enter the synchronized region have to wait. After the execution of the synchronized region is completed, the synchronized statement unlocks the same lock. At that time, one of the waiting threads succeeds in entering the object.

Java provides Thread and ThreadGroup classes to create and manage threads. A thread object is created using one of the Thread class constructor methods. To activate a thread object, some other thread should invoke the start method of the created thread object, which, implicitly invokes its mandatory run method. The activated thread remains alive until it is stopped either normally or abruptly. A live thread stops if the invoked run method terminates normally using a return statement or abruptly because of a thrown exception. In addition, the thread can be stopped by a successful invocation of the stop method of the thread object. Finally, a thread is stopped if some thread successfully invokes the exit method of the class Runtime, which stops all running threads on the Java Virtual Machine. Once a thread has been stopped, it is no longer alive and it cannot be restarted.

Each object in Java has a wait set for the set of threads.
When the object is first created, its wait set is empty. The class Object contains three methods to support an efficient transfer of control from one thread to another: wait, notify and notifyAll. These three methods use the wait sets of the objects. The wait method should be called only when the current thread already owns the lock on the object; otherwise, an illegal exception is thrown. When the wait method is called, the current thread places itself in the wait set for the object, unlocks the object and becomes disabled. Any other objects locked by the waiting thread remain synchronized. The waiting thread can be removed from the wait set and re-enabled for thread scheduling if some other thread invokes the notifyAll method for this object, or invokes the notify method for the object and the waiting thread happens to be chosen to be awakened. If the current thread, which calls notify or notifyAll method does not own the lock for the object, an illegal exception will be thrown. Successful notifyAll calling causes all the threads waiting on the object to be awakened. The notifyAll method is invoked automatically as the current thread that owns the lock dies. The thread can also be removed from the wait set and an interrupted exception is thrown, if the thread is interrupted by another thread, while it is waiting. If the wait method is associated with a time, the thread is re-enabled, if not being re-enabled by one of the above ways yet, when the specified amount of real time has elapsed. After re-enabling the thread, it competes in the usual manner with other threads to gain the object’s lock.

An alive thread can be suspended or resumed when it or another thread successfully calls the suspend or resume method of the thread object that it represents. A suspended thread does not release its locks. However, it remains idle.

When a thread calls the join() method of a Thread object, the current (caller) thread waits until the thread represented by the Thread object is no longer alive. Thus, if a Thread object is created, started and required to be waited for to finish its execution, its join() method should be called. In the following example, an aThread object is created and started. To wait for the thread represented by aThread object to stop, its join method is called.

```java
Thread aThread = new Thread(anObject);
aThread.start();
try {
    aThread.join();
} catch (InterruptedException ie) {...}
```

If the join() method is invoked with a time, the current thread waits until either the new current thread is stopped or a certain a mount of real time has elapsed.

yield() and sleep() methods cause the current thread to yield, allowing the thread scheduler to choose another runnable thread for execution. A yielding thread can be re-scheduled at any time. However, a sleeping thread is not allowed to be scheduled for further execution until the specified time has elapsed. This time is passed as a parameter to the sleep() method.

Finally, a thread can be destroyed without releasing its locks using the destroy() method. This method could lead to undesirable situations since it prevents other threads from entering monitors locked by the destroyed thread.

Thread can be prioritized. When a running thread creates a new Thread object, the priority of the created thread is by default set to the priority of the creating thread. However, it can be changed efficiently at any time by invoking the setPriority() method of the created Thread object. When there is a competition for gaining resources (i.e. monitors), threads with higher priorities are given preference over threads with lower priorities. However, this does not mean that a highest priority thread will always be running.

Standard Java is documented in Refs. [11,12]. In the rest of the paper, we refer to start(), stop(), wait(), notify(), notifyAll(), suspend(), resume(), join(), yield(), sleep() and destroy() methods as thread synchronization methods. For full details on the Java concurrency model and multithreading, the reader can refer to Ref. [15].

3. Data flow and synchronization anomalies

3.1. Data flow anomalies

In an earlier work, we have discussed the DDFA for single-thread Java programs [8]. The data encapsulation problem caused by the object-oriented nature of Java is solved using the explicit data name itself. The data object, class and the accessibility domain of the data are also implicitly included in the inserted probes showing the data manipulation event. Since, in Java programs, threads are represented by Thread objects, their associated data can be dealt with as discussed earlier. Extensions of the dynamic data flow analysis discussed in Ref. [8] are still valid to detect the data flow anomalies that may occur in concurrent Java programs. The additional effort will be in the test data generation phase. In testing concurrent programs, we have to execute every interleaving of the statements of all processes, which may be executing concurrently. When this is done for the instrumented SUT, all anomalies can be detected by tracking the sequence of actions of each group [8] for each execution, using the extended state transition diagram. Thus, in the rest of the paper, we shall focus our work on dealing with synchronization anomalies.

3.2. Synchronization anomalies

Actions that can be performed on a created thread, in a Java program, are start (s), stop (t), wait (w), notify (n), suspend (u), resume (r), join (j), yield (y), sleep (l), destroy (d) and dispatch (assign a processor) (h). The following are
the anomalies that we wish to detect according to the listed actions.

1. Communicating with a non-started thread. If this occurs in a program, no compilation error will be issued and such communications will be neglected. However, it is considered a suspicious activity. An execution during which this anomaly occurs will have an event sequence of the form “(*s)p” for some program thread, where (*s) is any synchronization action except start action, and p is an arbitrary event sequence.

2. Communicating with a started but not dispatched thread. An execution during which this anomaly occurs will have an event sequence of the form “s(*/h)p”, for some program thread.

3. Starting an alive thread. This occurs when the start method of a thread is called more than once. This is an error since no thread can be executed concurrently with itself. An execution during which this anomaly occurs will have an event sequence of the form “spsp”, for some program thread.

4. Communicating with a terminated thread. This is an error even if the terminated thread is restarted because it is not allowed to restart a terminated thread. An execution during which this anomaly occurs will have an event sequence of the form “t*p” or “d*p”, for some program thread.

5. Resuming a non-suspended thread. This anomaly includes resuming an active thread, which is permitted but negligible and indicates a suspicious activity. An execution during which this anomaly occurs will have an event sequence of the form “p(*/u)p”, for some program thread.

6. Communicating with a waited for thread before notifying it. Typically, a blocked thread must be notified before being accessed. An execution during which this anomaly occurs will have an event sequence of the form “pw(*/n)p”, for some program thread.

7. Communicating with a suspended thread before resuming it. This anomaly includes suspending a suspended thread, which is permitted but indicates a suspicious activity. An execution during which this anomaly occurs will have an event sequence of the form “pu(*/r)p”, for some program thread.

8. Communicating a yielded thread before being dispatched. An execution during which this anomaly occurs will have an event sequence of the form “py(*/h)p”, for some program thread.

9. Communicating with a sleeping thread. An execution during which this anomaly occurs will have an event sequence of the form “pl(*/h)p”, for some program thread.

10. Communicating with a notified but not dispatched thread. An execution during which this anomaly occurs will have an event sequence of the form “pn(*/h)p”, for some program thread.

11. Having a non-notified waiting thread (infinite waiting). An execution during which this anomaly occurs will have an event sequence of the form “pw”, for some program thread.

12. Having a non-resumed suspended thread, which is similar to the infinite waiting situation. An execution during which this anomaly occurs will have an event sequence of the form “pu” for some program thread.

13. Issuing a notification for threads waiting on an object while the object waiting set is empty (no waiting threads).

4. Extension of dynamic data flow analysis

Since we are focusing on dealing with synchronization anomalies, and not with other data flow anomalies, we consider threads as units. Threads are represented by objects. Thus, we will deal with objects and not with their attributes, which eliminates the data hiding problem caused by the object-oriented nature of Java programs.

In this section, we show how to deal with the threads in a concurrent Java program. We show two different techniques. In the first one, we track the use of the names associated with the threads in the Java program SUT. The other technique uses the explicit names and locations of the objects that represent the threads. For each of the listed techniques, we explain the information needed to be included in the inserted probes. Moreover, we discuss the locations at which the probes should be inserted. Next, we construct the state transition diagram and discuss the DDFA method for concurrent Java programs. Finally, we compare the two techniques.

4.1. Tracking thread names

In this technique, we associate a unique name to each thread, and then, we refer to this name whenever a thread action occurs. Thus, we can declare a class with an identifier attribute and a method that returns a unique identifier. The class should be declared public to be accessed everywhere in the SUT. Moreover, the attribute should be declared static to be global for all class instances. The attribute, as we assumed, can be an integer variable. This variable can be initialized to zero when it is defined or at the beginning of the program execution, and incremented whenever a thread is created. The following shows the code of such a class.

```java
class identifier {
    static int id = 0;
    int getId() { return id++; }
}
```

Since the getId method can be accessed by more than one thread at a time, more than one thread can have the same identifier number. For example, in the case of two threads,
trying to get Ids, both threads will have the same id value if one of them reads the value, and before incrementing it, the other one reads the same value. To overcome this concurrency problem, we can declare the getId method by the synchronized keyword as follows.

```java
synchronized int getId() { return id++; }
```

Normally, the run method of class Thread is overridden to meet the program specifications. For testing purposes, if the run method is not already overridden in the SUT, we have to override it. If the beginning of the overridden method blocks, we have to first create an identifier object. Then, we can name the thread. This can be done using the following code.

```java
identifier Oidentifier = new identifier();
this.setName(new String().valueOf(Oidentifier.getId()));
```

The first statement creates an Oidentifier object of type identifier. The second one associates the identifier to the thread presented by the object that calls the run method. The `valueOf()` method changes the integer type value returned by `getId` method to string to match the `setName` method parameter type. `setName` method belongs to the Thread Java class.

### 4.1.1. Contents and placement of probes

In this section, we discuss the place at which the probes for synchronization actions, required for dynamic testing purposes, should be inserted in the code of the SUT. Each probe should include the identifier of the thread which causes the action, the identifier of the thread at which the action is done and the type of the action. We assume that no action is done on a thread meant by an inserted probe between the execution of the action and the probe associated with it. This can be guaranteed using some flags. To simplify our analysis, we will classify the actions as follows.

#### 4.1.1.1. Start

The probe for the start action of a thread should be inserted in the implemented run method immediately after the statements that set the thread identifier. Since knowing that the current thread does nothing for the analysis process, it is not important to include it in the probe. As a result, we will have the following code.

```java
public void run() {
    ...// set the thread identifier as discussed above
    System.out.println("\"+ this.getName() + ",s\")
    ...// do something useful
}
```

The run method starts by identifying a unique name for the thread as discussed earlier. Then, a probe is instrumented to print the thread name and the character s to indicate its starting action. Thus, this probe will be printed whenever a thread calls the start method.

Moreover, since executing the run method includes, implicitly, that the thread is dispatched, we have to follow the starting action probe by a dispatching action.

#### 4.1.1.2. Join

Calling the join method causes the current thread to wait. Thus, before such a statement we have to instrument a joining action for the current thread. As an example, for the statement `anObject.join()`, we have to instrument the following probe:

```java
System.out.println("\" + anObject.getName() + ",\" + Thread.currentThread().getName() + ",t\")
//t denotes join action
```

#### 4.1.1.3. Others

The thread meant by any of the stop, suspend, resume, destroy, sleep, yield or wait actions is the one represented by the object associated with it (e.g. the action in the `anObject.stop();` statement is done on the thread represented by the `anObject` object). Determining the current thread is not important. Thus, as an example, for the above statement, the following probe should be inserted.

```java
System.out.println("\" + anObject.getName() + ",t\")
//t denotes stop action
```

It is more realistic to gather the thread status while it is active. Therefore, the probes for the stop, suspend, destroy, sleep, yield and wait actions should be inserted before the action statement. Moreover, the probe for the resume action should be inserted after the action.

A thread can be stopped implicitly through a normal return from the run method. Thus, at the end of the run method block and before the return statement, we have to instrument a probe indicating the stop action as follows.

```java
public void run() {
    ...// do something useful
    System.out.println("\" + this.getName() + ",t\")
    //t denotes stop action
    return;
}
```

When a sleeping, waiting or yielded thread becomes active (dispatched), the statement immediately after the sleep, wait or yield method call statement will be executed. Thus, a probe should be inserted after such calling statements to indicate the dispatching action. Waiting thread cannot be dispatched unless it is notified. Thus, an indication for the notification action should be instrumented too.

#### 4.1.2. State transition diagram

As discussed in Section 3, in Java programs, there are 11 types of actions that can be performed on a created thread.
They are: start (s), stop (t), wait (w), notify (n), suspend (u), resume (r), join (j), yield (y), sleep (l), destroy (d) and dispatch (assign a processor) (h). During the program execution, a thread can be at any of the following nine states: state C (created), state Y (ready), state R (running), state W (waiting), state J (joined), state D (dead), state S (sleeping), state U (suspended) and state A (abnormal). A thread enters state C when it is created. It enters state Y if it is started, notified, yielded, resumed, the sleeping time expires or the thread joined to it is destroyed or stopped. The thread enters state R if it is dispatched. The thread enters states J and U if it is joined and suspended, respectively. When a thread performs a wait or a sleep action, it enters states W or S, respectively. State D is entered if a thread is stopped or destroyed. Finally, a thread enters state A if one of the first 10 anomaly types listed in Section 3 occurs. Moreover, it is an anomaly to end the life of a thread on a state other than the D state. If such a case occurs, it indicates an infinite waiting. Fig. 1 shows the state transition diagram of a thread life cycle.

4.1.3. Analysis method
The analysis process is performed in two main steps. First, we instrument the SUT by all needed probes as discussed in Section 4.1.1. Then the sequence of actions is dynamically tracked.

In the second stage, we execute the instrumented program, which prints the sequence of actions executed by the program thread. The printed probes are then grouped in such a way that each group contains all probes for actions on one thread. Moreover, if the thread is joined to another, a copy of a printed destroys or stop action probe for the joining thread (if it exists) should be included in the joined thread group in its execution order. For example, if we have the following output probes: ..., (1, 2, j), ..., (r, 1, t), ..., we will have ..., (1, 2, j), (r, 1, t), ... in the thread ‘2’ group, and ..., (r, 1, t) in the thread ‘1’ group. Finally, we track the sequence of actions of each group using the state transition diagram of the thread life cycle.

4.2. Tracking object names
In this technique, we deal with objects that represent the threads instead of the threads themselves. The relation between the live threads and objects representing them is one-to-one. A single object can represent no more than one live thread. Thus, the information required to identify a thread is the same as that used to identify the object representing the thread. To identify an object, we should consider its name and the location at which it is declared. The declaration location includes the class name, the method name and method argument types (if the object is a local variable). Consider the following example.

```java
Class Test1 {
    void Method1(int k) {
        Thread T1 = new Thread();
    }
    void Method2(int k) {
        Thread T1 = new Thread();
    }
}
Class Test2 extends Test1 {
    void Method1(int k) {
        Thread T1 = new Thread();
        Thread T2 = new Thread();
        ...}
    void Method1(boolean flag) {
        Thread T1 = new Thread();
        ...}
    void Method2(int k) {
        Thread T1 = new Thread();
        ...}
}
```

For an action on the local variable T1 created in Test2, Method1(int k), if the object name is not included in the inserted probe, it will be ambiguous with object T2. Moreover, the ambiguity occurs with T1 in class Test1, T1 in Method2(int k) and T1 in Method1(boolean flag), if the class name, method name and method arguments types, respectively, are not included in the inserted probe.

As discussed in Section 4.1.1, each probe should include the identifier of the thread which causes the action, the identifier of the thread at which the action is done and the type of the action. Thus, a probe in the tracking object names technique should contain nine parameters. Four parameters are needed to identify each of the above two threads, and one for the action type. The placement of the probes is similar to that for the Tracking Thread Names Technique discussed in Section 4.1.2.

An instance variable objId of class String is used to hold the identifier for each of the created Thread objects. The identifiers should be determined immediately after a thread is created. This is done by a call to a method set ObjectId() with the object name, method name, method argument information and class name of the corresponding thread as string parameters. The setObjId() is a straightforward method that assigns the concatenated version of all its arguments to the instance variable objId of that object as follows.

```java
    void setObjectId(String oName, mName, mArgs, cName) {
        this.objId = oName + " " + mName + " " + mArgs + " " + cName;
    }
```

The thread object identifier is then referred to in the probe statements by using the dot notation on the instance variable as this.objId inside the run method or as objectname. objId, elsewhere in the program.

The same state transition diagram and analysis method used in the Tracking Thread Names Technique can then be used to detect the synchronization anomalies in the Tracking Object Names technique.
### 4.3. Comparison

Generally, determining the parameters of probes used in the tracking thread names technique is easier than determining those for the tracking object names technique. Moreover, the additional code to the SUT when using the first technique is less than that for the second technique. However, we may face some difficulties in determining the program object that represents a thread identified by its name, unless we add a probe that matches an object with its thread name when creating such an object. No such problem will be faced when the second technique is used.

However, in some cases, only one of the techniques can be applied. If the threads in the SUT have already names used in the program, we cannot change these names, unless the changes affect the program functionality or we consider these names when we assign names to threads (we do not assign new names to such threads). Consider the following code:

```java
AMethod() {
  ...
  if (anObject.getName() = = 'new') { ... }
}
```

If the name of the thread presented by the anObject object is ‘new’ and is changed for testing purposes, the code inside the if statement block will not be reached.

On the other hand, if we have different threads represented by different objects that have the same name and location, we cannot use the tracking object names technique. In this case, the objects will be considered wrongly as representing the same thread, and we will not be able to distinguish among them. Consider the following code:

```java
aMethod() {
  while(true) {
    Thread t = new Thread();
    t.start();
    ...
  }
}
```

In the aMethod, many threads are created and started and each of them is represented by different objects. However, each of these objects has the same name ‘t’ and located in the same class and method. Thus, if we apply the tracking object names technique, we will have undesired results. However, such a problem will not be faced, if the Tracking Thread Name technique is used.

### 5. Example

In the example (listed in Appendix A), class aClass extends the Thread class. Two threads are created: t1 and t2. However, t2 must execute the special code in the run method before t1. Therefore, we have used some flag variables and suspended thread t1 for a while.

To test the program, we instrument it. Appendix B lists the instrumented program and the output probes for each of the Tracking Thread Names and Tracking Object Names techniques. When probes are grouped, we will have two groups as shown in Tables 1 and 2 for the Tracking Thread Names and Tracking Object Names techniques, respectively. Tracing the actions on program threads using the state transition diagram shows that the program does not have any synchronization anomaly.

### 6. Summary and conclusion

In this paper, we have proposed an extension to the conventional dynamic data flow analysis and have shown how to detect synchronization anomalies that may occur in a concurrent Java program by observing execution traces obtained by appropriate program instrumentation. Execution traces including subtraces belonging to certain predefined classes of anomalies are then identified. Two different techniques are presented to analyze concurrent Java programs: the Tracking Thread Names technique and the Tracking Object Names technique. In the former, we use the names of the threads themselves. In the latter, we identify the threads using their object names and locations. In both techniques, a state transition diagram is used to trace the collected sequence of actions and detect the anomalous program behavior. Our dynamic approach to the detection of synchronization anomalies can be complemented using an algorithm described in Ref. [14] to obtain static information on program statements that may happen in parallel, in Java programs, which may cause race conditions on critical data. The complexity of this algorithm is cubic in the size of the program. However, both presented techniques are of linear complexity in the size of the SUT.
Appendix A

The Java program example used in Section 5.

class aClass extends Thread {
    static boolean flag1 = false, flag2 = false, flag3 = false;
    public void run() {
        while(!flag1)
            flag2 = true;
        //do some thing to be executed by t2 before t1
        flag3 = true;
    }
    public static void main(String argv[]) {
        aClass t1 = new aClass();
        aClass t2 = new aClass();
        t1.start();
        while (!flag2);
        t1.suspend();
        flag1 = true;
        t2.start();
        while (!flag3);
        t1.resume();
        System.out.println("(*," + t1.getName() + ",r");
        System.out.println("(*," + t1.getName() + ",h");
    }
}

The following is the printed probes for the instrumented program example when it is executed.

(*,o,s)
(*,o,h)
(*,o,u)
(*,1,s)
(*,1,h)
(*,1,t)
(*,0,r)
(*,0,h)
(*,0,t)

Appendix B

B.1. Analysis using Tracking Thread Names technique

The following is the complete and instrumented version of the program.

class identifier {
    static int id = 0;
    synchronized int getid() {
        return ++id;
    }
}
class aClass extends Thread {
    static boolean flag1 = false, flag2 = false, flag3 = false;
    public void run() {
        identifier Identiier = new identifier();
        this.setName(new String().valueOf(Identiier.getid()));
        System.out.println("(*," + this.getName() + ",s");
        System.out.println("(*," + this.getName() + ",h");
        while (!flag1)
            flag2 = true;
        //do some thing to be executed by t2 before t1
        flag3 = true;
        System.out.println("(*," + this.getName() + ",t");
    }
    public static void main(String argv[]) {
        aClass t1 = new aClass();
        aClass t2 = new aClass();
        t1.start();
        while (!flag2);
        t1.suspend();
        System.out.println("(*," + t1.getName() + ",u");
        flag1 = true;
        t2.start();
        while (!flag3);
        t1.resume();
        System.out.println("(*," + t1.getName() + ",r");
    }
}

B.2. Analysis using Tracking Object Names technique

The following is the instrumented version of the program.

class aClass extends Thread {
    static boolean flag1 = false, flag2 = false, flag3 = false;
    String objectId;
    void setObjectId(String oName, mName, mArgs, cName){
        this.objectId = oName + " + " + mName + "+ " + mArgs + " + " + cName);
    }
    public void run() {
        System.out.println("(*," + this.objectId + ",s");
        System.out.println("(*," + this.objectId + ",h");
        while (!flag1)
            flag2 = true;
        //do some thing to be executed by t2 before t1
        flag3 = true;
        System.out.println("(*," + this.objectId + ",t");
    }
    public static void main(String argv[]) {
        aClass t1 = new aClass();
        t1.setObjectName("t1", "main", "(String)", "aClass");
        aClass t2 = new aClass();
        t2.setObjectName("t2", "main", "(String)", "aClass");
        t1.start();
        while (!flag2);
        t1.suspend();
        System.out.println("(*," + t1.objectId + ",u");
        flag1 = true;
        t2.start();
        while (!flag3);
        t1.resume();
        System.out.println("(*," + t1.objectId + ",r");
System.out.println("(" + t1.objectId + ", h"); }
}

The following is the printed probes for the instrumented program example when it is executed.

(*, t1-main-(String)-aClass, s)
(*, t1-main-(String)-aClass, h)
(*, t1-main-(String)-aClass, u)
(*, t2-main-(String)-aClass, s)
(*, t2-main-(String)-aClass, h)
(*, t2-main-(String)-aClass, t)
(*, t1-main-(String)-aClass, r)
(*, t1-main-(String)-aClass, h)
(*, t1-main-(String)-aClass, t)

References