The Design and Implementation of Automata-based Testing Environment for Java Multi-thread Programs *

Heui-Seok Seo†, In Sang Chung‡, Byeong Man Kim*, and Yong Rae Kwon†

†Department of Computer Science
Korea Advanced Institute of Science and Technology, Korea
{hsseo, kwon}@salmosa.kaist.ac.kr

‡School of Information and Computer Engineering, Hansung University, Korea
insang@hansung.ac.kr

*School of Computer & Software Engineering
Kumoh National University of Technology, Korea
bmkim@cespc1.kumoh.ac.kr

Abstract

Deterministic execution testing has been considered a promising way for concurrent program testing because of its reproducibility. Since, however, deterministic execution requires that a synchronization sequence to be replayed is feasible and valid, it is not directly applicable to a situation in which synchronization sequences, being valid but infeasible, are taken into account. To resolve this problem, we had proposed automata-based testing in our previous works, where a concurrent program is executed according to one of sequences accepted by the automaton recognizing all sequences semantically equivalent to a given sequence. In this paper, we present the automata-based testing environment for Java multi-thread programs, and we design and implement key components - Automata Generator, Program Transformer and Replay Controller. Algorithms for generating the equivalence automaton of a given sequence are presented and a program transformation method is suggested in order to guide a program to be executed according to the sequence accepted by the automaton. The replay controller is also redesigned and implemented to adopt the automaton. By illustrating automata-based testing procedures with the gas station example, we show how the proposed approach does works in Java multi-threaded programs.

*This work is supported in part by the Ministry of Information & Communication of Korea (Support Project of University Foundation Research <'2000> supervised by IITA)

1. Introduction

With multi-threading of Java programs, execution of programs can be accelerated and the throughput can be enhanced. However, repeated runs of a Java multi-thread program with an identical input may yield different results due to its nondeterministic natures. This poses a problem for testing and debugging because it becomes difficult to detect and isolate program errors. Moreover, the reliability of a program cannot be established even when a correct result is obtained from an execution, because other execution with the identical input may produce an incorrect result [5, 10, 14]. To cope with such problems, a deterministic testing techniques that controls the execution of concurrent programs has been developed [4, 5, 11, 13, 12].

Specification languages for concurrent programs usually feature partial order semantics, which describe the sequencing constraints among synchronization events. Most previous works on deterministic testing have assumed that specifications and programs are equivalent [2, 6, 9]. All synchronization sequences derived from partially ordered sets in specifications can be observed in a deterministic testing. When either concurrent program accepts invalid synchronization sequences or it rejects valid synchronization sequences, it may be concluded that the program contains synchronization errors. One major limitation of classical deterministic testing is that not all synchronization sequences in specifications can be implemented according to the design decision. For example, the order between receiving a free

0-7695-1083-3/01 $17.00 © 2001 IEEE

tone of a caller and receiving a ringing tone of a callee is nondeterministic in a phone conversation. Two sequences are possible: a caller hears a free tone before a callee hears a ringing tone and a vice versa. The program should be correct with an implementation of either sequence. In this case, the program still meets its specification despite that all valid sequences are not feasible. However, classical deterministic testing based on equivalence relations would conclude that the program contains synchronization errors. To deal with such problems, we have defined the following three types of conformance relations between specifications and concurrent programs based on subsumption relations between valid event sequences and feasible event sequences in [8, 9].

- **Synchronization Equivalence:** A program P is the *synchronization equivalence* of a specification S if and only if valid(S) $\equiv$ feasible(P)
- **Synchronization Extension:** A program P is the *synchronization extension* of a specification S if and only if valid(S) $\subset$ feasible(P)
- **Synchronization Reduction:** A program P is the *synchronization reduction* of a specification S if and only if valid(S) $\supset$ feasible(P)

Here, "valid(S)" denotes a set of all valid event sequences in a specification S and "feasible(P)" denotes a set of all feasible event sequences in a program P.

To test concurrent programs with the synchronization reduction, we had proposed a new deterministic execution method, called *automata-based deterministic testing* or *automata-based testing*, in previous works [7, 8]. The basic idea of automata-based testing is to focus on the equivalence relations found among synchronization sequences. The synchronization sequences are grouped into equivalence classes so that a synchronization sequence in the same equivalence class with the given test sequence can be executed if it is feasible. When any sequence in the equivalence class is executed and produces correct results, it may be concluded that a tested program is reliable for the given sequence. Note that other sequences are not explored in the classical deterministic testing.

In this paper, we propose a testing environment where we can apply the automata-based testing method to Java multi-thread programs. We have designed and implemented three components of the environment - *Automata Generator*, *Replay Controller*, and *Program Transformer*. *Automata Generator* produces an equivalence automaton from a given sequence and a dependency table for synchronization events. An equivalence automaton is constructed to accept all sequences that show a semantically equivalent behavior with the given sequence. *Replay Controller* controls the program execution paths and *Program Transformer* transforms original Java multi-thread programs to slightly modified ones so that Replay Controller can accept alternative feasible sequences of the given sequence.

The rest of this paper is organized as follows. In Section 2, we briefly introduce classical deterministic testing. Section 3 describes the automata-based testing environment for Java multi-thread programs. In section 4, the algorithms for automaton generation, the implementation of the replay controller, and the transformation method for Java multi-thread programs are described. The gas station example is employed to show how the proposed approach works. In Section 5, we conclude our works and present the directions for future works.

2. Classical Deterministic Testing of Concurrent Programs

Classical deterministic testing approaches consist of two major steps: the selection of synchronization sequences and the forced execution of a program according to those selected sequences. Synchronization sequences can be collected by observing the order of event occurrences from the previous execution [11, 13]. It is also possible to obtain sequences by analyzing specifications and/or program structures [6, 9, 12, 14].

Once sequences are collected, we need to guide a program to follow the execution paths that are consistent with the selected sequences. Two typical methods of reproducing executions are available: the implementation-based approach and the language-based approach. In the *implementation-based approach* [1, 3], by modifying the language's implementation testers directly control executions by scheduler-controlled operations such as breakpoints, selecting the next running process, rearranging processes in various queues, and so on. This approach is efficient because only few codes are added. However, it is not portable because it is dependent on a particular compiler, a runtime system, or an operating system. In the *language-based approach* [4, 5, 10, 12, 13], a replay controller for controlling a program execution is added to a concurrent program. This controller is supposed to designate a processor to be executed on a particular execution path and delivers a program control to it blocking the execution of other processors. The controller also retrieves a program control from the processor and awakes blocked processors. The above operations are added to the front and rear of each operation in the processors. Thus, a concurrent program turns into a slightly different program because of a replay controller. Though efficiency may be affected by a replay controller and the added codes, it is highly portable and easy to develop and automate because the same mechanism can be applied to all programs implemented in the same language. We choose to follow the language-based approach.

Now, we explain the language-based approach for

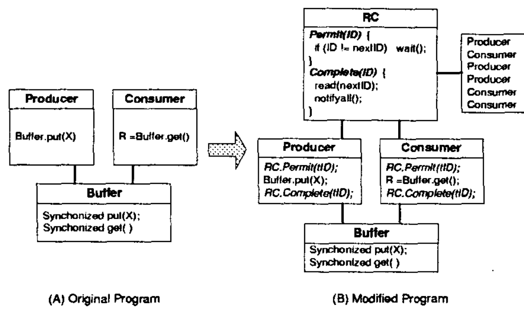

Figure 1. (a) the structure of the original producer/consumer program and (b) the structure of the modified program to apply a language-based deterministic testing technique

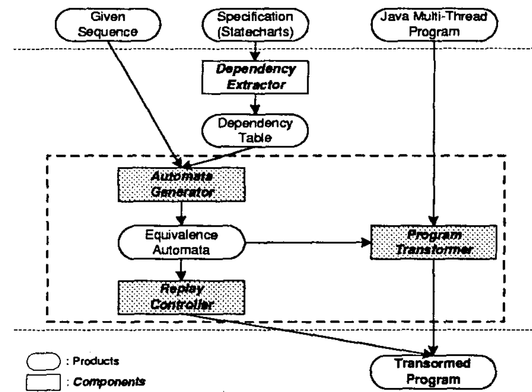

Figure 2. The automata-based testing environment for a Java multi-thread program

Java multi-thread programs in more details with the produce/consumer program (see Figure 1). In the original program in Figure 1(a), *Producer* and *Consumer* are concurrent threads. *Producer* deposits an item in buffer (*put(X)*) and *Consumer* takes out an item from buffer (*get(X)*). The program is modified to the one in Figure 1(b) to prepare for classical deterministic testing. In order to control execution of the program, *RC* is added to the program and *reqPermit(ID)/done(ID)* methods in *RC* are added to the front and rear of each method in threads. *Replay Controller(RC)* permits only one thread to proceed and blocks other threads until they get permission from *RC*. The method *reqPermit(ID)* checks whether a thread request may be granted according to a given sequence. If the thread ID matches with a current element of a given sequence, the thread obtains the program control. Otherwise the thread is blocked until its turn. After the requested action has been completed, the *done(ID)* method retrieves the program control from the thread and awakes the blocked from *RC*.

For example *Producer* in Figure 1(b) is the first current thread ID. Thus, *Producer* can gain the program control but *Consumer* is blocked. After the action of *Producer* is completed, the program control is released from *Producer* and the current thread ID becomes *Consumer*. If no thread gains a program control during the interval time defined by testers, they conclude that a given sequence is infeasible. In the classical deterministic testing, a program is guided to follow an execution path according to a given sequence. Thus, the given sequence is determined to be infeasible when the execution fails. However, this strategy won't work when the program is the synchronization reduction of its specification, i.e. some valid sequences cannot be realized in the corresponding implementation according to the design decision. This is one of the reasons why we adopt the automata-based deterministic testing approach [7, 8] for Java multi-threaded program testing.

3. Automata-based Testing Environment

Given a test sequence to be replayed and the specification of the original Java multi-threaded program, a Java program is slightly modified and then executed in the automata-based testing environment. Figure 2 represents the automata-based testing environment consisting of Dependency Extractor, Automata Generator, Replay Controller, and Program Transformer. **Dependency Extractor** generates a dependency table of synchronization events by analyzing specifications. **Automata Generator** generates an automaton which represents an equivalence class with a given sequence from a given sequence and a dependency table of events. **Replay Controller** controls the execution of the transformed program according to one of sequences accepted by the equivalence automaton. **Program Transformer** generates a transformed Java multi-thread program using an equivalence automaton. In this paper, only Automata Generator, Replay Controller, and Program Transformer are presented.

Figure 3 illustrates the process of generating an equivalence automaton with Automata Generator. This process consists of two steps: (1) the dependence graph generation and (2) the equivalence automata generation. The dependence graph represents the partial order to be satisfied in a given sequence. The equivalence automaton accepts all sequences that are equivalent to a given sequence. With this automaton, Replay Controller guides the program to follow one of sequences accepted by the automaton. To this end, the program is slightly modified with additional codes for communicating with Replay Controller as shown in Figure 4. The major difference between our approach and the clas-

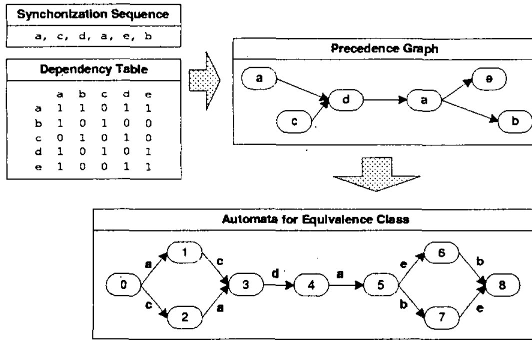

Figure 3. The example for generating the equivalence automaton: the *precedence graph* is generated from the synchronization sequence and the dependency table, and the *equivalence automaton* is generated from the precedence graph

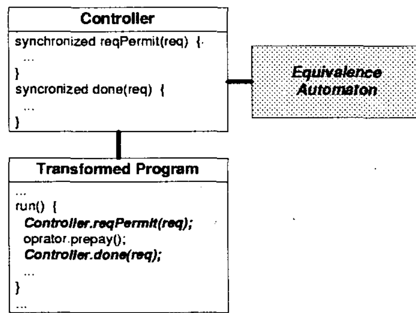

Figure 4. The structure of the modified Java multi-thread program for automata-based testing

sical deterministic testing is that the equivalence automaton of a given sequence is used instead of the sequence itself.

4. Automata-based Testing Process

In this section, we present algorithms for generating an equivalence automaton from a given sequence and a dependency table of events. Each model in automata-based testing is formally defined in [7, 8]. We also explain how to implement Replay Controller and how to transform Java multi-thread programs. We are going to demonstrate how the automata-based testing scheme does works in our environment with a gas station example.

4.1. Gas Station Problem

The gas station problem is often used as a typical example in concurrent programming [6]. Figure 5 shows a

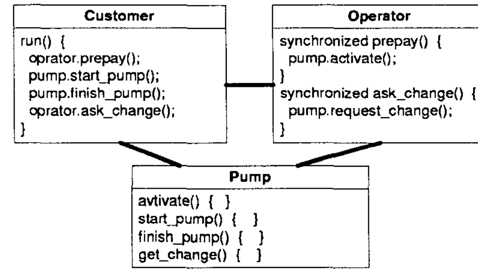

Figure 5. The structure of the gas station example: *Customer* is a thread class

structure of the gas station problem. In this example, a customer must prepay the operator for a selected pump (*prepay()*). The operator activates a selected pump if it is not being used, otherwise a customer waits for his/her turn (*activate()*). After a selected pump is activated, a customer starts pumping (*start_pump()*), finishes pumping (*finish_pump()*), and then receives changes from the operator (*ask_change()*). The operator knows the amount of the changes from the pump (*request_change()*). In this example, customers are concurrent processors and the both operator and pumps are shared resources. Hence customers are implemented as threads and methods of the class operator are implemented as the **synchronized** methods to prevent concurrent accesses from customers.

Synchronization events are the events used in concurrent threads and their execution orders are naturally nondeterministic. In the gas station problem a synchronization event is defined as E_{ijk}, which means that the customer i requests the event k for the pump j. The requested event k is one of four events: prepay ($k = 1$), start_pump ($k = 2$), finish_pump ($k = 3$), ask_change ($k = 4$). The dependency between synchronization events is defined when their order is deterministic. There exists the dependency among synchronization events requested by the same customer and among synchronization events using the same pump. Because only one Operator exists events relative to *Operator* also have the dependency, i.e., for dependency relation **D**,

$$(E_{ijk}, E_{pqr}) \leftrightarrow (i = p) \; or \\ (j = q) \; or \\ ((k=1 \; or \; 4) \; and \; (r=1 \; or \; 4))$$

For simplicity of discussion, we assume that the gas station consists of one operator, three customers, and two pumps. We also assume that customer1 and customer3 use pump1, and customer2 uses pump2. In this case, 12 events are defined: 4 events of customer1 using pump1 ($E_{111}, E_{112}, E_{113}, E_{114}$), 4 events of customer2 using pump2 ($E_{221}, E_{222}, E_{223}, E_{224}$), and 4 events of customer3 using pump1 ($E_{311}, E_{312}, E_{313}, E_{314}$). Table 1

Events	E_{111}	E_{112}	E_{113}	E_{114}	E_{221}	E_{222}	E_{223}	E_{224}	E_{311}	E_{312}	E_{313}	E_{314}
E_{111}	1	1	1	1	1	0	0	1	1	1	1	1
E_{112}	1	1	1	1	0	0	0	0	1	1	1	1
E_{113}	1	1	1	1	0	0	0	0	1	1	1	1
E_{114}	1	1	1	1	1	0	0	1	1	1	1	1
E_{221}	1	0	0	1	1	1	1	1	1	0	0	1
E_{222}	0	0	0	0	1	1	1	1	0	0	0	0
E_{223}	0	0	0	0	1	1	1	1	0	0	0	0
E_{224}	1	0	0	1	1	1	1	1	1	0	0	1
E_{311}	1	1	1	1	1	0	0	1	1	1	1	1
E_{312}	1	1	1	1	0	0	0	0	1	1	1	1
E_{313}	1	1	1	1	0	0	0	0	1	1	1	1
E_{314}	1	1	1	1	1	0	0	1	1	1	1	1

Table 1. The dependency table in the simplified gas station example

shows their dependencies, where "1" indicates that there is dependency between two events, and "0" indicates that there is not.

4.2. Equivalent Automata Generation

The process of generating equivalence automata consists of two steps: 1) generating a *precedence graph* of a given sequence with a dependency table and 2) generating an *equivalence automaton* from a precedence graph. The precedence graph represents the partial orders among events which a synchronization sequence must satisfy. **Algorithm 1** shows how to generate the precedence graph for a given sequence using a dependency table.

Algorithm 1 (Precedence Graph Generation)

```
1: Global Variables
2:   int eve_count;                       // The number of events (=12)
3:   int seq_count;                       // The length of a given sequence
4:   String Events[eve_count];            // A set of synchronization events
5:   int Dependency[eve_count][eve_count];   // A dependency table
6:   String Sequence[seq_event];             // A given sequence
7: private void get_Graph ()
8:   PrecedenceGraph = new int[seq_count][seq_count];
9:   for( int i=0 ; i<seq_count ; i++)
10:     for( int j=0 ; j<seq_count ; j++)
11:       PrecedenceGraph[i][j] = 0;
12:   end for;
    // Two events: we identify the dependency between them
13:   int pre_event=0, post_event=0;
14:   for( int i=seq_count-2 ; i>=0 ; i-)
15:     for( int k=0 ; k<eve_count ; k++)
16:       if( Sequence[i].equals(Events[k]) )
17:         pre_event = k;
18:     end for;
19:     for( int j=i+1 ; j<seq_count ; j++)
20:       for( int k=0 ; k<eve_count ; k++)
21:         if( Sequence[j].equals(Events[k]) )
22:           post_event = k;
23:       end for;
24:       if( Dependency[pre_event][post_event] == 1 && PrecedenceGraph[i][j] == 0 )
25:         PrecedenceGraph[i][j] = 1;
26:       for( int l=j+1 ; l<seq_count ; l++ )
27:         if( PrecedenceGraph[j][l] != 0 )
28:           PrecedenceGraph[i][l] = PrecedenceGraph[j][l]+1;
29: end all;
```

The algorithm initializes a precedence graph (8-12). In the precedence graph, "0" indicates that there is no dependency between two events, which means that two events are concurrent. The positive value indicates that dependency exists between two events. After initialization, the last event in a given sequence is assigned to the *pre_event* (14-18) and the next of the *pre_event* is assigned to the *post_event* (19-23). If dependency existing between the *pre_event* and the *post_event* is not represented in the precedence graph (24), the algorithm includes it in the graph as well as all other dependency caused by it (24-28). The dependency table generated by Algorithm 1 represents all partial orders in a given sequence [7, 8]. In the gas station example, assume that the sequence [E_{111}, E_{221}, E_{222}, E_{223}, E_{112}, E_{113}, E_{114}, E_{311}, E_{312}, E_{224}, E_{313}, E_{314}] is given. Then, we can obtain the dependency table shown in Table 2 by applying Algorithm 1. Figure 6 is a graphical representation of dependencies in Table 2, where nodes and edges of the graph denote events and dependencies, respectively.

Events	E_{111}	E_{221}	E_{222}	E_{223}	E_{112}	E_{113}	E_{114}	E_{311}	E_{312}	E_{224}	E_{313}	E_{314}
E_{111}	0	1	2	3	1	2	3	4	5	5	6	6
E_{221}	0	0	1	2	0	0	1	2	3	3	4	4
E_{222}	0	0	0	1	0	0	0	0	0	2	0	3
E_{223}	0	0	0	0	0	0	0	0	0	1	0	2
E_{112}	0	0	0	0	0	1	2	3	4	4	5	5
E_{113}	0	0	0	0	0	0	1	2	3	3	4	4
E_{114}	0	0	0	0	0	0	0	1	2	2	3	3
E_{311}	0	0	0	0	0	0	0	0	1	1	2	2
E_{312}	0	0	0	0	0	0	0	0	0	0	1	2
E_{224}	0	0	0	0	0	0	0	0	0	0	0	1
E_{313}	0	0	0	0	0	0	0	0	0	0	0	1
E_{314}	0	0	0	0	0	0	0	0	0	0	0	0

Table 2. The dependency table obtained by applying *Algorithm 1* to the sequence [E_{111}, E_{221}, E_{222}, E_{223}, E_{112}, E_{113}, E_{114}, E_{311}, E_{312}, E_{224}, E_{313}, E_{314}]

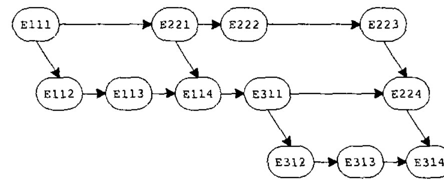

Figure 6. The dependency graph which represents dependencies in Table 2

Algorithm 2 shows how to generate an equivalence automaton from a precedence graph. In an equivalence automaton, a state is defined as a set of executed events and a transition is defined as the 3-tuple = *(a current state, an executing event, a target state)*.

Algorithm 2 (Equivalence Automata Generation)

```
1: Global Variables
2:   int tr_count, st_count          //The number of transition and states
3:   int eve_count;                  // The number of events (=12)
4:   int seq_count;                  // The length of a given sequence
5:   String Events[eve_count];       // Events
6:   int PrecedenceGraph[eve_count][eve_count];  // Precedence Graph
7: public void get_Automata()
8:   int sum_col;            // The number of preceding events
9:   int current=0, next;    // a source state and a target state
10:   int[] temp_state = new int[seq_count];
11:   int max_state = power(2, seq_count);
12:   States = new int[max_state][seq_count];            // States of an automaton
13:   Automata = new int[seq_count * max_state][3];  // An equivalence automaton
14:   for( int i = 0 ; i < seq_count ; i++ )
15:     States[0][i] = 0;
16:     st_count = 1;
17:     while( current < st_count )
18:       for ( int i = 0 ; i < seq_count ; i++)
19:         sum_col = 0;
20:       // check the existence of a executable event
21:       if (States[current][i] == 0)
22:         for( int j = 0 ; j < seq_count ; j++ ) // Checks preceding events
23:           if (States[current][j] == 0)
24:             sum_col = sum_col + PrecedenceGraph[j][i];
25:         end for;
26:         if ( sum_col == 0 ) // There is no preceding event
27:           copy_array(States[current], temp_state, seq_count);
28:           temp_state[i] = 1;
29:           next = - 1;
30:           // check the existence of a target state
31:           for( int k = 0 ; k < st_count ; k++ )
32:             if ( equals_array(temp_state, States[k], seq_count) )
33:               next = k;
34:           end for;
35:           if (next == -1)
36:             next = st_count;
37:             copy_array(temp_state, States[st_count], seq_count);
38:             st_count++;
39:           end if
40:           // Add a transition to an automaton
41:           Automata[tr_count][0] = current;
42:           Automata[tr_count][1] = i;
43:           Automata[tr_count][2] = next;
44:           tr_count++;
45:     end if; end if; end for;
46:     current++;
47: end all;
```

First, Algorithm 2 initializes a state, which indicates no event has been executed (14-16). Then, it finds executable events in the initial or current state, which is *States[current]*. An executable event is an event yet to be executed (18-21); it has no preceding events to be executed first and eventually becomes executable in the current state (22-26). For each executable event, the target state is constructed and made executable. Such a target state is inserted as a new state if it does not exist (35-39). Finally, the algorithm adds a transition to an automaton (40-44). The process from line 17 to line 46 is repeated for every new state. Figure 7 represents an equivalence automaton obtained from the precedence graph of Table 2. It accepts all sequences in a class which is equivalent to a given synchronization sequence.

4.3. Modification of Java Multi-thread Programs with Replay Controller

Controller(Replay Controller) permits or blocks the program's threads using an equivalence automaton. The transformed threads obtain and release the program control by communicating with Controller. Figure 8 shows the structure of the modified gas station program. For clarity, we explain Controller and the program transform method in more details.

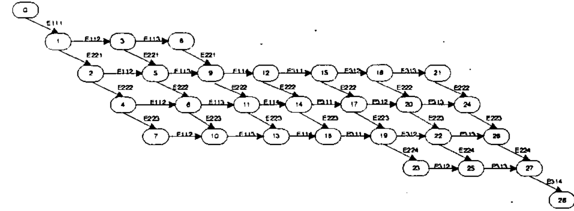

Figure 7. The equivalence automaton generated from a precedence graph in Figure 6

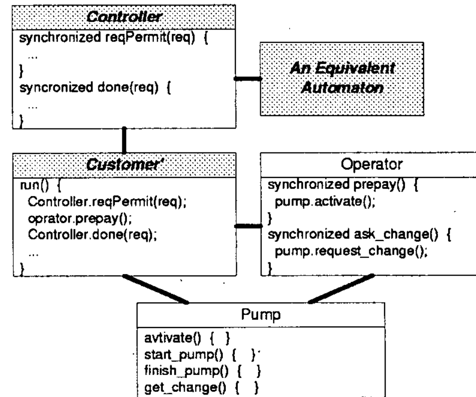

Figure 8. The skeleton structure of the modified gas station example for applying automata-based testing

Controller guides a program to follow a desirable execution path. The Controller in our approach is almost the same as the one used in a classical deterministic testing except that it chooses a desirable execution path by using an equivalence automaton instead of a given sequence. As shown in Figure 9, Controller includes three important methods: *reqPermit(req)*, *done(req)*, and *equals(req)*. The method *reqPermit(req)* permits a thread if the event *req* required by the thread is acceptable by the automaton, otherwise it blocks the thread using the Java method *wait()*. Generally, it is undecidable to determine whether any program can execute a specific sequence because it cannot determine when the program terminates. Therefore, we conclude that the sequences equivalent to the given sequence are infeasible if a specific thread is continuously blocked for a fixed number of times (20 times in Figure 9). The method *done(req)* identifies a new current state in an automaton and activates

```
public class Controller {
   ...
   private static Controller ctrl = new Controller();
   ...
   private Controller()  { ... }
   public synchronized static Controller getController() {
             return ctrl;
   }
   public synchronized void reqPermit(Request req) throws InterruptedException  {
             long cnt = 0;
             boolean tired = false;
             int inautomata = 0;
             while (count != next || (!tired && (inautomata = equals(req)) == 0)) {
                wait(500);
                if (cnt 20)
                   cnt++;
                else
                   tired = true;
             }
             ...
   }
   public synchronized void done(Request req) throws InterruptedException {
             count++;
             notifyAll();
   }
   private int equals(Request req)  {
             int result = 0;          // 0:error,  1:in automata, 2:not in automata
             ...
             if ( (result = at.feasible(curState, curEvent)) != 0)  {
                curState = at.get_nextState(curState, curEvent);
                next++;
             }
             return result;
   }
   ...
}
```

Figure 9. *Controller(Replay Controller)* in the modified program: *Controller* forces a program to follow the desirable execution path

```
public class Customer extends Thread {
   ...
   public void run() {
      try {
        Request req;                                            // *
        Controller ctrl = Controller.getController();           // *
        opr = Operator.get_operator();

        while(!opr.is_available(pumpNumber));

        req = new Request(threadID, pump.getID(), PRE_PAY);     // *
        ctrl.reqPermit(req);                                    // *
        opr.prepay(myNumber, myMoney, pumpNumber, pump);
        ctrl.done(req);                                         // *

        req = new Request(threadID, pump.getID(), START_PUMPING);
        ctrl.reqPermit(req);                                    // *
        pump.start_pumping();
        ctrl.done(req);                                         // *

        req = new Request(threadID, pump.getID(), FINISH_PUMPING);
        ctrl.reqPermit(req);                                    // *
        pump.finish_pumping();
        ctrl.done(req);                                         // *

        req = new Request(threadID, pump.getID(), ASK_CHANGE);
        ctrl.reqPermit(req);                                    // *
        int change = opr.ask_change(myNumber, pumpNumber, pump);
        ctrl.done(req);                                         // *
      }
      catch (InterruptedException e) {};
   }
   ...
}
```

Figure 10. The Transformed *Customer* in order to control the program execution with *Controller*

all blocked threads using the Java method *notifyAll()*. The method *equals(req)* determines whether an automaton accepts the event *req* in a current state. To maintain only one Controller in a program, the constructor of Controller class is declared as **private** one.

To control the execution of the program by communicating with Controller, each *Customer* thread in the gas station program is transformed as in Figure 10, where "//*" denotes the codes added by *Program Transformer*. The thread calls the method *reqPermit(req)* in order to acquire permission from Controller before executing the event *req*, and the method *done(req)* in order to inform Controller that the event *req* is executed after executing the event *req*.

4.4. An Example of Automata-based Testing

To show how the proposed approach works in the case of synchronization reduction, we assume that pumping in pump2 can start as soon as pump1 begins pumping, when both pumps are activated. Figure 11 shows the class *Customer* reflecting the above assumption.

```
public class Customer extends Thread  {
   ...
   public void run() {
      ...
      opName = Controller.PRE_PAY;
      opr.prepay(myNumber, myMoney, pumpNumber, pump);
      ...

      // added code for assumption
      while ( (pump.getID() == 1) &&
            pump.act_state() && otherPump.act_state() &&
            !pump.pump_state() && !otherPump.pump_state() ) ;
      ...
      opName = Controller.START_PUMPING;
      pump.start_pumping();
      ...
   }
   ...
}
```

Figure 11. *Customer* in the gas station program implemented with synchronization reduction

The gas station example under this assumption cannot execute the sequence $[E_{111}, E_{221}, E_{222}, E_{223}, E_{112}, E_{113}, E_{114}, E_{311}, E_{312}, E_{224}, E_{313}, E_{314}]$, because the event E_{222} precedes the event E_{112}. In this situation, classical deterministic testing would conclude that the given sequence is infeasible without further exploring other sequences. However, in the automata-based testing, it is obvious that the program implements the sequence because an alternative sequence showing the same behavior with the given sequence, i.e., the sequence in Table 3 is allowed. Note that, in automata-based testing, a sequence is not necessarily implemented although it is allowed, However, it is guaranteed that some equivalent sequences are implemented.

In Table 3, *"Required Events"* indicates nondeterministic events in the thread *Customer* which are required to obtain permission from *Controller*. *"Re-required Events"* indicate the events in blocked threads are required to obtain the permission. In Table 3, the first required event E_{111} is accepted by the automaton and obtains permission. But the event E_{222} which is not accepted by the automaton is blocked until its turn. Consequently, the gas station program executes the sequence $[E_{111}, E_{221}, E_{112}, E_{222}, E_{112}, E_{223}, E_{114}, E_{311}, E_{224}, E_{312}, E_{313}, E_{314}]$. This sequence

Required Event	Re-required Event	Acceptance in Automaton	Blocked Event	Executed Event
E_{111}	-	*Yes*	-	E_{111}
E_{221}	-	*Yes*	-	E_{221}
E_{222}	-	No	E_{222}	-
E_{112}	-	*Yes*	-	E_{112}
-	E_{222}	*Yes*	-	E_{222}
E_{113}	-	*Yes*	-	E_{113}
E_{311}	-	No	E_{311}	-
E_{223}	-	*Yes*	E_{311}	E_{223}
-	E_{311}	No	E_{311}	-
E_{224}	-	No	E_{311}, E_{224}	-
E_{114}	-	*Yes*	-	E_{114}
-	E_{311}	*Yes*	E_{224}	E_{311}
	E_{224}	*Yes*	-	E_{224}
E_{312}	-	*Yes*	-	E_{312}
E_{313}	-	*Yes*	-	E_{313}
E_{314}	-	*Yes*	-	E_{314}

Table 3. The executed sequence by applying automata-based testing to the gas station program implemented with synchronization reduction

which is different from the given sequence is an equivalent sequence showing the same behavior. Therefore, we can conclude that the gas station program is implemented properly for the given sequence.

5. Conclusion and Future Works

In this paper, we proposed the automata-based testing environment for Java multi-thread programs and we had designed and implemented three component of the environment - *Automata Generator*, *Replay Controller*, and *Program Transformer*. *Automata Generator* generates an equivalence automaton from a given sequence and a dependency table, *Replay Controller* guides Java multi-thread programs to follow one of sequences equivalent to the given sequence based on the equivalence automaton, and *Program Transformer* transforms programs into modified ones for executing them according to the sequence allowed by Replay Controller.

While classical deterministic testing checks that a given sequence is just feasible, automata-based testing checks that one of sequences equivalent to a given sequence is feasible. Therefore, the automata-based testing is proper to test concurrent programs that implement their specification partially. We showed how the automata-based testing worked in the proposed environment through the gas station example.

Our future work is to design the *Dependency Extractor*. In this paper, it is assumed that a test sequence and a dependency table are given. So we need the method for extracting such information from specifications. It has three issues: What dependencies exist in specifications, how extracts such dependencies, and what a test criterion is used to generate test sequences. Finally, we will implement the automata-based testing environment consisting of four components: *Dependency Extractor*, *Automata Generator*, *Replay Controller*, and *Program Transformer*.

References

[1] A. Bechini, J. Cutajar, C. A. P. V. Bochmann, and A. Petrenko. A tool for testing of parallel and distributed programs in message-passing environments. In *Proceedings of the 9th Mediterranean Electrotechnical Conference 1998 (MELECONE98)*, volume 2, pages 1308–1312, 1998.

[2] G. V. Bochmann and A. Petrenko. Protocol testing: Review of methods and relevance for software testing. In *Proceedings of the International Symphosium on Software Testing and Analysis*, pages 109–124, 1994.

[3] A. F. Brindle, R. N. Taylor, and D. F. Martin. A debugger for ada tasking. *IEEE Transaction on Software Engineering*, 15(3):293–304, March 1989.

[4] X. Cai and J. Chen. Control of nondeterminism in testing distributed multithreaded programs. In *Proceedings of the 1st Asia-Pacific Conference on Quality Software*, pages 29–38, 2000.

[5] R. H. Carver and K. C. Tai. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66–74, March 1991.

[6] R. H. Carver and K. C. Tai. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transaction on Software Engineering*, 24(6):471–490, June 1998.

[7] I. S. Chung and B. M. Kim. Yet another approach to deterministic execution testing for distributed programs. In *Proceedings of the IASTED International Conference on Software Enginerring and Applications*, pages 55–60, 2000.

[8] I. S. Chung, B. M. Kim, and H. S. Kim. A new approach to deterministic execution testing for concurrent programs. In *Proceedings of the International Workshop on Distributed System Validation and Verification*, pages E59–E66, 2000.

[9] I. S. Chung, H. S. Kim, H. S. Bae, Y. R. Kwon, and B. S. Lee. Testing of concurrent programs based on message sequence charts. In *Proceedings of the International Symphosium on Software Engineering for Parallel and DisTributed Systems (PDSE99)*, pages 72–82, 1999.

[10] P. A. Emrath, S. Ghosh, and D. A. Padua. Detecting nondeterminacy in parallel programs. *IEEE Software*, 9(1):69–77, January 1992.

[11] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transaction on Computers*, C-36(4):471–482, 1987.

[12] H. W. Sohn, D. C. Kung, and P. Hsia. State-based reproducible testing for corba applications. In *Proceedings of the International Symphosium on Software Engineering for Parallel and Distributed Systems (PDSE00)*, pages 29–38, 2000.

[13] K. C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Transaction on Software Engineering*, 17(1):45–63, January 1991.

[14] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Transaction on Software Engineering*, 18(3):206–215, March 1992.