An Integrated Regression Testing Framework to Multi-Threaded Java Programs

Bixin Li[1,2], Yancheng Wang[1], and LiLi Yang[1]

[1]School of Computer Science and Engineering, Southeast University
No.2 Sipailou Road, Nanjing 210096, Jiangsu Province, P.R.China
[2]State Key Lab. for Novel Software Technology, Nanjing University
No.22 Hankou Road, Nanjing 210093, Jiangsu Province, P.R.China
bx.li@seu.edu.cn; http://cse.seu.edu.cn/people/bx.li

Abstract. Regression testing is a process to retest modified programs to examine whether or not new bugs were introduced by a modification. Currently, most of the selective regression testing methods have been presented to test non-concurrent programs, but few of them discussed the regression testing of concurrent programs. In this article, a selective regression testing framework based on reachability testing is proposed to solve the retesting problems in testing multi-threaded Java programs, where both the identification of related components and the selection of test cases are mainly concerned. The integration of selective regression testing techniques and the idea of concurrent programs reachability testing makes the framework be efficient. The adaptation of *ESYN-sequence* based test data coverage adequacy criterion improves the ability to find bugs.

Key words: Regression testing; Multi-threaded program; Reachability testing

1 Introduction

Regression testing is a process to retest modified programs to examine whether or not new bugs were introduced by a modification. In other word, one goal of regression testing is to ensure that new functionality will not affect adversely the correct functionality inherited from the original program. Selective regression testing attempts to identify and retest only those parts of the program that are related to a modification. There are two important problems need to be solved in selective regression testing[3][4]: how to identify those existing tests that must be rerun since they may exhibit different behavior in the changed program? and how to identify those program components that must be retested to satisfy some coverage criterion? But it is very pity that even though many regression testing methods have been proposed to test sequential programs, few discussed the regression testing of multi-threaded Java programs. On the other hand, Java is one of current main stream languages that is widely used to develop software in different application areas, where Java supports concurrent programming with threads. A thread is in fact a single sequential flow of control within a program, and each thread has a *beginning*, an *execution sequence*, and an *end*. However, a thread itself is not a program, it can not run on its own, it must runs within a program. Programs that has multiple synchronous threads are called *multi-threaded programs*. Fig. 1 shows a simple concurrent Java program that implements the *Producer-Consumer* problem.

```
ce1. public class Prod_Coms {
me2.  public static void main(String[] args) {
s3.   Buffer q = new Buffer();
s4.   new Thread(new Producer(q)).start();
s5.   new Thread(new Comsumer(q)).start();
          }
      }
ce6. class Producer implements Runnable {
ce7.    Buffer q;
ce8.  String name;
e9.     public Producer(Buffer q)
          {
             s10.  this.q=q;
          }
te11. public void run(){
s12.  int i=0;
s13.  while(true)
          {
s14.        if(i==0){
s15.        name="EvenNumber";
             }
          else {
s16.        name="OddNumber";
             }
s17.  i=(i+1)%2;
s18.  q.Write(name);
          }
          }
      }
ce19. class Comsumer implements Runnable{
ce20.  Buffer q;
ce21.  String inname;
e22.    public Comsumer(Buffer q){
s23.        this.q=q;
          }
```

```
te24.  public void run(){
s25.   while(true){
s26.    inname=q.Read();
s27.    use();
            }
          }
s28.  public use()
         {
          ...
s29. System.out.println(" Used name is:"+inname);
          ...
         }
       }
ce30.  class Buffer {
ce31.  String name="unknown";
ce32.  boolean bFull=false;
me33.  public synchronized void Write(String   value){
s34.   if ( bFull )
s35.   try {wait();} catch (Exception e) {}
s36.   name=  value;
s37.   try {Thread.sleep(1);} catch (Exception e) {}
s38.   bFull=true;
s39.   notify();
         }
me40.  public synchronized String Read(){
s41.   if(!bFull)
s42.   try {wait();} catch (Exception e) {}
s43.   bFull=false;
s44.   notify();
s45.    return name;
          }
      }
```

Fig. 1. A Java program describing the *Producer-Consumer* problem

The program creates two threads *Producer* and *Consumer*. The *Producer* generates an even or an odd alternatively, and stores it in a Buffer object. The *Consumer* consumes all integers from the Buffer as quickly as they become available. Threads *Producer* and *Consumer* in this example share data through a common Buffer object.

To execute the program correctly, two conditions must be satisfied: the *Producer* can not put any new integer into the Buffer unless the previously putted integer has been picked up by the *Consumer*; the *Consumer* must wait for the *Producer* to put a new integer into the Buffer if it is empty. In order to satisfy the these two conditions, the behaviors of the *Producer* and *Consumer* must be synchronized in two ways: ① the two threads must not simultaneously access the Buffer. A Java thread can handle this through the use of *monitor* to lock an object. When a thread holds the monitor for a data item, other threads are locked out and cannot inspect or modify the data. ② the two threads must do some simple cooperation. That is, the *Producer* must have some way to inform the *Consumer* that the value is ready and the *Consumer* must have some way to inform the *Producer* that the value has been picked-up. This can be done in Java by using a collection of methods of **Object** class, where method wait() is for helping threads wait for a condition, and notify() or notifyAll() is for notifying other threads when that condition changed.

In this article, we suggest an integrated regression testing framework to test the multi-threaded Java programs. The rest of this article is organized as follows: section 2 introduces several basic concepts and terminologies; section 3 introduces the integrated framework; section 4 discusses the selective regression testing which will be adopted in our framework; section 5 introduces the reachability testing based on extended synchronization sequence; section 6 gives the case study; section 7 concludes the article and discusses the works in the future.

2 Several concepts

In this section, we will clarify the meanings of some key concepts used in this article so that we have a common concept foundation.

A *synchronization object* refers to an accessed shared variable. A *synchronization operation* refers the operation on a synchronization object, it can be divided into *synchronization reading operation* and *synchronization writing operation* on shared variables.

A *synchronization event* is the process of synchronization operations on synchronization objects. A *synchronization sequence* (or *SYN-sequence*) means a sequence of synchronization events arranged in time order, which is the executive order of the synchronization events in concurrent programs. A Feasible *SYN-sequence* means the synchronization sequence that can be really executed in the source code, while a valid *SYN-sequence* refers to the ones that are specified to be able to be executed by a software specification. In general, the feasible *SYN-sequences* and the valid ones of a program should be consistent, otherwise, there is an error in the implementation of the program under test[5]. An *event code block* (or *ECB*) is defined as the code fragments that are related to an event happened, it can be divided into *synchronization event code block* (or $SECode$) and *non-synchronization event code block* (or $NECode$). $SECodes$ means the code fragments relating to a synchronization event, $NECode$ means the code fragments relating to a non-synchronization event.

3 Basic Idea of the Framework

3.1 Test-data Adequacy Criterion

A test-data adequacy criterion is a minimum standard that a test suite for a program must satisfy[1]. An adequacy criterion is specified by defining a set of program components and what it means for a component to be exercised. An example is the *all-statements* criterion, which requires that all statements in a program must be executed by at least one test case in the test suite. Here statements are the program components and a statement is exercised by a test if it is executed when the program is run on that test. Satisfying an adequacy criterion provides some confidence that the test suite does a reasonable job for testing the program. In this article, the test-data adequacy criterion is a criterion based on Java multi-threaded Flow Diagram (or JMFD), we call it **all-feasible-ESYN-sequences** criterion:

– The *all-feasible-ESYN-sequence*s criterion is satisfied by a test suite T if for each *ESYN-sequence* S there is some test case X in T that exercises S. An *ESYN-sequence* is exercised by test case X if it is executed when the program is run with input X.

3.2 Basic Testing Steps

In this framework, the regression testing method to test multi-threaded Java programs is suggested based on traditional selective regression testing and improved reachability testing. The regression testing steps are listed here, but the detailed discussion will be presented in section 4 and section 5:

1. Identify all *ECBs* to be tested.
2. Based on the criterion of covering all feasible *ESYN-sequences*, we select an appropriate test case subset T' from T, satisfying $T' \subseteq T$.
3. Compute the feasible *ESYN-sequence* and test it deterministically for each test case in T' based on the idea of reachability testing.
4. Judge whether or not it is necessary to design new test cases to meet the coverage criterion. If the answer is positive, we should create new test cases T''.
5. Compute the feasible *ESYN-sequence* and test it deterministically for each test case in T'' based on the idea of reachability testing.
6. Create the new available test case set for the modified program based on T, T' ,T'' and record the related running information that is useful to the regression testing performed next time.

Being similar to traditional regression testing methods, step 1 and 2 are the basic tasks to select the test case set. The big difference is that this method chooses to cover all feasible *ESYN-sequences* as the criterion, but traditional methods choose to cover all feasible *SYN-sequences*, paths or branches. In this framework, the *ESYN-sequence* is composed of one or more *ECBs*, whereas each *ECB* consists of one or more $SECodes$ and $NECodes$.

4 Identifying All *ECBs* To Be Tested

To identify the *ECBs* is to find all *ECBs* that are related to a modification, different kinds of modifications will have different affections on a *ECB*, so it is necessary to clarify which types of modification will be included in this article.

4.1 Types of the Modification

The types of program modification included in this article should be *corrective modification* and *progressive modification*, thereinto:

1. Corrective modification only changes the internal behavior of a *ECB*, but doesnt change the dependence relationships between two *ECBs*.
2. Progressive modification not only changes the internal behavior of a *ECB*, but also change the dependence relationships between two *ECBs*.

Each of them can still be divided into following three sub-types of modifications: ① *statement modification* means doing some modifications to a statement or a control predicate. ② *statement insertion* means inserting a statement or a control predicate to a program. ③ *statement deletion* means deleting a statement or a control predicate from a program.

We have different ways to identity the related *ECBs* when we do different modifications to the multi-threaded programs. In this article, we will borrow concurrent program slicing techniques to identify and capture those interested *ECBs* .

4.2 Slicing Multi-Threaded Java Programs

As to concurrent programs, there are several kinds of techniques are adopted to slice them, thereinto, the technique based on *MDG* (multi-threaded dependence graph), which was proposed by Zhao[9], is the representative one of them. For easy to understand, we iterate it here in brief. The *MDG* of a concurrent Java program is composed of a collection of thread dependence graphs each representing a single thread in the program, and some special kinds of dependence arcs to represent thread interactions between different threads. Then, the two-pass slicing algorithm based on *MDG* can be described as follows: in the first pass, the algorithm traverses backward along all arcs except *parameter-out* arcs, and set marks to those vertices reached in the *MDG*; In the second pass, the algorithm traverses backward from all vertices having marks during the first step along all arcs except *call* and *parameter-in* arcs, and sets marks to reached vertices in the *MDG*. The slice is the union of the vertices of the *MDG* has marks during the first and second steps. Similarly, we can also apply the forward slices of concurrent Java programs. In addition to computing static slices, the *MDG* is also useful for computing dynamic slices of a concurrent Java program.

4.3 Identifying the *ECBs*

In multi-threaded Java programs, the *ECBs* to be identified is the same level as method, there are two strategies to identify the related *ECBs* using program slicing: according to the first strategy, we first compute the statement-level static slice with respect to the slicing criterion ¡s, v¿, where s is the modified point and v is the modified variable; if a *ECB* includes a statement or control predicate in the static slice, then mark the *ECB*. By this way, we can identify all related *ECBs*. Obviously, we can do this easily by using the method proposed by Zhao[9]. According to the second strategy, we can use hierarchical slicing model[8] to identify all related *ECBs*.

In this article, we will discuss how to use the first strategy to identify *ECBs* to be related to the modification, the steps that we propose in this article to identify *ECBs* are as follows:

1. Create Java multi-threaded program dependence graph (*MDG*).
2. Compute statement-level static slice using the modified statement and variable as the slicing criterion, basing on the graph-reachability algorithm.
3. Mark the *ECBs* that include the statements or control predicates in the static slice.

Forward slicing can be used to identify the *ECBs* affected directly or indirectly by the modified value of the variables. while, the backward slicing algorithm can be used to identify the *ECBs* which directly or indirectly affect the values of variables to be modified.

There are different identification methods of *ECBs* for different types of program modification. In this article, as examples, we only discuss the types of modifications and identification methods listed in Tab. 1. In general, for each symbol + in Tab. 1, we should compute a slice.

Tab. 1. The identification methods and the types of modifications

Types of modification / Identifying methods	Corrective modification			Progressive modification		
	Statement correction	Statement deletion	Statement insertion	Statement correction	Statement deletion	Statement insertion
Backward slicing	-	-	+	+	-	+
Forward slicing	-	+	+	+	+	+

Corrective Modification For corrective modification, the dependence representation in the dependence graph of program P is completely same as that of program P', because such modification does not change the dependence relationships between *ECBs*, Therefore, for program P', it is enough to build the internal dependence relationships only for the *ECBs* related to the modification. Based on the modified statement and the variables used in it, we can compute the corresponding slice for three types of modifications so as to identify the *ECBs* related to these modifications. The computing steps are as follows:

1. *Statement modification*: compute the forward or backward slice for such modified statement in program P'. The *ECBs* that include a statement or control predicate in the slice will be regarded as the related *ECBs*.
2. *Statement insertion*: compute the forward or backward slice for such inserted statement in program P'. The *ECBs* that include a statement or control predicate in the slice will be regarded as the related *ECBs*.
3. *Statement deletion*: compute the backward slice for such inserted statement in program P. The *ECBs* that include a statement or control predicate in the slice will be regarded as the related *ECBs*.

Progressive Modification For progressive modification, we need to build complete *MDGs* for program P and P' respectively, because the dependence relationships between *ECBs* have been changed after the modification, we should treat them differently. As we know, there are three kinds of dependence relationships between *ECBs*, i.e., *synchronization dependence*, *data dependence* and *control dependence*. Synchronization dependence is produced by calling methods wait and notify to activate event synchronizing. Data dependence is produced by the definition of a variable in one *ECB*, whereas the use of the variable in another *ECB*. Control dependence is produced by the happening of one event will be dependent on the condition in another event of the same thread. For progressive modification, we can deal with it regarding to following two cases: (1) The dependence relationships between *ECBs* have been changed but the structure of organizing *ECBs* remains unchanged. Under this condition, the modification to *ECBs* will cause the change of data dependence and control dependence, but won't cause synchronization dependence to change. For that, we can identify those *ECBs* related to computing the corresponding slice over the *MDG* of the modified program P'

based on above three types of modification: *statement modification*, *statement insertion* and *statement deletion*. (2) Both the dependence relationships between *ECBs* and the structure of organizing *ECBs* have been changed after the modification. The reason that causes the structure of organizing *ECBs* to change is the *insertion* and *deletion* of the synchronization *ECBs*. Therefore, we can identify those related *ECBs* as following steps:

1. The deletion of the synchronization *ECBs*: In the *ECBs* deleted from program P, find the statement and variables which have dependence relationships with other *ECBs* in program P and use these statements and variables as slicing criteria to compute the backward slices. The *ECBs* related to these slices will be regarded as the related *ECBs*.
2. The insertion of the synchronization *ECBs*: In program P', find those inserted *ECBs* and determine the statements and variables which are dependent on other *ECBs* in program P', we use these statements and variables as slicing criteria to compute the forward slices and backward slices. The *ECBs* related to these slices will be regarded as the related *ECBs*.
3. The movement of the synchronization *ECBs* can be used replace the insertion and deletion of the synchronization *ECBs*. If the result of moving synchronization *ECBs* causes the dependence relationships to change, we use the changed statements and variables as the slicing criteria to compute slices so as to determine those related *ECBs*. Otherwise, we needn't do anything.

4.4 Identification Algorithm: Identifying All Related *ECBs* By Using Both Backward and Forward Slicing

Program modification includes corrective modification and progressive modification. Progressive modification consists of two kinds of types: the first type has not affected the structure no matter what change you have done, the second type has changed the structure when you do some modification. The second type is a kind of complex one that can be regarded as the composition of many single-statement modifications, so it can be divided into single statement modifications. To deal with such modification, it is needed not only rebuild the program dependence graph of P', but also repartition the set of *ECBs* of P' . We compute the *ECBs* to be related for each modification to simple statement, the resulting set of these *ECBs* will be the set of *ECBs* related to the second modification.

Once the related *ECBs* are identified, we can perform the regression testing to Java multi-threaded programs as following steps : (1) choose appropriate test cases based on the relationships between old test cases and the identified *ECBs*; (2) compute the feasible *ESYN-sequences*; (3) do the deterministic testing based on selected test cases and the feasible *ESYN-sequences*.

How to build the relationships between old test cases and the identified *ECBs* and how to choose properly test case are two important and complex questions, we won't discuss them in details in this article. For simplicity, here we will focus on the identification of related *ECBs* after the introduction of the regression testing method and the construction of *ESYN-sequence*. As a case study, we use simple selection criterion: *for a given test case, if some related ECB is included in one of its ESYN-sequence, the test case is also regarded as a selected test case*. Even through the precise is not very high, this technique can insure that the set of selected test cases is safe.

5 Reachability Testing

In this section, we explain how to generate effective test sequences to satisfy the *all-feasible-ESYN-sequences* criterion.

5.1 *JMFD*: A Java Concurrent Model

To describe exactly the concurrent mechanism of a multi-threaded Java program so as to generate *ESYN-sequence*, we borrow the model from Li's method and extend its functionality by adding some new elements[7]. We call this model Java multi-threaded flow diagram (*JMFD*), where the node denotes *event*, the edge denotes *flow*. The steps for constructing *JMFD* are as follows:

1. Use square with round corner notation to denote the start node and end node of the program, marked with start or end.
2. Use square notation, with the formula S_r or S_w being filled in it, to denote synchronization read or write event respectively; use ellipse notation with a name to denote a non-synchronization event; the *creation* and *run* event of a thread is denoted as a non-synchronization event with *Tname.start* being filled in it.
3. Use solid line directed edge to denote the control flow in a program; use uniform dashed line directed edge to denote the concurrent flow in multithreaded programs.
4. Use nonuniform dashed line directed edge to denote the synchronization control flow among the synchronization events of different threads.
5. Starting from the main thread, construct the *JMFD* hierarchically till the *JMFD* for the whole program has been constructed.

In this article, we mainly concerns the behavior feature of a multi-threaded Java program for a given test case. Therefore, firstly, we should build the *JMFD* of a multi-threaded Java program for the special test case. Fig 2 shows the *JMFD* of the program in Fig. 1.

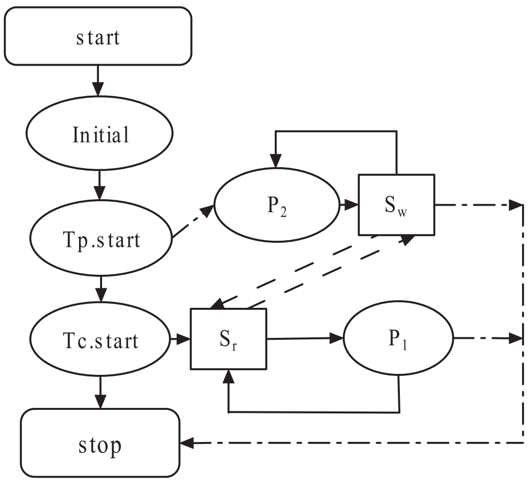

Fig. 2. The *JMFD* of *Producer-Consumer* program

5.2 Computation of the *ESYN-sequence*s

For the feasible *SYN-sequence* S of concurrent program P, the prefixes of other feasible *SYN-sequence* of P are called the *race variants*[5], and accordingly, for the feasible *ESYN-sequence* S' of concurrent program P', the prefixes of other feasible *ESYN-sequence* of P' are also called the race variants.

We can compute the race variants of synchronization sequence by building a race variant diagram that is a tree, where the node denotes a general prefix of a given feasible synchronization sequence S or denotes one of its race variants. The creation process of the tree nodes in the race variant diagram is completed by considering the all possible orderings of synchronization read or synchronization write event. The building process of race variant diagram is in fact the process to compute the variants of synchronization sequences, when the whole race variant diagram is constructed, the computation process of race variants of a synchronization sequence ends. To compute other feasible synchronization sequence, we must use a given test case and corresponding race variant of S to execute the replay operation of a concurrent program based on the prefix[2]. From the definition of *ESYN-sequence*, we have following proposition:

Proposition 1 If the time-order remains unchanged, a *SYN-subsequence* of a feasible *ESYN-sequence* is a feasible *SYN-sequence*.

So, we can compute *EYSN-sequence* by extending the computing algorithm of *SYN-sequence* so that it can deal with non-synchronization event[5][6]. For instance, *SYN-sequence*$[\![1]\!] = (S_w^1, S_r^1, S_w^2, S_r^2, ...)$ is a feasible synchronization sequence in the *Producer-Consumer* program in Fig 1, the *Consumer* thread exercises the event sequence $S[\![2]\!] = (S_r^1, P_1, S_r^2, P_1, ...)$, meanwhile, the *Producer* thread exercises the event sequence $S[\![3]\!] = (P_2, S_w^1, P_2, S_w^2, ...)$. The event sequence $S[\![3]\!]$ shows that the synchronization event S_w^1 must happen after the non-synchronization event P_2, and that the non-synchronization event P_2 must happen before the synchronization event S_w^2. Similarly, the non-synchronization event P_1 must happen between the synchronization events S_r^1 and S_r^2. So, if we insert the non-synchronization event P_1, P_2 into *SYN-sequence*$[\![1]\!]$ according to the *event sequence constraint conditions* during the thread execution[5], we can obtain the *ESYN-sequence*. Tab. 2 shows a set of *ESYN-sequences*, which is computed over the *SYN-sequence*$[\![1]\!]$.

Tab. 2. The feasible *ESYN-sequences* computed from the *SYN-sequence*$[\![1]\!]$

No.	*ESYN-sequence*
1	$(P_2, S_w^1, P_2, S_r^1, P_1, S_w^2, S_r^2, ...)$
2	$(P_2, S_w^1, P_2, S_r^1, S_w^2, P_1, S_r^2, ...)$
3	$(P_2, S_w^1, S_r^1, P_2, P_1, S_w^2, S_r^2, ...)$
4	$(P_2, S_w^1, S_r^1, P_2, S_w^2, P_1, S_r^2, ...)$

The *event sequence constraint condition* during the thread execution are as follows:

- All events in the thread, including synchronization event and non-synchronization event, must be in time-order, i.e., the event sequence belonging to the same thread in a *ESYN-sequence* must be consistent with the thread executuon event sequence

– The running of events in a thread must happen after the thread is created. As a example, in Fig.2, the event in *Consumer* thread must happen after the Initial event in the main thread.

The algorithm for constructing *ESYN-sequence* is as follows:

1. For a given test case X and a given *SYN-sequence* S, compute the race variants of *SYN-sequence* S, by changing its race condition.
2. Basing on the event sequences constraint conditions during the thread execution, add all the non-synchronization events related to race variants of the *SYN-sequence* to the race variant so as to construct the race variant of the *ESYN-sequence*.
3. Using test case X and the race variant of each the *ESYN-sequence* of S to perform the multithreaded program replay operation based on prefixes so as to compute the other feasible *SYN-sequences* and feasible *ESYN-sequence* produced.
4. For each new *SYN-sequence*, repeat steps 1,2 and 3 till no new *ESYN-sequence* produced.

6 Case Study

6.1 Corrective Modification

Fig. 1 is a typical Java multithreaded program of the *Producer-Consumer* problem. The program code can be divided into five non-synchronization *ECBs* including Initial, Tp.start, Tc.start, P_2 and P_1, and two synchronization *ECBs* including S_w and S_r. Fig 2 is a *JMFD* of the program in Fig.1 where statement 3 forms the non-synchronization event Initial representing the initialization operation, statement 4 and 5 forms non-synchronization events for the creation of threads Tp.start and Tc.start, respectively.p is the name of *Producer* thread, c is the name of *Consumer* thread. The statements from 10 to 17 form the non-synchronization event P_2 in the *Producer* thread, representing the event of producing data; statement 18 calls synchronization method Write, which forms the synchronization event, marked as S_w. Similarly, statement 26 calls synchronization method Read, which forms the synchronization event, marked as S_r.

The method called by statement 27 forms the non-synchronization event representing the consuming operation. Now, suppose that we do some changes to the program, for example, we change statement s38:bFull=true to bFull=false, then the process is as follows using our regression testing method: firstly, we should create the multi-threaded program dependence diagram, and know from the *MDG* where such modification has not caused the change of dependence relationships, so we can operate along the *case-modification* branch in the identification algorithm for identifying *ECBs* for the corrective modification. In program P', we use ¡s38,bFull¿ as slicing criterion to compute forward slice and backward slice: the backward slice is –s41,s42˝ and the forward slice is –me2, s4, te11,s13,s18,me33,s38˝. The *ECBs* are related to these resulting slices are: non-synchronization event code blocks Tp.start and P_2, and synchronization event code blocks S_w and S_r. All these *ECBs* are the *ECBs* related to the statement modification. Finally, we should select appropriate test case to finish the reachability testing, and ensure each related *ECB* will be covered by a test case at least. In other words, the coverage criterion is to cover all feasible

ESYN-sequences which includes the related *ECBs*. The testing result shows that there are some faults with the corrective modification to statement 38, it makes infeasible *ESYN-sequence*=(P_2, S_w^1, P_1, S_w^2) become feasible *ESYN-sequence*

6.2 Progressive Modification

Suppose that we delete the statement s39 in the program in Fig.1, the consequence is that the dependence edge from statements s39 to s42 will be deleted. Such modification causes the dependence relationships between *ECBs* to change, it belongs to the type of progressive modification. After the construction of *MDG* and *JMFD* of program P, we can obtain the set of *ECBs* of program P by identifying the *ECBs* along the type of statement deletion of progressive modification. The concrete steps are: (1) compute the backward static slice w.r.t. slicing criterion ¡s39,monitor¿ in program P based on the identification algorithm of related *ECBs*, the result is –s39,s42˝. The related *ECBs* that we have are synchronization event code blocks S_w and S_r; (2)select enough test case related to cover those related *ECBs* so as to do reachability testing and ensure each related *ECB* will be covered by at least a test case; (3)perform the reachability testing process. The result shows that there are some faults with the deletion of statements s39,it will cause the deadlock of the program and make the feasible *ESYN-sequences* covering S_w and S_r become infeasible *ESYN-sequences*.

7 Conclusion

In this article, we suggest a regression testing framework to test Java multi-threaded programs based on the integration of both the improved reachability testing and traditional selective regression testing of sequential programs. The adoption of selective regression testing technique makes the efficient be high, the use of reachability testing solves the problems that caused by the non-deterministic behavior of the multi-threaded programs. Meanwhile, program slicing techniques are borrowed to identify the related *ECBs* so as to increase the safety and decrease the total cost of the regression testing.

Acknowledgments. This work is partially supported by the National Science Foundation of China (No. 60473065) and partially supported by the open foundation of State Key Lab. for Novel Software Technology, Nanjing University(No.A2005 08). The authors also thank those anonymous reviewers for their valuable suggestions on the draft.

References

1. D. Binkley. *The application of program slicing to regression testing*. Information and Software Technology (I&ST) special issue on program slicing, 40 (11-12): 583-594, 1998.
2. R.Carver and K. Tai. *Replay and testing for concurrent programs*. IEEE Software, 3(1991):66-74
3. T. L. Graves, M. J. Harrold, J. Kim, A. Porters, G. Rothermel. *An empirical study of regression test selection techniques*. ACM Transactions on Software Engineering and Methodology. 10(2), 2001.

4. R. Gupta, M. J. Harrold, and M. L. Soffa *An approach to regression testing using slicing*. In: Proceedings of the Conference on Software Maintenance, November 1992.
5. G. H. Hwang, K. C. Tai, and T. L. Huang. *Reachability testing: an approach to testing concurrent software*. In: Proceedings of First Asia-Pacific conference on software Engineering, 246-255, 1994.
6. J. Lei, R. Carver. *Reachability testing of concurrent programs*. Technical Report GMU-CS-TR-2005-1, George Mason University.
7. S. Li, H. Chen, and Y. Sun. *A framework of reachability testing for Java multi-threaded programs* IEEE International Conference on System, Man and Cybernetics, 3(2004):2730-2734
8. B. Li, X. Fan, J. Pang, J. Zhao. *A model for slicing Java programs hierarchically*. J. Comput. Sci. & Technol, 19(6):848-858, 2004.
9. J. Zhao. *Slicing concurrent Java programs*. In: Proceedings of Seventh International Workshop on Program Comprehension, 126 -133, 1999