Code and Model Based Test Sequence Generation for Multithreaded Programs

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Submitted By
Vipin Verma
(801231031)

Under the supervision of:
Mr. Vinay Arora
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2014

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*Code and Model Based Test Sequence Generation for Multithreaded Programs*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mr. Vinay Arora* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Vipin Verma

(Vipin Verma)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Vinay

(Mr. Vinay Arora)
Assistant Professor,
CSE Department

Countersigned by

(Dr. Deepak Garg)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

First of all, I would like to thank the Almighty, who has always guided me to work on the right path of the life. Due to mercy of God, it has been possible for me to reach so far.

This work would not have been possible without the encouragement and valuable guidance of my supervisor **Mr. Vinay Arora**, Assistant Professor, Thapar University, Patiala. I thank my supervisor for his time, patience, discussions and valuable comments.

I am equally grateful to **Dr. Deepak Garg**, Associate Professor and Head, Computer Science and Engineering Department, for motivation and inspiration that triggered me for the thesis work. I am also thankful to the entire faculty and staff members of Computer Science and Engineering Department for their direct-indirect help, cooperation, love and affection, which made my stay at Thapar University memorable.

I will be failing in my duty if I don't express my gratitude to **Dr. S. K. Mohapatra,** Senior Professor and Dean of Academic Affairs the University, for making provisions of infrastructure such as library facilities, computer labs equipped with net facilities, immensely useful for the learners to equip themselves with the latest in the field.

Last but not least, I would like to thank my parents for their wonderful love and encouragement, without their blessings none of this would have been possible. I would also like to thank my sister for her constant support throughout the thesis.

Abstract

Test sequence generation is considered as an important sub-phase of testing and its automation makes the process of testing easier. Test sequence generation through code involve a flow graph viz. control flow graph (CFG), concurrent control flow graph (CCFG), event graph *etc*. as an intermediate form. In this thesis, work in the area of test sequence generation using code and UML is analyzed in a systematic way. A novel approach has been proposed to generate test sequences for Java7 fork/join concept. Interference dependence with all path and all node coverage criteria metrics has been used for generating test sequences.

Almost all the previously presented approaches for test sequence generation from UML need a flow graph as an intermediate representation. An approach has been proposed for test sequence generation from UML activity diagram without any intermediate graph. Here, an activity diagram is first converted to equivalent XML file, which is then traversed to find test sequences by applying a combined form of depth first search (DFS) and breadth first search (BFS) with an optimization. Infeasible test sequences and additional transformation costs are avoided in the proposed approach.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

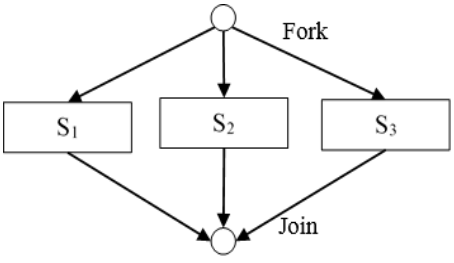
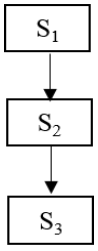
This chapter describes basic concepts related to concurrency, software testing, testing of concurrent programs, methods of generating test cases for concurrent programs and Java7 fork/join framework. An example is also illustrated for explaining the problem of inconsistency in concurrent programs.

1.1 Concurrency

Two or more processes are concurrent if there is an illusion that these processes are being carried out in parallel. Concurrency can be achieved for multi-processor systems as well as single processor systems. In an operating system, concurrent processes are obtained by interleaving the operations of processes on central processing unit (CPU) [1]. The term concurrency is used when two or more processes collaborate with each other to attain a common outcome [2].

In case of sequential execution of programs, only one program is executed at a particular time in a sequential manner. When one process finishes its execution, only then the next process gets the processor. Table 1.1 presents a differentiation among sequential and concurrent processes.

Table 1.1 Comparison of sequential and concurrent processes [3]

S. No.	Concurrent Process	Sequential Process
1.	Operation of one process gets started before the operation of other process completes.	Operations are carried out strictly one at a time.
2.	Start and end nodes are represented by fork and join.	There is no such concept of start and end node.
3.	 Figure 1.1 Execution flow in a concurrent process [3]	 Figure 1.2 Execution flow in a sequential process

1.2 Software Testing

Testing is a systematic process of finding out errors in a program or a software under test (SUT). Testing carries prime importance in the software development process that is why it is a different phase in software development life cycle (SDLC). Many software firms say that testing process requires approximately 40% of the efforts whereas the other 60% is utilized in the development of any software [4].

According to IEEE Standard 829-1983 [5], in the process of testing, a software item is analyzed for detecting the alterations between the current and desired conditions (i.e. bugs) and for assessing the features.

According to G. J. Myers [6], testing is the process in which the program is executed with the intent of uncovering the errors.

1.3 Testing of Concurrent Programs

Concurrent programs possess a non-deterministic nature of execution due to which testing of concurrent programs is a difficult task. Concurrent programs can also suffer with a problem of inconsistency as illustrated in the following example.

A problem with concurrency: Consider two customers C_1 and C_2 have a shared bank account with a balance of ₹ 200 [7].

The operations, performed on the shared bank account, are as follows: -

i. Customer C_1 tries to withdraw ₹ 50 from the account.
   a. He reads balance as ₹ 200.
   b. And tries to set new balance as ₹ 150.

ii. Customer C_2 tries to withdraw ₹ 50 before the previous transaction gets completed.
   a. He reads balance as ₹ 200.
   b. He also tries to set new balance as ₹ 150.

iii. So, final balance in the shared account is ₹ 150 instead of ₹ 100.

From this discussion, it is clear that concurrency leads to unexpected behavior and results of the program. The shared account shows final balance as ₹ 150 instead of ₹ 100. Users can access the shared account in a random manner, which can cause problem of inconsistency. Figure 1.3 illustrates this problem in detail for its better understanding.

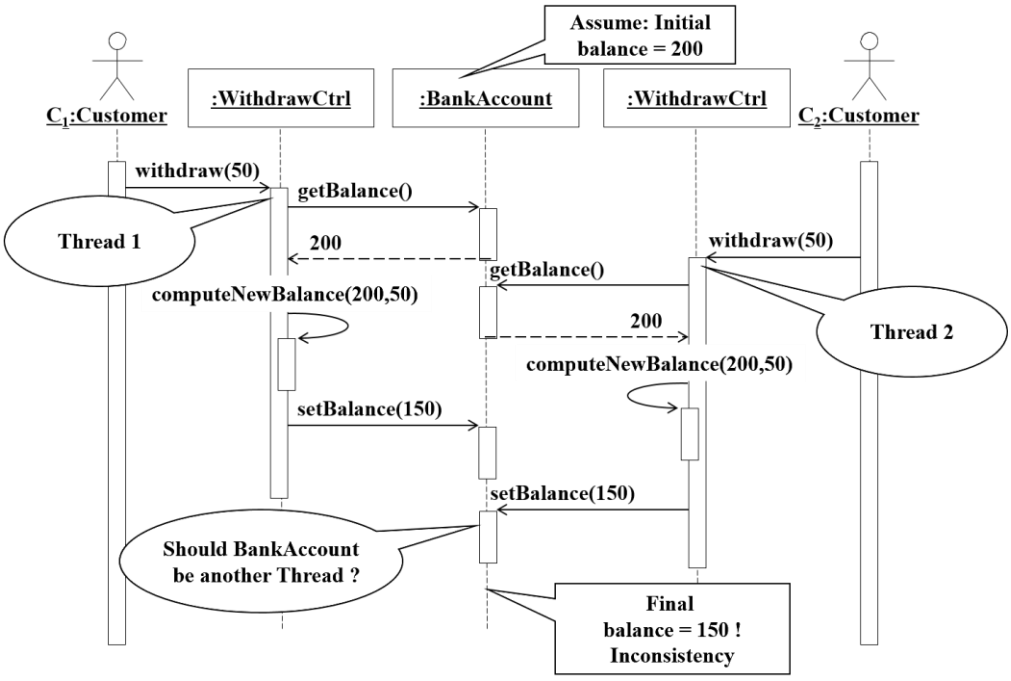


Figure 1.3 Problem of inconsistency in concurrent system [7]

Solution to the inconsistency problem: For solving the problem encountered in above example, the customers C_1 and C_2 should access the shared account in a synchronized manner as presented in the Figure 1.4.

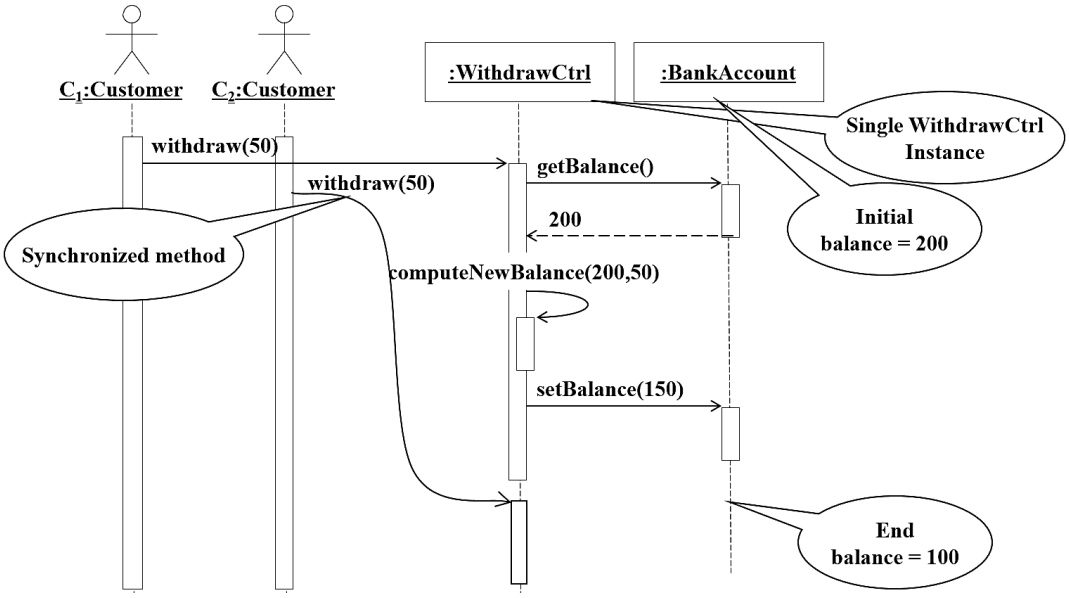


Figure 1.4 Solution to inconsistency [7]

From the above discussion, it is clear that concurrent programs are more prone to errors like inconsistency, deadlock *etc.* Also, testing and test case generation are very crucial activities to get a quality product.

1.4 Test Case Generation for Concurrent Programs

According to IEEE Standard 160.12-1990 [8], a test case is defined as a collection of inputs, execution conditions and predicted outputs for analyzing programs. Test cases also check for conformance of programs to the requirements gathered at starting phase of software development. Test cases for an example program, are shown in Table 1.2 to find out whether a triangle is equilateral, isosceles or scalene [6].

Table 1.2 Example test cases for a triangle

Test Input	**Execution Condition**	**Anticipated Output**
[4,2,3]	Sum of any two sides of triangle must be greater than third side.	Scalene triangle
[3,3,5]		Isosceles triangle
[6,6,6]		Equilateral triangle

Some basic introduction about test case generation using UML and using code is presented in this section.

1.4.1 Test Case Generation Using UML (Unified Modelling Language)

UML [9] is a standard modelling language for designing the blueprints for software systems. UML can be used to visualize the systems in the form of models and to specify the models unambiguously. These models are directly mapped to any programming language to construct the code. UML also facilitates easy documentation of various aspects of any software system. UML diagrams that can represent concurrency are described in this sub-section.

a) **Activity Diagram**

It represents the behavior of a software system in the form of activities. It also illustrates flow of activities in a given scenario from start to end with a possible order of execution. Activity diagrams show sequential, branch, loop and simultaneously executable

sections of any system. Concurrency is shown by using fork and join symbols and branch node is depicted by diamond symbol as demonstrated in Figure 1.5.

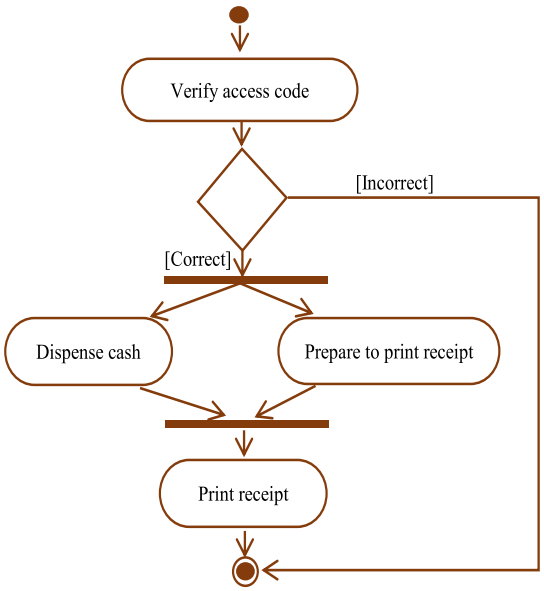


Figure 1.5 Activity diagram for ATM [10]

b) **Statechart Diagram**

Statechart diagram shows the flow of control from one state to another and deals with dynamic view of a software system. Transition from one state to another represents events and is guided by the guard conditions. It can be used to model the behavior of an interface or a class. Fork and join are used for representing concurrency as shown in Figure 1.6.

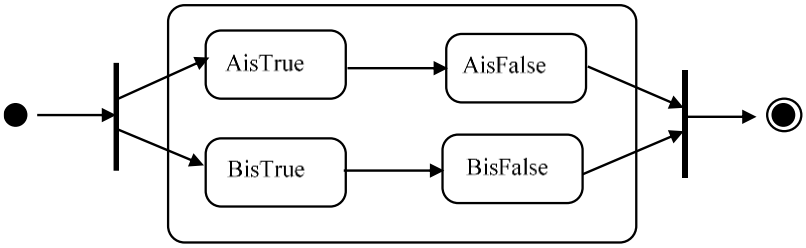


Figure 1.6 Statechart diagram with concurrency [11]

c) **Sequence Diagram**

It is an arrangement of objects and the time ordering of messages interchanged among the set of objects. These objects can be of both types *i.e.* named and anonymous. Lifelines are used to represent the time for which an object communicates with other objects. An example sequence diagram is shown in Figure 1.8 for the code showing concurrency as in Figure 1.7.

```
class cell {
private int value;
public void swap(cell other){
synchronized(this){
synchronized(other){
int newValue = other.value;
other.value = value;
value = newValue;} }
}
}
```

Figure 1.7 Example code for swapping [12]

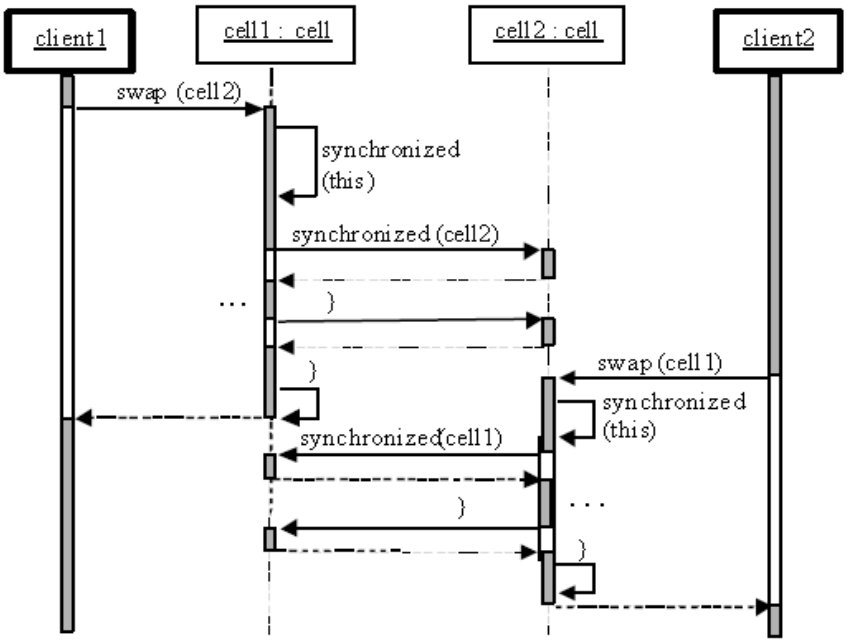


Figure 1.8 Sequence diagram showing concurrency [12]

d) **Collaboration Diagram**:

Collaboration diagram shows organization of various objects with send and receive message passing among those objects. Collaboration diagrams and sequence diagrams are isomorphic in nature i.e. these two diagrams can be used interchangeably. Collaboration and sequence diagram can represent the same system in the same manner. However there is no concept of lifelines in collaboration diagrams. Therefore, when there is need of lifelines i.e. the time up to which the objects are in communication with one another, sequence diagrams are preferred. As represented in Figure 1.9, concurrency is shown with the help of parallel messages like A.1, A.2, A.3 *etc.*

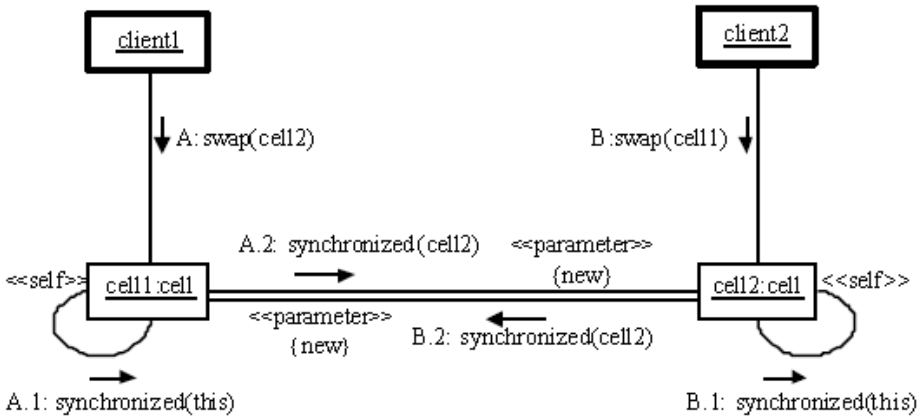


Figure 1.9 Collaboration diagram showing concurrency [12]

1.4.2 Test Case Generation Using Code

Test cases/sequences can also be generated by using the code itself. One approach for test case generation is to transform the code into a graph first, e.g. CFG (control flow graph) and then traverse the graph for some adequacy criteria.

a) **CFG (Control Flow Graph):**

A control flow graph is a directed graph in which nodes represent the basic blocks and edges between any two basic blocks shows the flow of control. A basic block is defined as a sequence of instructions executed one after the other and has one entry and one exit point [13].

b) **Dependence Graphs:**

The variables and statements of a program can be represented in the form of data, control, interference and ready dependence. Various dependencies that exist in any program are as follows:

Data Dependence: In a sequential program, a statement *m* is data dependent on statement *n*, if *n* defines some variable and statement *m* uses the same variable along a control-flow path [14].

Control Dependence: A statement *m* is control dependent on statement *n*, if the control can pass to *m* if and only if the control passes from statement *n*. In other words, statement *n* controls the execution of statement *m* [15].

Interference Dependence: It is a special type of data dependence among the instructions of a concurrent program. Say, a variable *x* of any object is being written by a thread T_1 at statement *n* and it is read by another thread T_2 at a statement *m*. In such a case, node *m* is interference dependent on node *n* [14]. Figure 1.10 represents interference dependence and depicts the control flow.

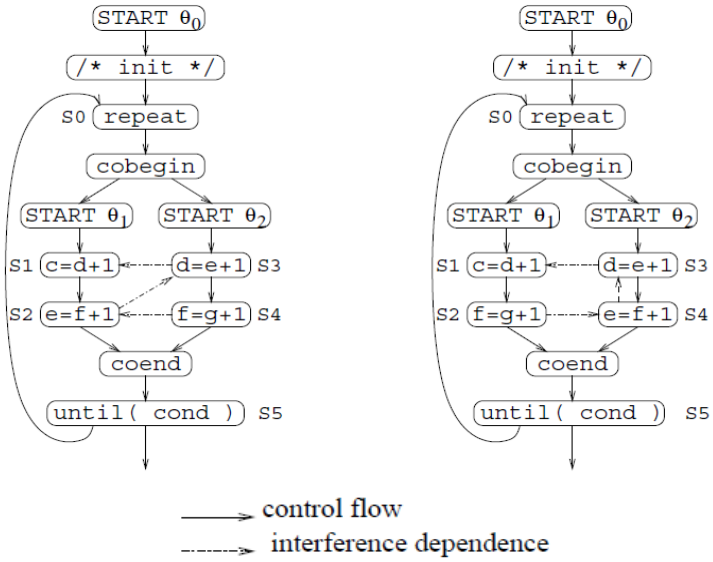


Figure 1.10 Control and interference dependence [16]

Ready Dependence: It is a type of control dependence, which is found in concurrent programs. A statement *m* is said to be ready dependent on statement *n*, if the execution of *m* can be delayed for infinite time due to the fact that *n* could not be successfully executed [14]. For example, both *m* and *n* are in same thread and *n* is a wait() statement which, is never notified.

c) Test Coverage

According to IEEE Std 610.12-1990 [8], test coverage of a test case is defined as the degree of addressing all the stated requirements of a given system.

Statement Coverage: It is defined by a ratio $S_c / (S_e - S_i)$ [17], where S_c is the count of statements covered by the test case. S_i is the count of unreachable statements and S_e is the total count of statement. For a test case to be adequate according to statement coverage, the ratio should be 1.

Decision Coverage: It is defined as the ratio $D_c / (D_e - D_i)$ [17], where D_c is the count of decisions covered. D_e is the total count of decisions in the system and D_i is the count

of unreachable decisions. A test case is said to be adequate according to decision coverage if this ratio is 1.

1.5 Java Fork/Join

In Java7, a new concept was launched called Java fork/join [18] framework. It is the implementation of the '*java.util.concurrent.ExecutorService*' interface. It is introduced to divide the work in smaller parts recursively and those parts can be handled by multiple processors simultaneously. Therefore, it allows to get the most out of multi-processor systems and as a result the performanace will improve. This implementation uses the work-stealing algorithm i.e. whenever some threads don't have any work to do, the work can be stolen from other busy threads. The class '*java.util.concurrent.ForkJoinPool*' uses the work-stealing algorithm and can execute various '*java.util.concurrent.ForkJoinTask*' processes. Java7 fork/join facility is appreciably useful for making proper use of the powers of multi processor systems.

Figure 1.11 shows the basic structure of code that must be written to make use of Java7 fork/join utility. Divide and conquer approach is used here, if the work to be done is large.

```
if(the work to be done is small enough)
   do the work
else
   divide the work in two pieces
   invoke the two pieces, wait for result
```

Figure 1.11 Java7 fork/join framework [18]

Chapter 2
Literature Survey

Test cases can mainly be generated by using code or a model like UML (Unified Modelling Language). UML is a standard language for specifying the design of a software system. Another advantage of using UML is its capability to provide different diagrams for representing the various views of any software system. Moreover, the process of test case generation using UML is easy to automate [19]. In the present era of multi-processors, concurrent programs are widely used. Also, concurrent programs are very much exposed to errors like inconsistency, deadlock *etc*., which makes testing a critical activity. This chapter presents different tools and techniques for test case generation for concurrent programs. This chapter also lists some of the advantages, disadvantages and future works related to the discussed approaches. Test case generation using code can be done using event graphs and event interaction graphs. Work related to test case generation for BPEL4WS (business process execution language for web services) processes is also described in this section. In the end of this chapter, various commercial, open source and freely available tools are summarized in Table 2.1.

2.1 Test Case Generation from UML Models

This section presents the literature survey for generating the test cases from UML activity diagrams, sequence diagrams and statechart diagrams.

2.1.1 Generating Test Cases from Activity Diagrams

C. Mingsong *et al.* [20] presented a technique to generate automatic test cases and illustrated a tool AGTCG (activity graph test case generator). Test cases are generated at random and the execution traces are compared with the activity diagram to get a reduced set of test cases. This is the first approach in this area to generate the test cases fully automatically. The authors used improved depth first search (DFS) algorithm to find out the paths and instrumentation algorithm for collecting all the execution traces. The approach is able to check the consistency of execution traces and cost consumed is less with a decreased probability of errors.

H. Kim *et al.* [21] converted the activity diagram into IOAD (input output explicit activity diagram) in which the inputs and outputs are taken under consideration. This intermediate form is then transformed into a directed graph from which the test cases are derived. The researchers used depth first search algorithm for calculating the paths of activity diagram. The problem of state space explosion is also handled as fewer test cases are generated. Cost and time are saved with no compromises to quality. However, the approach is still to be automated and generalized for various coverage criterion. Figure 2.1 shows an IOAD for the activity of order processing.

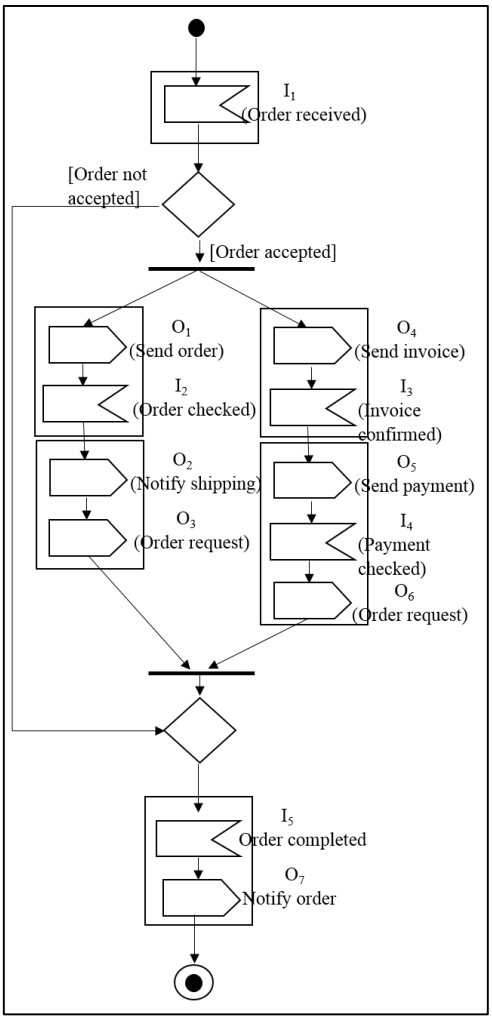


Figure 2.1 IOAD for order processing activity [21]

D. Kundu and D. Samanta [22] converted the activity diagram into another intermediate representation i.e. activity graph and test cases are then generated on the basis of path coverage criteria. This approach is more effective than other approaches. However, the work for multiple use cases' activity diagrams is still to be considered. An example activity graph, corresponding to the activity diagram for use case of order cancellation as in Figure 2.2, is presented in Figure 2.3.

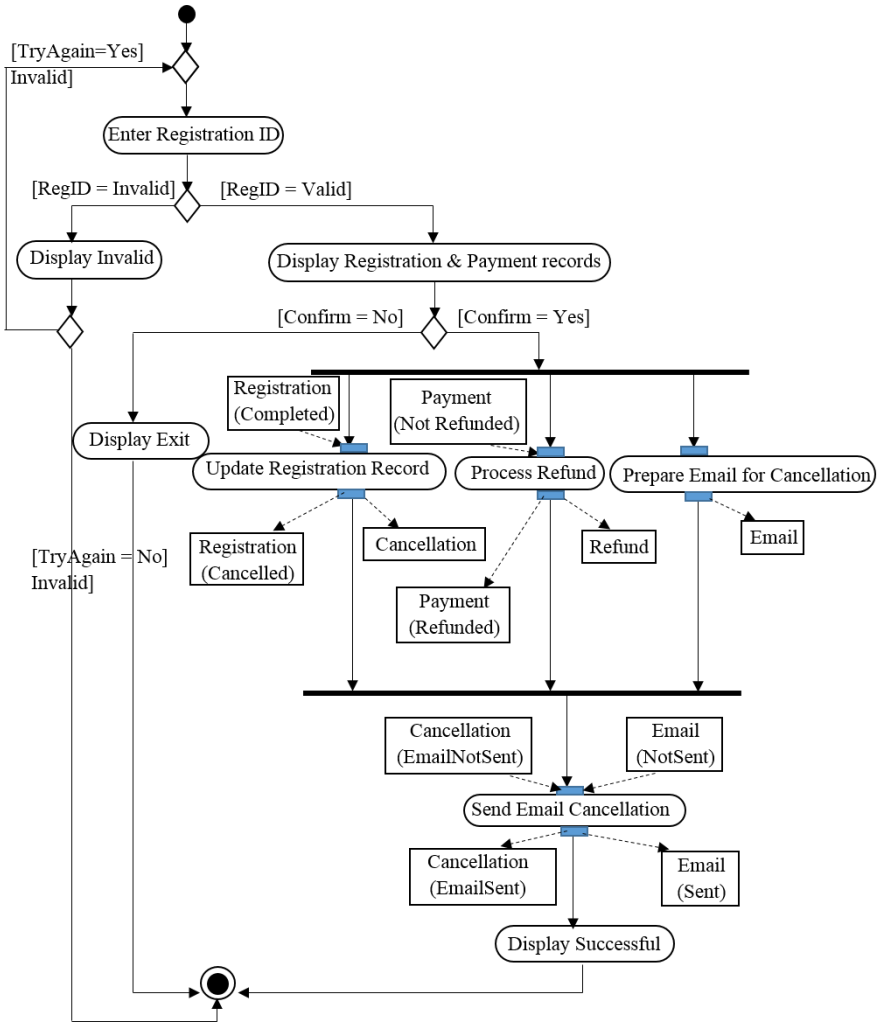


Figure 2.2 Activity diagram for order cancellation [22]

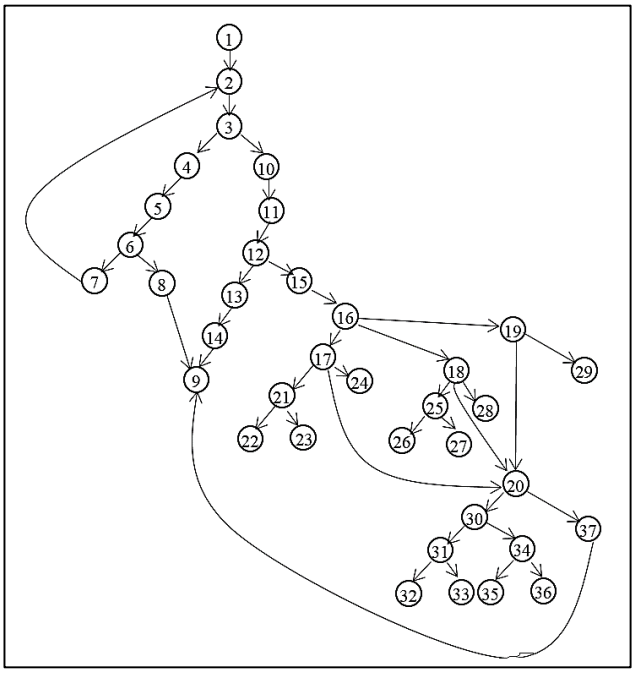


Figure 2.3 Example activity graph for order cancellation [22]

C. Sun [23] also converted the activity diagram into BET (binary extended AND_OR tree) which is then traversed using depth-first traversal to generate the test scenarios. Better resource management and lesser costs due to early fault detection are the benefits of this approach. A tool TCaseUML is also presented by the author. Figure 2.4 represents an example BET.

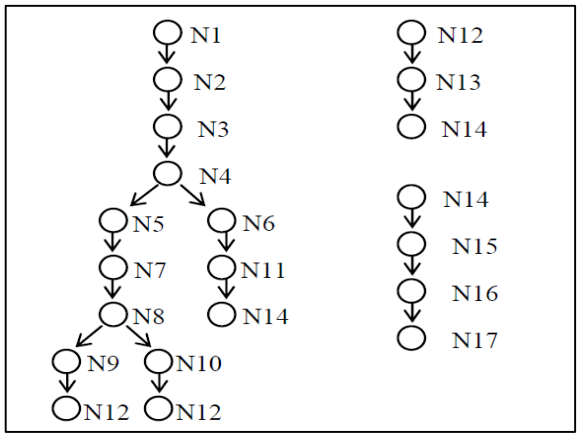


Figure 2.4 Example of binary extended AND_OR tree [23]

M. Khandai *et al.* [19] presented a survey on test case generation from UML Models and stated two main approaches for the same. In first approach, activity diagram is converted to activity graph and by traversing that test cases are generated. And in second approach, activity diagram is converted to an intermediate form like control flow graph using some transformation rules and then test cases are generated.

B. Lei *et al.* [24] presented a tool named as tof4j (testing of concurrency for Java program) and further extended the activity diagram that is traversed on the basis of path analysis technique. The tool also detected data race and inconsistency conditions using data race detection and inconsistency detection algorithms . The cost of run time computation is reduced as testing tool analyzes the execution traces offline. However, this is not applicable to legacy systems because the models are not present.

X. Fan *et al.* [25] decomposed activity diagram into sub-activity diagrams by dividing the activities into sub-activities. Then an ADCT (activity diagram composition tree) is constructed for which test cases are first generated and then round-robin strategy is applied to generate the test cases for the whole system. Testing efficiency is increased in this approach and lesser number of test cases are generated. Figure 2.5 shows an example activity diagram composition tree where AD is the highest level activity diagram and AD_1 and AD_2 etc are sub-activities, represented as parent and children.

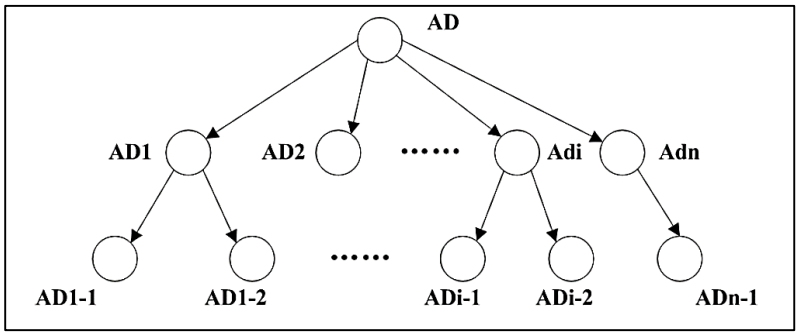


Figure 2.5 Example activity diagram composition tree [25]

M. Chen *et al.* [26] presented a prototype tool for automatically generating the test cases. In this approach, there are some transformation rules for automatic conversion of UML models to formal specification. Many good model checking techniques are combined for improving the process of test case generation. This tool takes 3 inputs namely UML activity diagram, type definition and context information. The advantages of this approach include lesser time for test case generation and reduced number of test cases.

2.1.2 Generating Test Case from Sequence Diagrams

S. K. Swain *et al.* [27] presented a semi-automatic tool ComTest (comprehensive test) to generate the test cases. The sequence diagram is first converted to concurrent control flow graph (CCFG) and then test cases are generated on the basis of full predicate coverage criteria. Also the operational and synchronization faults are detected. This approach can be used for integration and regression testing. However, the work of test data generation is to be done in future. An example CCFG is shown in Figure 2.6.

M. Khandai *et al.* [28] presented a technique to convert the sequence diagram into concurrent composite graph (CCG), an intermediate representation which is then traversed to generate the test cases. The algorithm used to traverse the graph is known as generate message sequence path algorithm. Deadlock and synchronization issues are handled and problem of test case explosion is also avoided. However, test case generation for distributed system and test case prioritization for reducing the regression testing cost are still to be done in the future. Figure 2.7 shows concurrent composite graph for an ATM system.

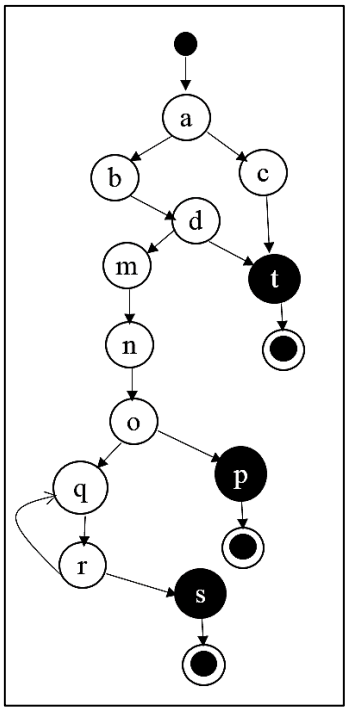


Figure 2.6 Example CCFG for ATM [27]

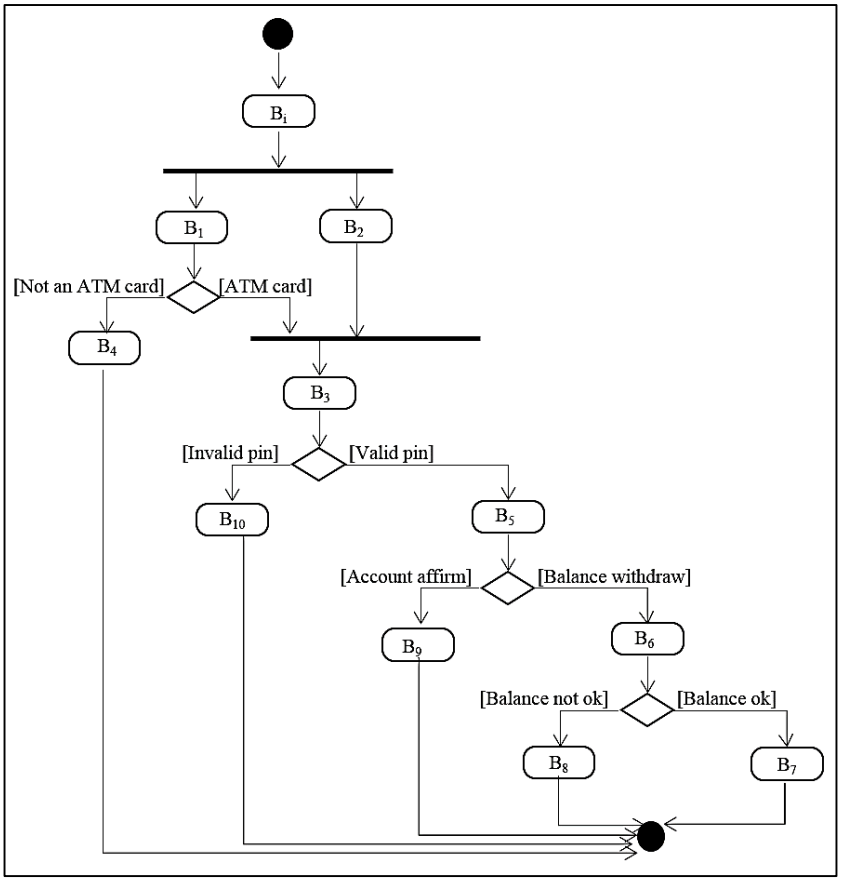


Figure 2.7 CCG for ATM [28]

M. Shirole and R. Kumar [10] presented an approach in which the sequence diagram is first converted to an activity diagram using some rules. An algorithm named as

concurrent queue search (CQS) is presented to traverse the activity diagram and hence to generate the test sequences. This algorithm is better than depth first search (DFS) and breadth first search (BFS). This approach is also able to find the data safety errors that might occur in concurrent programs.

2.1.3 Generating Test Cases from Statechart Diagrams

Y. G. Kim *et al.* [29] converted a statechart diagram into an EFSM (extended finite state machine) which is then converted to a control glow graph (CFG). CFG thus created is analysed on the basis of control flow and data flow to generate the test cases. However, this approach is not automatic and the inter-relationships among classes are not considered.

P. Chevalley and P. T. Fosse [30] studied the efficiency of test case generation from UML statechart diagram satisfying the transition coverage criteria. Also, the authors took the research version of avionics system in Java i.e. flight guidance system as the experimental setup. Test cases are evaluated based upon mutation analysis. In future, more number of mutants are to be checked. The process used to generate test cases is shown in Figure 2.8.

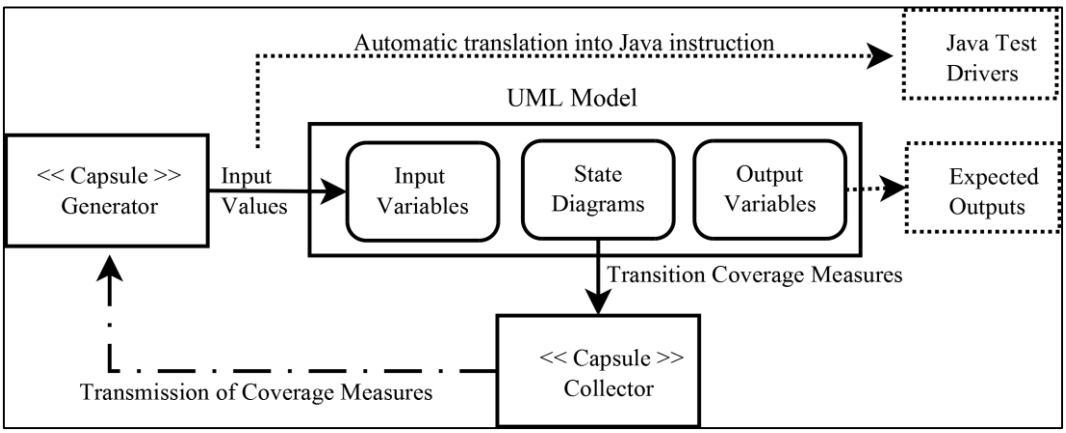


Figure 2.8 Test generation process [30]

S. K. Kim *et al.* [31] extended the UML state machines to capture the information about the number of blocked and waiting threads on an object. This extended UML state machine is given as input to a model checker like SAL (symbolic analysis laboratory) model checker to generate the sequences. To convert those sequences into test sequences, a testing tool like ConAn is used. This testing technique is systematic and improved than other existing approaches.

L. C. Briand *et al.* [32] automated the process of test case generation from the UML statecharts. Firstly, the paths are derived according to a coverage criteria and those paths are taken into consideration to develop the fully specified test cases. The researchers developed a prototype tool called contract-based constraint derivation tool (CBCDTool) for the same purpose. However, this approach is slow in the worst case and the algorithm does not give results if the constraints are not satisfied.

P. Samuel *et al.* [33] presented a novel method for automatic test case generation from UML statechart diagrams by making use of control and data flow. The authors presented a tool called UTG (UML test case generator). Test cases are generated by traversing the state machine graph and checking conditions at each transition. These test cases satisfy the transition path coverage criteria. This approach is fully automatic and can be applied to both class and cluster level. However, globally optimized solution is not given by this approach. Also, other UML models are to be investigated. Figure 2.9 shows the activity diagram for the methodology for test case generation used by the researchers in paper [33].

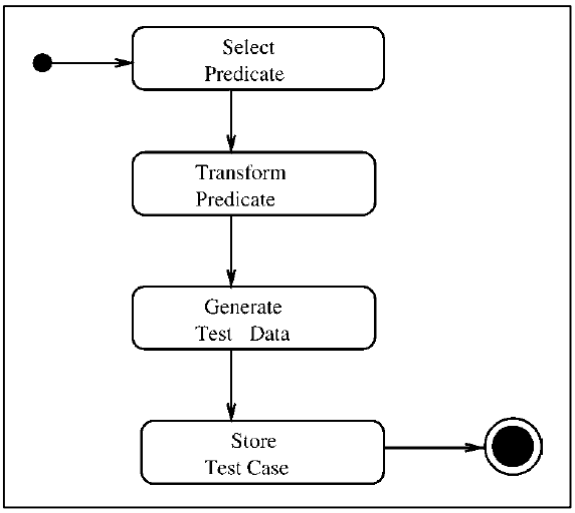


Figure 2.9 Method of test case generation [33]

D. Patnaik *et al.* [34] converted the statechart diagram into an event tree and then an event graph is generated which is traversed to generate the test cases. Backtracking algorithm is used to generate the test suite and find whether there is a deadlock in the system. Figure 2.10 shows an event graph to represent a deadlock situation. In this scenario,customer details are accessed at the same time, hence deadlock condition occurs.

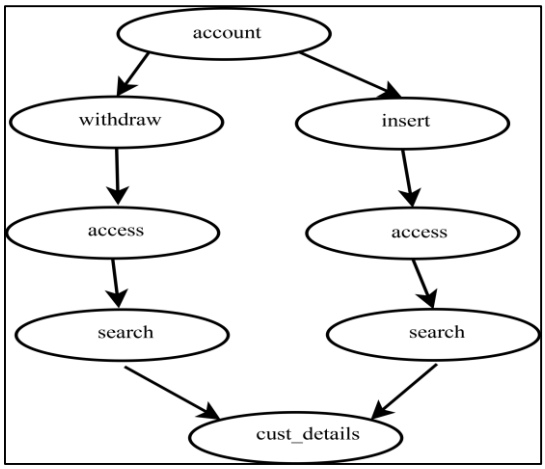


Figure 2.10 A deadlock represented using event graph [34]

M. Aggarwal and S. Sabharwal [35] presented a survey based on generating the test cases using state machines by comparing various techniques. It was concluded that guard condition is a necessary condition to generate feasible test cases.

2.2 Test Case Generation from Code

Test cases can also be generated from code by using an intermediate graphical representation, which is generally a control flow graph. This section presents work related to test case generation by making use of event graphs and test case generation techniques for BPEL4WS i.e. business process execution language for web services.

2.2.1 Generating Test Cases from Event Graphs

An event graph is a control flow graph of one unit of a concurrent program. Event interaction graph (EIAG) [36] is a graph that represents the behavior of a concurrent program which has the events and the interactions as the main components. Interactions are either synchronization, communications or wait. EIAGs depend on the source code and co-paths (cooperated paths) on EIAG provide the test cases. An example of event interaction graph for dining-philosopher problem is shown in Figure 2.11.

T. Katayama *et al.* [36] generated the co-paths automatically and are also able to detect unreachable statements and communication errors in testing. The problem of test case explosion is avoided, however, the tool TCGen (test case generator) is to be enhanced for communication and wait constructs.

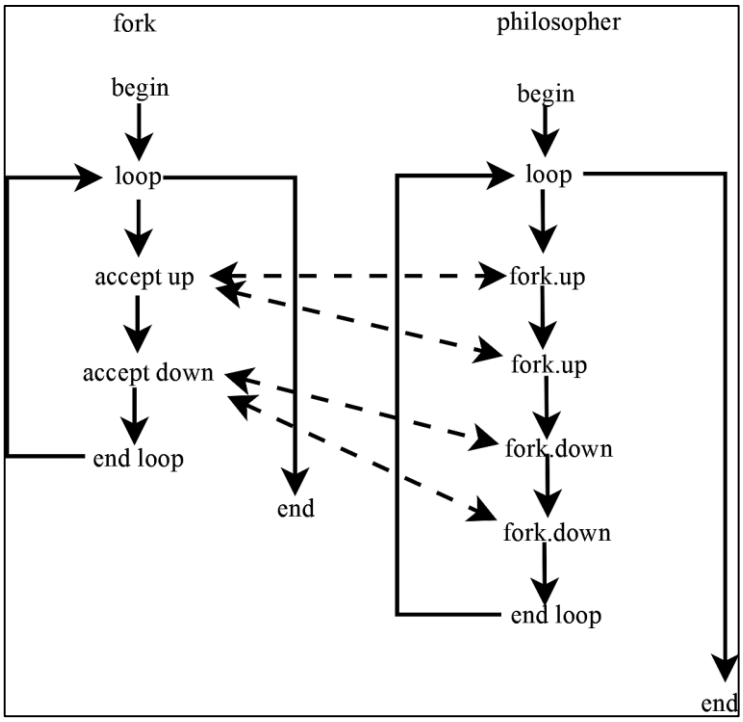


Figure 2.11 EIAG for dining-philosopher problem [37]

Later T. Katayama *et al.* [37] improved the method of test case generation by enhancing the event interaction graph and presented more accurate behavior for concurrent programs. This approach is more reliable for detecting complete communication errors, however, number of test cases are increased.

Afterwards T. Katayama *et al.* [38] used the enhanced EIAG and described the tool TCgen (test case generator) for Ada programs. The co-paths are able to detect the complete communication errors, unreachable statements and some deadlocks. However, TCgen is not able to handle exceptions.

T. Katayama *et al.* [39] then used the interaction sequence testing criteria (ISTC) for generating the co-paths. These test cases are able to find out unreachable statements, some communication errors and deadlock also. However, the tool TCgen is still to be extended for various other Ada programs.

X. Bao *et al.* [40] generated the test cases for concurrent programs based upon the event graphs. Test cases, also known as sub-event graphs, are generated by the analysis of event graph. The advantages of this approach include the feasibility of all the test cases and no problem of state explosion. However, large number of test cases are generated in this approach.

2.2.2 Generating Test Cases for BPEL4WS (Business Process Execution Language for Web Services)

BPEL is standard language just like XML which is used for modelling the web services. It is a semi-formal language comprising of various complex features viz. concurrency, dead-path elimination *etc.* This language is based on the construct *flow* to represent synchronization and concurrency [41].

Y. Yuan *et al.* [42] created a BPEL flow graph (BFG) which is nothing but an extension of control flow graph. Then the BFG is traversed using a constraint solving method and the test paths are combined for generating the test cases. However, this approach is to be extended for exceptions and some other advanced features of BPEL. The process used to generate test cases is shown in Figure 2.12.

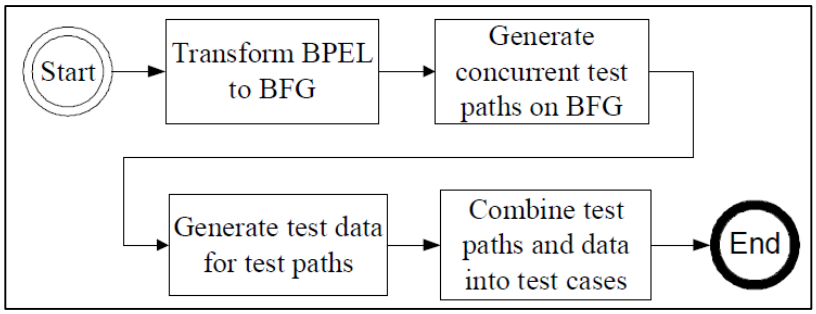


Figure 2.12 Approach for test case generation [42]

J. Yan *et al.* [41] created an extended control flow graph (XCFG) to express a BPEL program. Then all the sequential test paths are generated. On combining the sequential test paths, the concurrent test paths are generated. A constraint solver BoNus is used for solving the constraints and generating the feasible test cases. The generated test cases are kept lesser in number by using appropriate test criteria and this work is also useful in regression testing. This is the first paper for describing test case generation for unit test framework of BPEL. An example XCFG is shown in Figure 2.13.

Y. Zheng *et al.* [43] used SPIN (Simple PROMELA (process meta language)) model checker as test generation engine. For control flow testing, state and transition coverage are used and for data flow testing, all-du (def-use)-path coverage is used. The generated test cases are then executed on JUnit test execution engine. However, this approach is to be enhanced for scalable models.

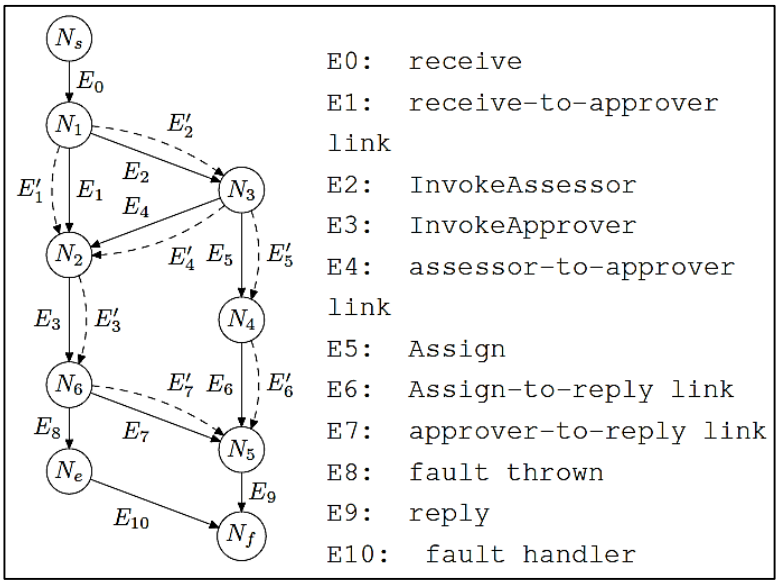


Figure 2.13 An example XCFG [41]

2.3 Testing Tools

In this section, Table 2.1 summarizes some of the commercial, free and open source tools that aid in testing by generating the test cases or by verifying the models.

Table 2.1 List of testing tools

S. No.	Tool Name	Description
1.	AgitarOne Agitator [44]	• Helps in interpreting the behavior of user code. • Prevents bugs. • Helps in maintenance of code.
2.	JUnit [45]	• Open Source. • Unit testing framework written in Java. • Helps in writing the tests.
3.	Java Path Finder (JPF) [46]	• Open source by NASA. • Translates Java into PROMELA model for further checking from SPIN model.
4.	TefKat [47]	• Open source, Eclipse modelling framework model transformation engine. • Works for a model-driven development language.
5.	MOFScript [48]	• Open source. • Helps to transform model into text and Eclipse modelling framework.
6.	SeDiTec [49]	• Generates automatic test stubs from sequence diagrams.

7.	T-VEC [50]	• Helps in automation. • Aids in analysis of Simulink model. • Produces unit, integration and system level test cases.
8.	Safety test builder [51]	• By chiasTek. • Automatically generates the test cases for Stateflow® and Simulink based implementations. • Also reduces testing time with an improved reliability.
9.	BEACON for Simulink (B4S) [52]	• By applied dynamics international (ADI). • B4S designer converts Simulink and Stateflow®'s simulation-oriented models to designs of production quality. • B4S code generators then convert design to ANSI C or Ada code.
10.	Reactis [53]	• Commercial tool for validation of Simulink/Stateflow® model. • Implemented in Standard ML.
11.	Simulink Design Verifier [54]	• Identifies design errors by using formal methods without running the program or extensive test cases.
12.	aPET [55]	• Generates test cases for distributed asynchronous languages which are constructed on concurrent objects.
13.	MultithreadedTC [56]	• Unit testing tool based on Java. • Allows to write tests for specific interleavings of concurrent program.
14.	Pex [57]	• A white box unit testing tool for .NET. • Provides high code coverage with a small test suite.
15.	PET [58]	• Works on subset of Java bytecode program. • Generates test cases based on partial evaluation. • Employs white box technique.
16.	Extension of TCM–Toolkit for Conceptual Modelling [59]	• First tool for verifying activity diagrams. • First usable front end to model checkers. • Verifies workflow models specified in UML activity diagram.
17.	Tool by R. Ferreira *et al.*(MoCAT) [60]	• Model coverage analysis (MoCAT) tool. • Takes UML state machine model and test suite as input. • Shows model test coverage results in the form of colored state machine model.
18.	TSGen [61]	• Tool for model based testing of activity diagrams. • Test scenarios are automatically generated.

		• Saves efforts of tester and helps in better planning of testing schedules.
19.	ConAn [62]	• Concurrency analyser (ConAn) tool. • Unit testing tool for testing concurrent Java components. • Automates generation of test drivers from test sequences.
20.	UMLTGF [63]	• Uses gray box method to generate test cases from activity diagrams directly. • Implements category partitioning.
21.	TorX [64]	• Based on specification. • Works on the fly testing for functional correctness of the system.
22.	BONUS (constraint solver) [65]	• Helps in path analysis and white box testing. • Solves path conditions and identifies infeasible paths.
23.	SPIN model checkers [66]	• Simple promela interpreter (SPIN) validates correctness of distributed software model. • Freely available. • Saves memory and improves performance.
24.	NuSMV model checker [67]	• Open source. • First tool based on binary decision diagram (BDD) for checking the models.
25.	CHESS [68]	• Open source concurrency testing tool. • Discovers and reproduces heisenbugs. • Ensures that each run produces dissimilar interleaving for testing. • Reproduces error by repeating the same interleaving for debugging.
26.	GAMBIT [69]	• Combines heuristic guided fuzzing for speed and stateless model checking for reliability, progress and reproducibility. • Faster in finding bugs than previous stateless tools.
27.	ConTest [70]	• Discovers and removes bugs related to concurrency. • Advances quality of testing and shrinks development costs.
28.	ConSUITE [71]	• Extension to EVOSUITE. • Helps in unit testing. • Automatically generates tests using a new criteria, concurrency coverage criteria.
29.	EVOSUITE [72]	• Unit testing tool.

		• Generates test suites automatically for Java classes.
30.	TEAGER [73]	• Based on UML state machine. • Conforms UML semantics. • Resolves non-determinism probabilistically.
31.	QTP [74]	• Quick test professional (QTP) tool. • Uses VB scripting language. • Graphical interface with record playback automation.
32.	TGV [75]	• Acronym for test generation with verification. • Integrated into toolset CADP. • Generates test cases based on specifications of the system and a test purpose.
33.	CADP [76]	• Construction and analysis of distributed processes (CADP) is a toolbox for design of asynchronous concurrent systems. • Input languages are LOTOS and BCG. • Internally works with labelled transition systems.
34.	SAMSTAG [77]	• Generates conformance test suites.

Chapter 3
Gap Analysis and Problem Statement

This chapter illustrates the research gaps encountered by systematically analyzing previously work done in the area of test case generation using code and UML. Based on the gap analysis, problem statement is framed for the research work.

3.1 Gap Analysis

Based on the systematic literature review of code and model based test case generation, following gaps have been identified:

- Interference dependence has not been taken into account while generating test sequences [15, 16, 79].
- Till date, simple statechart diagrams have only been considered for test case generation and little consideration has been given for generating test cases through statechart diagrams with concurrent sub-states and events [35].
- There is a need of better algorithm for considering data safety errors, which are not covered by depth first search (DFS), while generating the test sequences [10].

3.2 Problem Statement

After thorough review of literature related to model based test case generation in concurrent programs, this has been analyzed that an approach better than already existing ones can be proposed that use a combined form of DFS and BFS (breadth first search) with an optimization to reduce the number of test cases. Already existing approach [63], uses DFS on XML to generate test sequences. Applying DFS is inefficient as compared to a combination of DFS and BFS because DFS can generate test sequences that do not detect data safety errors [10]. Also, interference dependence between the statements of a concurrent program has not been considered as a criteria to generate the test sequences from code. Enhancing a test sequence with more coverage criterion increases the chances of detecting errors that can occur in a program. This enhancement provides a way to test the untested portions of the program [17].

Chapter 4
Methodology

In the present approach, test sequences are generated using code for Java7 fork/join program and UML activity diagram for ATM system as presented in this chapter.

4.1 Test Sequence Generation from Code

This section presents the methodology used for generating test sequences from code. Test sequences are generated for a Java7 fork/join example program. Outline of the proposed methodology is presented in Figure 4.1.

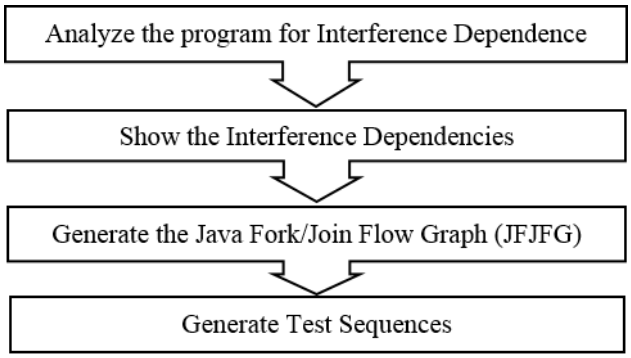


Figure 4.1 Methodology of the proposed approach

A Java7 program, for adding the elements of an array by utilizing the Java7 fork/join capability as shown in Figure 4.2, is taken as input. If number of elements to be added are less than or equal to 5000 i.e. *SEQUENTIAL_THRESHOLD*, the work is carried out sequentially otherwise the work is divided using fork() method.

The proposed methodology is summarized as four steps presented in this section.

4.1.1 Identifying Interference Dependence

Definitions and uses of variables inside the compute() method i.e. for simultaneously executable sections, are treated as interference dependence. The steps for finding the interference dependence are given in the Algorithm 1.

Algorithm 1: Identifying interference dependence

/* *interference* is the output adjacency matrix having interference dependencies*/

Input: Java7 fork/join program

Output: Interference dependence matrix

1. Initialize each cell of the matrix *interference*[][] to 'false'.
2. Provide numbering to all statements of the input program.
3. Traverse compute() method statement by statement. //because compute() method //has fork/join section which makes parallel executions inside the program.
4. If a variable *v* is defined at statement L_1 and used at statement L_2, Then
   interference[L_2][L_1] = true. //statement L_2 is dependent on statement L_1.

4.1.2 Visualizing Interference Dependence

After identifying interference dependence among the various statements, these are shown in the form of a directed graph for better understanding of the concepts. Steps for generating the directed graph of interference dependence among statements of the program is given in Algorithm 2.

Algorithm 2: Visualizing interference dependence

/* *Visited* is the list of nodes already drawn, *interference* is the adjacency matrix for interference dependence */

Input: Adjacency matrix for interference dependence.

Output: Directed graph

1. Initialize array *Visited* = Φ.
2. Traverse the interference dependence matrix i.e. *interference*[][] for each cell.
3. Repeat the step 4 until all the nodes are visited.
4. If *interference*[*i*][*j*] = true, Then
   a. If *i* OR *j* OR both nodes ∉ *Visited*, Then
      Draw the corresponding node(s).
      Add *i* OR *j* OR both nodes to *Visited*.
   b. Draw directed line form node *i* to *j*, showing node *i* is dependent on node *j*.

Output of Algorithm 2, i.e. interference dependencies for Java7 fork/join example program are shown in Figure 5.3 in the results chapter. It can be interpreted from the code shown in Figure 4.2 that statement number 28 and 29 are dependent on

themselves, statement number 35 and 36 are dependent on statement number 34 and similarly other statements are dependent on one another.

```
1.    import java.util.concurrent.ForkJoinPool;
2.    import java.util.concurrent.RecursiveTask;
3.
4.    class Globals
5.    {
6.    static ForkJoinPool fjPool = new ForkJoinPool();
7.    }
8.
9.    class Sum extends RecursiveTask<Long>
10.   {
11.   static final int SEQUENTIAL_THRESHOLD = 5000;
12.
13.   int low;
14.   int high;
15.   int[] array;
16.
17.   Sum(int[] arr, int lo, int hi)
18.   {
19.   array = arr;
20.   low   = lo;
21.   high  = hi;
22.   }
23.   protected Long compute()
24.   {
25.    if(high-low<=SEQUENTIAL_THRESHOLD)
26.    { // if the task to be done is small: do the work now
27.    long sum=0;
28.    for(int i=low;i<high;i++)
29.       sum=sum+array[i];
30.     return sum;
31.    }
32.    else
33.    { //the task to be done is too big: divide the work
34.    int mid=low+(high-low)/2;
35.    Sum left=new Sum(array, low, mid);
36.    Sum right=new Sum(array, mid, high);
37.    left.fork();
38.    long rightAns=right.compute();
39.    System.out.println("This is the sample program");
40.    long leftAns=left.join();
41.    return leftAns+rightAns;
42.    }}
43.   static long sumArray(int[] array)
44.   {
45.   return Globals.fjPool.invoke(new Sum(array,0,array.length));
46.   }}
```

Figure 4.2 Input Java file [78]

4.1.3 Generating Java Fork/Join Flow Graph (JFJFG)

JFJFG is drawn for the compute() method of the input program. Call to the fork() method is shown as call to the parallel tasks which invokes the compute() method for

that variable. And call to the compute() after call to the fork() method, invokes other parallel activity. Whereas call to the join() method returns the value of the thread on which fork() was called. The execution is just like sequential methods up to fork() method call and after join() method call. The steps for drawing the JFJFG are presented in the Algorithm 3 and output of Algorithm 3 is shown in Figure 5.2 in the results chapter.

Algorithm 3: Drawing JFJFG (Java7 Fork/Join Flow Graph)

/* array '*fork_join*' is the array to store line numbers of call to fork() and join() */

Input: Java7 fork/join program

Output: Java7 fork/join flow graph (JFJFG)

1. Initialize array *fork_join* = Φ.
2. Search for compute() method. //in this method, fork() and join() calls are considered.
3. Repeat step 4 for each fork/join call.
4. Note statement number of fork(). Say it is at statement L_1 and corresponding join() is at statement L_2, for object *v*.
   $fork\_join_v[0] = L_1$ and $fork\_join_v[1] = L_2$.
5. Repeat step 6 for each *fork_join* variable entry in *fork_join* array.
6. Generate flow graph using *fork_join* array by using the following steps:
   Show all the statements in sequential order up to $fork\_join_v[0]$ statement.
   Show the statements between $fork\_join_v[0]$ and $fork\_join_v[1]$ statements in parallel in flow graph. //because these statements can execute in parallel.
   Show the flow from $fork\_join_v[0]$ to the statement in which compute() method is called by directed line.
   Show all the remaining statements in compute() method in sequential manner after the statement number $fork\_join_v[1]$.

4.1.4 Generating Test Sequences

After generating Java fork/join flow graph (JFJFG), it is traversed on the basis of all node and all path coverage criteria considering the interference dependence in order to find out the test sequences. Steps for generating the test sequences are shown in the Algorithm 4.

Algorithm 4: Generate Test Sequences

/* V_p is the present node being explored and V_{end} is the end node of compute() method, *visited* is the array that stores the status of the nodes whether nodes are visited or not*/

Input: A Java7 fork/join flow graph (JFJFG) G (V, E)

Output: Test sequences

1. Start from the beginning of compute() method.
2. Repeat until V_p != V_{end}.
3. If V_p is a call to fork() method, use algorithm breadth first search (BFS):
    a. Mark all the nodes as unvisited.
    $\forall\ V_i \in V$, set *visited*[V_i] = false.
    b. Enqueue the present node V_p.
    c. Dequeue from the front of queue. Mark it as V_p. Set *visited*[V_p] = true.
    d. Enqueue all the nodes adjacent to V_p.
    e. Repeat the steps 3.b to 3.d until the queue is empty.
    f. Exit when the node join() is found.
4. If V_p is any other statement, use algorithm depth first search (DFS):
    a. Mark all the nodes as unvisited.
    $\forall\ V_i \in V$, set *visited*[V_i] = false.
    b. Push the present node V_p on the stack.
    c. Pop from the top of stack. Mark it as V_p. Set *visited*[V_p] = true.
    d. Push all the nodes adjacent to V_p on the stack.
    e. Repeat the steps 4.b to 4.d until the queue is empty.
    f. Exit.
5. End of repeat.

4.2 Test Sequence Generation from Model

This section presents the proposed methodology used to generate the test sequences from a UML activity diagram of ATM system [10]. This example contains a decision node, and fork and join constructs of activity diagram. The techniques presented in papers [10, 21, 22, 23, 25 and 28] use an intermediate testing model for generating the test cases from UML and include transformation overheads [63]. However, the

proposed approach uses no intermediate model for test sequence generation, which reduces the transformation cost.

4.2.1 Generation of Activity Diagram

In the present time, various tools are available with free trial versions for drawing UML diagrams. One such tool is MagicDraw 17.0.5 [80]. It provides an easy-to-use interface for drawing the activity diagram in a project. The example taken in this section is the activity diagram for the functioning of an ATM system [10]. Figure 4.3 represents an activity diagram generated by using MagicDraw 17.0.5 tool.

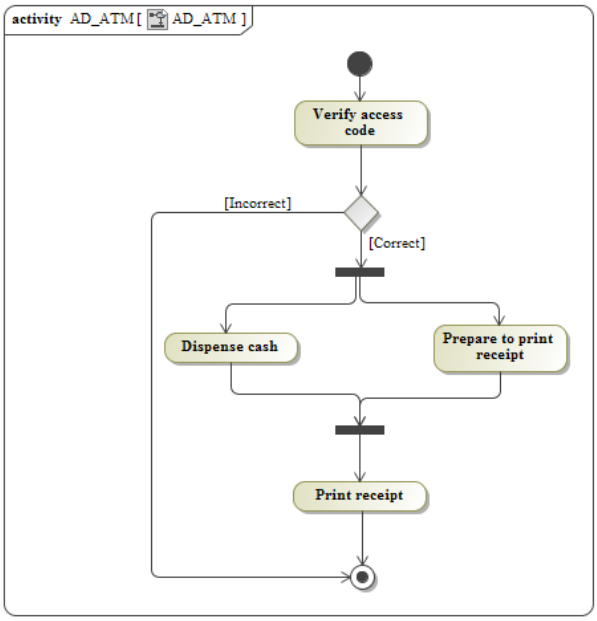


Figure 4.3 Activity diagram for ATM system [10]

4.2.2 Converting the Activity Diagram to XML

Self- descriptive language, XML provides us an easy way to analyze and traverse the file. MagicDraw converts the UML project into XML format whose part is shown in Figure 4.4.

4.2.3 Finding the Incoming and Outgoing Edges

In this step, XML file is traversed to find out incoming and outgoing edges for each node of the activity diagram. The algorithm to find out the incoming and outgoing edges for each node is presented in Algorithm 5. The output of Algorithm 5 i.e. matrices, is used as an input to Algorithm 6.

```
<node xmi:type='uml:InitialNode' xmi:id='_17_0_5_13240322_1402215535645_860160_3402'
visibility='public'>
  <outgoing xmi:idref='_17_0_5_13240322_1402254094655_275484_3542'/>
</node>
<node xmi:type='uml:CallBehaviorAction'
xmi:id='_17_0_5_13240322_1402254094626_484753_3540' name='Verify access code'
visibility='public'>
  <incoming xmi:idref='_17_0_5_13240322_1402254094655_275484_3542'/>
  <outgoing xmi:idref='_17_0_5_13240322_1402254154912_128180_3572'/>
</node>
<node xmi:type='uml:DecisionNode' xmi:id='_17_0_5_13240322_1402254121435_196672_3545'
visibility='public'>
  <incoming xmi:idref='_17_0_5_13240322_1402254154912_128180_3572'/>
  <outgoing xmi:idref='_17_0_5_13240322_1402254178745_145733_3584'/>
  <outgoing xmi:idref='_17_0_5_13240322_1402255116722_270681_3770'/>
</node>
<node xmi:type='uml:ForkNode' xmi:id='_17_0_5_13240322_1402254170286_251758_3575'
visibility='public'>
  <incoming xmi:idref='_17_0_5_13240322_1402254178745_145733_3584'/>
  <outgoing xmi:idref='_17_0_5_13240322_1402254396810_631132_3672'/>
  <outgoing xmi:idref='_17_0_5_13240322_1402254422747_545085_3691'/>
</node>
<node xmi:type='uml:CallBehaviorAction'
xmi:id='_17_0_5_13240322_1402254291354_755240_3599' name='Dispense cash'
visibility='public'>
  <incoming xmi:idref='_17_0_5_13240322_1402254422747_545085_3691'/>
  <outgoing xmi:idref='_17_0_5_13240322_1402254443836_765312_3701'/>
</node>
<node xmi:type='uml:CallBehaviorAction'
xmi:id='_17_0_5_13240322_1402254298209_843337_3613' name='Prepare to print receipt'
visibility='public'>
  <incoming xmi:idref='_17_0_5_13240322_1402254396810_631132_3672'/>
  <outgoing xmi:idref='_17_0_5_13240322_1402254479679_249931_3716'/>
</node>
<node xmi:type='uml:JoinNode' xmi:id='_17_0_5_13240322_1402254435800_236463_3694'
visibility='public'>
  <incoming xmi:idref='_17_0_5_13240322_1402254443836_765312_3701'/>
  <incoming xmi:idref='_17_0_5_13240322_1402254479679_249931_3716'/>
  <outgoing xmi:idref='_17_0_5_13240322_1402255072972_191099_3747'/>
</node>
<node xmi:type='uml:CallBehaviorAction'
xmi:id='_17_0_5_13240322_1402255063019_973277_3732' name='Print receipt'
visibility='public'>
  <incoming xmi:idref='_17_0_5_13240322_1402255072972_191099_3747'/>
  <outgoing xmi:idref='_17_0_5_13240322_1402255092798_451943_3758'/>
</node>
<node xmi:type='uml:ActivityFinalNode'
xmi:id='_17_0_5_13240322_1402255077781_487114_3750' visibility='public'>
  <incoming xmi:idref='_17_0_5_13240322_1402255092798_451943_3758'/>
  <incoming xmi:idref='_17_0_5_13240322_1402255116722_270681_3770'/>
</node>
```

Figure 4.4 Part of XML file

Algorithm 5: Find_Incoming_and_Outgoing_Edges

Input: XML file of UML activity diagram

Output: *incoming*[][] and *outgoing*[][] matrices

/**incoming*[][] is the matrix to store the *id* and *type* of each node and its incoming edges. *outgoing*[][] matrix stores each node's *id*, *type* and outgoing edges from that node*/

1. For each *line* of the XML file, repeat the steps 2 through 5.
2. Split the *line* w.r.to blank space.
3. Search for starting of <node> tag. Then find the *type* and *id* of the node within the attributes of <node> tag. Store the *type* and *id* at 0^{th} and 1^{st} columns i.e. *incoming*[][0], *incoming*[][1], *outgoing*[][0] and *outgoing*[][1] respectively.
4. Repeat the step 5, until the closing tag </node> is not found.
5. Search for the starting tags <incoming> and <outgoing>. Store the incoming and outgoing edges' reference ids in the same row of the respective nodes in the corresponding matrix.

4.2.4 Finding Out the Test Sequences

After finding the incoming and outgoing edges in the form of matrices, test sequences are generated using Algorithm 6. This algorithm uses an optimization to reject the infeasible paths therefore, the problem of test case explosion is also avoided in this approach.

Algorithm 6: Generate_Test_Sequences

Input: *incoming*[][] and *outgoing*[][] matrices

Output: Test sequences

1. Start with the outgoing edge of *StartNode* of activity diagram. //*outgoing*[0][2]
2. Repeat the steps 3 to 8 until all the edges are not explored. //i.e. all the entries of matrices //*incoming*[][] and *outgoing*[][] are not visited.
3. Find the *next* node to which the present edge is incoming.
4. For this node, push all the outgoing edges onto the stack.
5. If *FinalNode* is found and all nodes are not covered, create a new path.
6. If the present node is a *DecisionNode*
    a. Store the *previous* path.

b. Traverse one path along the true guard condition and mark the path as visited. Append this path to *previous* path.
c. Traverse the other path with the guard condition as false and append it to the *previous* path to get the other path.

7. If the present node is *ForkNode*
    a. Traverse all the parallel paths simultaneously until *JoinNode* is not found.
    b. Append the nodes to the current path.
8. Until the stack is not empty, pop the edges from the stack and repeat the steps 3 to 7 for each edge.
9. Reject all the paths containing *ForkNode* and not having *JoinNode* or having *JoinNode* and not containing *ForkNode*.

Chapter 5
Implementation and Results

For the methodology proposed in the Chapter 4, a prototype tool is implemented in Java7 to generate the test sequence from code and model. The snapshot of the prototype is presented in Figure 5.1.

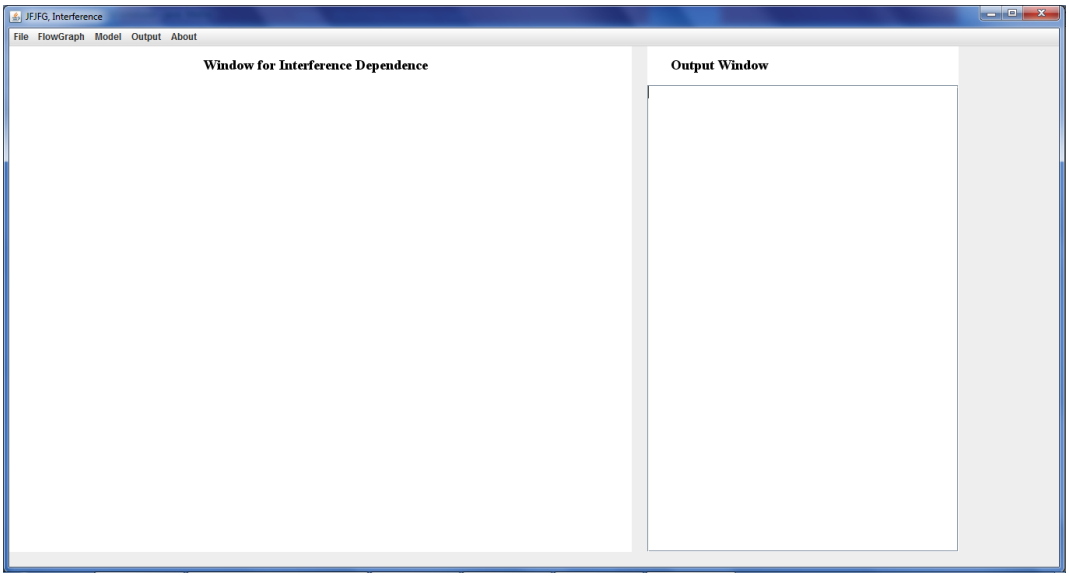


Figure 5.1 Graphical user interface of prototype tool

The GUI of the prototype tool, implemented using Java7 has the following 3 sections:

Menus: The GUI contains different menus for carrying out particular tasks as indicated by their names. The menu ‘File’ contains ‘Open’, ‘Save’ and ‘Exit’ menu items, which are used for choosing the input Java file, saving the canvas of GUI and for exiting the application respectively. The menu ‘FlowGraph’ has the menu item ‘JFJFG’, which is used to generate the Java fork/join flow graph. On clicking this menu item, the Java fork/join flow graph appears in a new JFrame. Menu ‘Model’ contains two menu items ‘Choose’ and ‘GenTestSeq’ to choose the XML file and to generate the test sequences respectively from model. The menu ‘Output’ contains a menu item ‘Show the Output’ to present the resultant test sequences in the output window shown in the GUI.

Window for Interference Dependence: It is a JPanel attached to the JFrame of GUI and contains the interference dependence graph generated after choosing the input Java file from the menu item ‘Open’ as shown in Figure 5.2 and Figure 5.3.

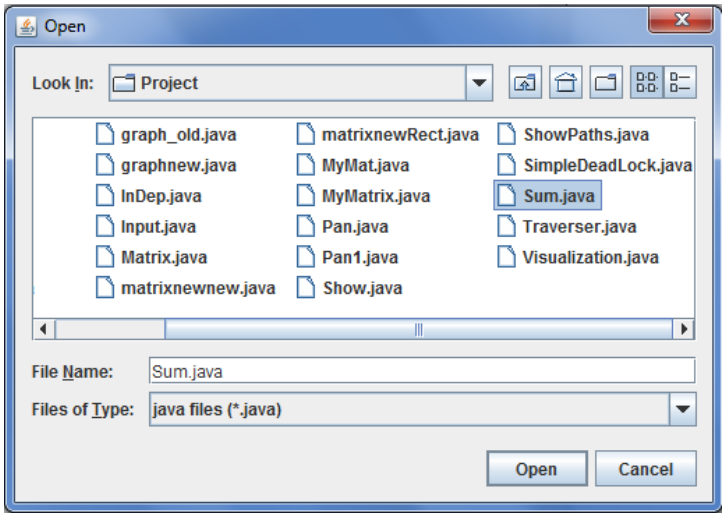


Figure 5.2 Choosing the input Java file

Output Window: It is a JTextArea placed on the JFrame of main GUI that contains the test sequences generated by analyzing the input Java and XML files.

5.1 Output Generated from Code

This section presents the outputs and results generated by applying the algorithms on the example taken as input.

5.1.1 Interference Dependence Graph

Output of Algorithm 2 is a directed graph as illustrated in Figure 5.3 that shows interference dependencies among statements of the source code.

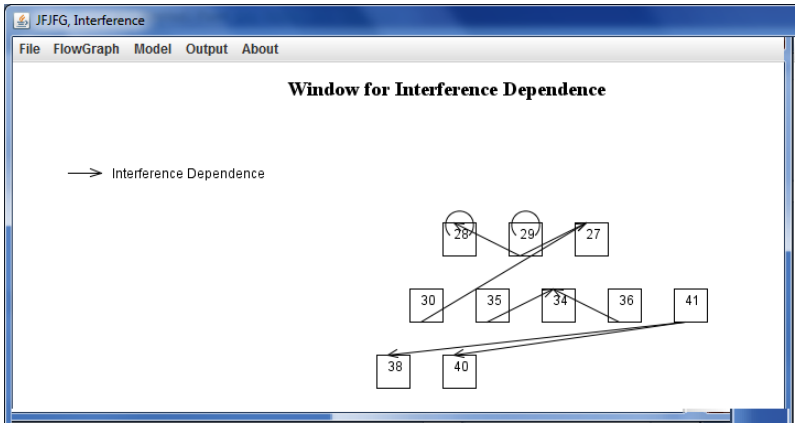


Figure 5.3 Directed graph showing interference dependencies

Self-loop on the statement number 28 and 29 represents that there exists interference dependence of the statements on themselves. Statement number 29 and 30 are

dependent on statement number 27. Statement number 41 depends on statement number 38 and 40 and so on.

5.1.2 Test Sequence Generated from Code

Output of Algorithm 3 is a Java7 fork/join flow graph (JFJFG). In the input program, call to fork() method is at statement number 37 and call to join() method is at statement number 40. Figure 5.4 shows the JFJFG for the input example program.

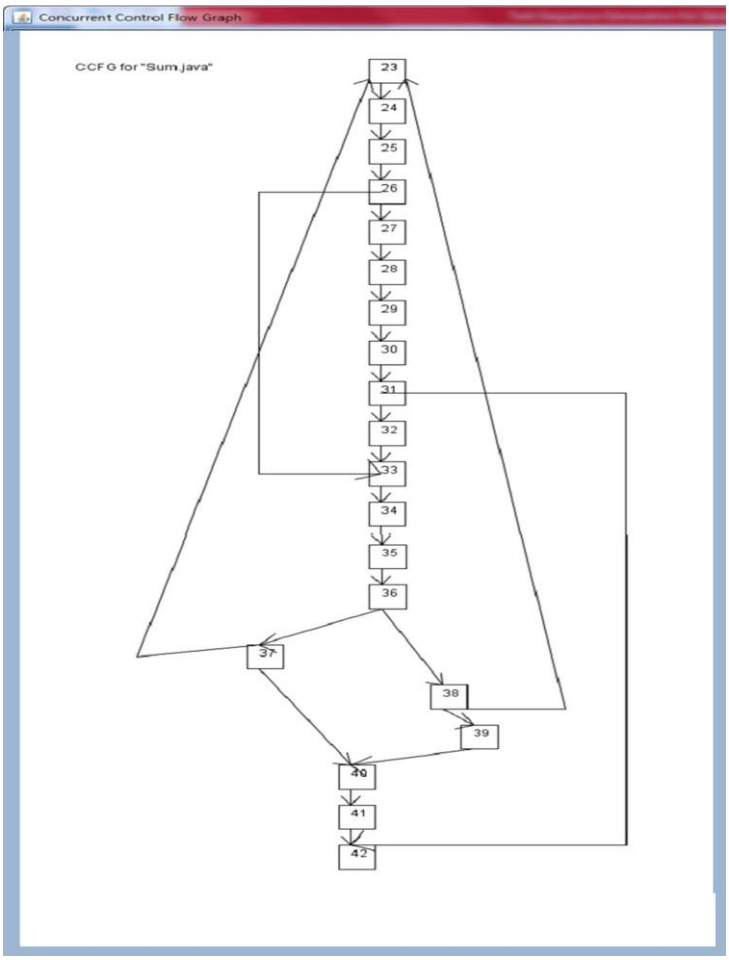


Figure 5.4 Java fork/join flow graph (JFJFG)

Table 5.1 shows the description of those nodes that are present in JFJFG. The method compute() starts at statement number 23 and ends at statement number 42. Inside the compute() method, 'if' block is from statement number 26 to 31. The 'else' block is from statement 32 to 42. The call to fork() and join() methods are at statement number 37 and 40.

The Algorithm 4 generates the test sequences for a Java fork/join flow graph. Whenever there is a call to fork() method, concurrent paths are there in the structure of the

program. Therefore algorithm BFS is applied to cover those concurrent paths simultaneously. Otherwise, for sequential paths, he algorithm DFS is used for traversing the graph and hence finding the test sequences.

Table 5.1 Description of nodes in JFJFG

Node No.	Description
23	Start of compute() method
26	Start of if() block
31	End of if() block
32	Start of else block
37	Call to fork()
40	Call to join()
42	End of else block and compute() method

The test sequences generated in the form of node numbers i.e. statement numbers by the proposed algorithm are as follows:

23→ 24→ 25... 31 → 42
23→ 24 → 25 → 26 → 33... 36→37→ 38→ 39→ 40→ 41→ 42
23→ 24 → 25 → 26 → 33... 36→ 38→ 39→37→ 40→ 41→ 42

Where *x*... *y* means nodes traversed from node *x* to node *y* in serial order.

Test Sequence 1:

Start of compute() method.
The threshold value is > difference of high and low, so 'if' part gets executed.
End of compute method.

Test Sequence 2:

Start of compute() method.
The threshold value is < difference of high and low, causing the 'else' block to execute.
If the algorithm finishes the work of fork() first, the order of execution would be like this test sequence. Or the algorithm BFS takes the left child into first consideration.

Test Sequence 3:

Start of compute() method.

The threshold value is < difference of high and low, causing the ‘else’ block to execute. If the algorithm finishes the work of compute() first, the order of execution would be like this test sequence. Or the algorithm BFS takes the right child into first consideration.

5.2 Test Sequences Generated from Model

This section presents the test sequences generated for an activity diagram for the ATM system. In this approach, infeasible test sequences are selected on the basis of two rules:

- Test sequences having fork node and no join node.
- Test sequences having join node with no fork node.

After selecting such test sequences, those infeasible paths are deleted from the set of test sequences. Test sequences generated as the output of Algorithm 6 are as follows:

Test Sequence 1: StartNode → Verify access code → Decision_Node → EndNode.

Test Sequence 2: StartNode → Verify access code → Decision_Node → ForkNode → Dispense cash → Prepare to print receipt → JoinNode → Print receipt → EndNode.

Chapter 6
Conclusion and Future Scope

This chapter concludes the proposed approach for generating the test sequences from code and UML activity diagram. Some points, that can be considered in future, are also mentioned in this section.

6.1 Conclusion

An approach is proposed for generating the test sequences for Java7 fork/join program and UML activity diagram. The coverage criteria for test sequence generation is enhanced and provides more chances to detect the errors. The proposed approach is different from already existing approaches in following points:

- A user interface is developed for the proposed approach to help the users in generating the test sequences from code and UML activity diagram.
- The problem of test case explosion is avoided in this approach by optimizing the algorithm considering both DFS and BFS.
- However, the proposed approach does not use any intermediate graphical form for test sequence generation, which was previously used by almost all the approaches.
- All node and all path coverage criteria based on interference dependence has been considered for generating test sequences from code.

6.2 Future Scope

This work focuses on test sequence generation from activity diagram and code of Java fork/join programs. However, there are some points to be explored further.

- Test sequence generation can be extended for more coverage criteria e.g. ready dependence.
- Other UML diagrams like sequence, collaboration diagrams can also be considered for generating the test sequences.

References

[1] D. M. Dhamdhere, "Processes and Threads," in *Operating Systems: A concept based approach*, 2nd ed., McGraw Hill, 2006, Ch. 5.

[2] T. Norvell. "What is concurrent programming." Internet: www.engr.mun.ca/~theo/Courses/cp/pub/cp0.pdf [Dec. 28, 2013].

[3] P. B. Hansen, "Concurrent programming concepts," *ACM Computing Surveys (CSUR)*, vol. 5 (4), pp. 223-245, ACM, 1973.

[4] I. Jovanović, "Software testing methods and techniques," *The IPSI BgD Transactions on Internet Research*, Vol. 5(1), pp. 30-41, Internet Journals, 2009.

[5] "IEEE standard for software test documentation," *IEEE Std 829-1983* , pp.1-48, 1983

[6] G. J. Myers, *Art of Software Testing*, 2nd ed. Hoboken, New Jersey: John Wiley & Sons Inc., 2004.

[7] B. Bruegge and A. H. Dutoit. "Object-oriented software engineering using UML, patterns and Java." Internet: www.cs.bilkent.edu.tr/~ugur/teaching/cs319/material/ch07lect1_UD.ppt [Jan. 1, 2014].

[8] "IEEE standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp.1-84, Dec. 31 1990.

[9] G. Booch, J. Rumbaugh and I. Jacobson, *Unified Modeling Language User Guide*, 1st ed. Addison Wesley, 1998.

[10] M. Shirole and R. Kumar, "Testing for concurrency in UML diagrams," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 5, pp. 1-8, 2012.

[11] P. Stevens, "UML and concurrency," *Abstract State Machines*, pp. 151-166. Springer Berlin Heidelberg, 2003.

[12] K. Mehner and A. Wagner, "Visualizing the Synchronization of Java threads with UML," *Proc. of IEEE International Symposium on Visual Languages*, pp. 199-206. IEEE, 2000.

[13] F. E. Allen, "Control Flow Analysis," *ACM SIGPLAN Notices*, vol. 5, no. 7, pp.1-19, ACM, 1970.

[14] V. P. Ranganath and J. Hatcliff, "Pruning interference and ready dependence for slicing concurrent Java programs," *Compiler Construction*, pp. 39-56, Springer, 2004.

[15] J. Krinke, "Context-sensitive slicing of concurrent programs," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 178-187. ACM, 2003.

[16] M. G. Nanda, S. Ramesh. "Slicing concurrent programs," *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 5, pp. 180-190, ACM, 2000.

[17] A. P. Mathur, *Foundation of Software Testing*, Pearson/Addison Wesley, 2008.

[18] "Fork/Join." Internet: http://docs.oracle.com/Javase/tutorial/essential/concurrency/forkjoin.html, 15 Oct. 2013 [Jan. 2, 2014].

[19] M. Khandai, A. A. Acharya and D. P. Mohapatra, "A survey on test case generation from UML model," *International Journal of Computer Science and Information Technologies*, vol. 2, no. 3, pp. 1164-1171, 2011.

[20] C. Mingsong, Q. Xiaokang and L. Xuandong, "Automatic test case generation for UML activity diagrams," *Proc. of the 2006 International Workshop on Automation of Software Test*, pp. 2-8, 2006.

[21] H. Kim, S. Kang, J. Baik and I. Ko, "Test cases generation from UML activity diagrams," *Proc. of Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, vol. 3, pp. 556-561, 2007.

[22] D. Kundu and D. Samanta, "A novel approach to generate test cases from UML activity diagrams," *Journal of Object Technology*, vol. 8, no. 3, pp. 65-83, May 2009.

[23] C. Sun, "A transformation-based approach to generating scenario-oriented test cases from UML activity diagrams for concurrent applications," *Proc. of 32nd Annual IEEE International Computer Software and Applications (COMPSAC)*, pp. 160-167, 2008.

[24] B. Lei, L. Wang and X. Li, "UML activity diagram based testing of Java concurrent programs for data race and inconsistency," *Proc. of 1st International Conference on Software Testing, Verification and Validation*, pp. 200-209, 2008.

[25] X. Fan, J. Shu, L. L. Liu and Q. J. Liang, "Test case generation from UML subactivity and activity diagram," *Proc. of Second International Symposium on Electronic Commerce and Security (ISECS)*, vol. 2, pp. 244-248, 2009.

[26] M. Chen, P. Mishra and D. Kalita, "Efficient test case generation for validation of UML activity diagrams," *Design Automation for Embedded Systems*, vol. 14, no. 2, pp. 105-130, 2010.

[27] S. K. Swain, D. P. Mohapatra and R. Mall, "Test case generation based on use case and sequence diagram," *International Journal of Software Engineering*, vol. 3, no. 2, pp. 21-52, 2010.

[28] M. Khandai, A. A. Acharya and D. P. Mohapatra, "A novel approach of test case generation for concurrent systems using UML sequence diagram," *Proc. of 3rd International Conference on Electronics Computer Technology (ICECT)*, vol. 1, pp. 157-161, 2011.

[29] Y. G. Kim, H. S. Hong, D. H. Bae and S. D. Cha, "Test Cases Generation from UML State Diagrams," *IEE Proc. of Software*, vol. 146, no. 4, pp. 187-192, August 1999.

[30] P. Chevalley and P. T. Fosse, "An empirical evaluation of statistical testing designed from UML state diagrams: the flight guidance system case study," *Proc. of 12th International Symposium on Software Reliability Engineering (ISSRE)*, vol. 254, no. 263, pp. 27-30 Nov. 2001.

[31] S. K. Kim, L. Wildman and R. Duke, "A UML approach to the generation of test sequences for Java-based concurrent systems," *Proc. of Australian Software Engineering Conference*, pp.100-109, 29 March-1 April 2005.

[32] L. C. Briand, Y. Labiche and J. Cui, "Automated Support for Deriving Test Requirements from UML Statecharts," *Software & Systems Modeling*, vol. 4, no. 4, pp. 399-423, 2005.

[33] P. Samuel, R. Mall and A. K. Bothra, "Automatic test case generation using unified modeling language (UML) state diagrams," *IET Software*, vol. 2, no. 2, pp. 79-93, April 2008.

[34] D. Patnaik, A. A. Acharya and D. P. Mohapatra, "Generating test cases for concurrent systems using UML state chart diagram," *Proc. of International Conference on Advances in Information Technology and Mobile Computing*, pp. 100-105, 2011.

[35] M. Aggarwal and S. Sabharwal, "Test case generation from UML state machine diagram: A survey," In *Computer and Communication Technology (ICCCT), 2012 Third International Conference on*, pp. 133-140, IEEE, 2012.

[36] T. Katayama, Z. Furukawa and K. Ushijima, "Event interactions graph for test-case generation of concurrent programs," *Proc. of Asia Pacific Software Engineering Conference*, pp. 29-37, 1995.

[37] T. Katayama, Z. Furukawa and K. Ushijima, "A test-case generation method for concurrent programs including task-types," *Proc. of Asia Pacific Software Engineering Conference (APSEC) and International Computer Science Conference (ICSC)*, pp. 485-494, 1997.

[38] T. Katayama, Z. Furukawa and K. Ushijima, "Design and implementation of test-case generation for concurrent programs," *Proc. of Asia Pacific Software Engineering Conference*, pp. 262-269, 1998.

[39] T. Katayama, E. Itoh, Z. Furukawa and K. Ushijima, "Test-case generation for concurrent programs with the testing criteria using interaction sequences," *Proc. of Sixth Asia Pacific Software Engineering Conference*, pp. 590-597, 1999.

[40] X. Bao, N. Zhang and Z. Ding, "Test case generation of concurrent programs based on event graph," *Proc. of Fifth International Joint Conference on INC, IMS and IDC, NCM*, pp. 143-149, 2009.

[41] J. Yan, Z. Li, Y. Yuan, W. Sun and J. Zhang, "BPEL4WS unit testing: Test case generation using a concurrent path analysis approach," *Proc. of 17th International Symposium on Software Reliability Engineering* (*ISSRE)*, pp.75-84, 2006.

[42] Y. Yuan, Z. Li and W. Sun, "A graph-search based approach to BPEL4WS test generation," *Proc. of International Conference on Software Engineering Advances*, pp. 14-14, 2006.

[43] Y. Zheng, J. Zhou and P. Krause, "A model checking based test case generation framework for web services," *Proc. of Fourth International Conference on Information Technology ITNG*, pp. 715-722, 2007.

[44] "AgitarOne." Internet: http://www.agitar.com/solutions/products/agitarone.html [Apr. 10, 2014].

[45] P. Louridas, "JUnit: unit testing and coiling in tandem," *Software*, vol. 22, no. 4, pp. 12-15, 2005.

[46] K. Havelund and T. Pressburger, "Model checking Java programs using Java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366-381, 2000.

[47] "Tefkat" Internet: http://tefkat.sourceforge.net/ [Apr. 10, 2014].

[48] "MOFScript Home page." Internet: http://www.eclipse.org/gmt/mofscript/ [Apr. 10, 2014].

[49] F. Fraikin, T. Leonhardt, "SeDiTeC-testing based on sequence diagrams," *Proc. of 17th IEEE International Conference on Automated Software Engineering*, pp. 261-266, IEEE, 2002.

[50] "T-VEC Tester for Simulink." Internet: http://t-vec.com/solutions/simulink.php [Apr. 10, 2014].

[51] "Safety Test Builder." Internet: http://www.chiastek.com/products/stb.html [Apr. 8, 2014].

[52] "B4S." Internet: http://www.adi.com/products/b4s/ [Apr. 8, 2014].

[53] S. Sims and D.C. DuVarney, "Experience report: the reactis validation tool," *ACM SIGPLAN Notices*, vol. 42, no. 9, pp. 137-140, ACM, 2007.

[54] "Simulink Design Verifier." Internet: http://www.mathworks.in/products/sldesignverifier/ [Apr. 8, 2014].

[55] E. Albert, P. Arenas, M. Gómez-Zamalloa and P.Y. Wong, "aPET: A test case generation tool for concurrent objects," *9th Joint Meeting on Foundations of Software Engineering*, pp. 595-598, ACM, 2013.

[56] W. Pugh and N. Ayewah, "Unit testing concurrent software," *Proc. of 22nd IEEE/ACM International Conference On Automated Software Engineering*, pp. 513-516, ACM, 2007.

[57] N. Tillmann and J.D. Halleux, "Pex–white box test generation for. NET," *Proc. of 2nd International Conference on Tests and Proofs*, pp. 134-153, Springer Berlin Heidelberg, 2008.

[58] E. Albert, M. Gómez-Zamalloa and G. Puebla, "PET: a partial evaluation-based test case generation tool for Java bytecode," *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pp. 25-28, ACM, 2010.

[59] R. Eshuis and R. Wieringa, "Tool support for verifying UML activity diagrams," *IEEE Transactions on Software Engineering*, vol. 30, no. 7, pp. 437-447, 2004.

[60] R.D.F. Ferreira, J.P. Faria and A.C.R. Paiva, "Test coverage analysis of UML state machines," *Proc. of 3rd International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 284-289, IEEE, 2010.

[61] C.A. Sun, B. Zhang, J. Li, "TSGen: A UML activity diagram-based test scenario generation tool," *Proc. of International Conference on Computational Science and Engineering (CSE'09)*, vol. 2, pp. 853-858, IEEE, 2009.

[62] B. Long, D. Hoffman, P. Strooper, "Tool support for testing concurrent Java components," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 555-566, 2003.

[63] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, Z. Guoliang, "Generating test cases from UML activity diagram based on gray-box method," *Proc. of 11th Asia-Pacific Software Engineering Conference*, pp. 284-291, IEEE, 2004.

[64] H. Bohnenkamp, A. Belinfante, "Timed testing with TorX," *Formal Methods*, pp. 173-188. Springer Berlin Heidelberg, 2005.

[65] J. Zhang, X. Wang, "A constraint solver and its application to path feasibility analysis," *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 02, pp. 139-156, 2001.

[66] G. Holzmann, "The SPIN model checker: Primer and reference manual," vol. 1003, Reading: Addison-Wesley, 2004.

[67] "NuSMV: a new symbolic model checker." Internet: http://nusmv.fbk.eu/ [Apr. 12, 2014].

[68] "CHESS: Find and Reproduce Heisenbugs in Concurrent Programs," Internet: http://research.microsoft.com/en-us/projects/chess/ [Apr. 12, 2014].

[69] K.E. Coons, S. Burckhardt, M. Musuvathi, "GAMBIT: effective unit testing for concurrency libraries," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 15-24, ACM, 2010.

[70] "ConTest - A tool for testing multi-threaded Java applications." Internet: https://www.research.ibm.com/haifa/projects/verification/contest/ [Apr. 12, 2014].

[71] S. Steenbuck and G. Fraser, "Generating unit tests for concurrent classes," *Proc. of Sixth International Conference on Software Testing, Verification and Validation (ICST)*, pp. 144-153, IEEE, 2013.

[72] G. Fraser and A. Arcuri, "EvoSuite: On the challenges of test case generation in the real world," *Proc. of IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*, pp. 362-369, IEEE, 2013.

[73] T. Santen and D. Seifert, "TEAGER-Test automation for UML state machines," *Software Engineering*, vol. 79, pp. 73-84, 2006.

[74] "Quick Test Professional." Internet: http://qtpfaq.wordpress.com/2007/08/03/what-is-quick-test-professional-tool/ [Apr. 10, 2014].

[75] J.R. Calamé, *Specification-based test generation with TGV*, Centrum voor Wiskunde en Informatica, 2005.

[76] M. del Mar Gallardo, P. Merino, D. Sanán and C. Joubert, "On-the-fly model checking for C programs with extended CADP in FMICS-jETI," *Proc. of 12th IEEE International Conference on Engineering Complex Computer Systems*, pp. 321-329, IEEE, 2007.

[77] J. Grabowski, R. Scheurer, Z.R. Dai and D. Hogrefe, "Applying SaMsTaG to the B-ISDN protocol SSCOP," *Testing of Communicating Systems*, pp. 397-415, Springer, US, 1997.

[78] D. Grossman. "Beginner's Introduction to Java's ForkJoin Framework." Internet: http://homes.cs.washington.edu/~djg/teachingMaterials/grossmanSPAC_forkJoin Framework.html [Jan. 10, 2014].

[79] J. Graf, M. Hecker, M. Mohr and B. Nordho, "Lock-sensitive interference analysis for Java: Combining program dependence graphs with dynamic pushdown networks," *Proc. of 1st International Workshop on Interference and Dependence*. 2013.

[80] "Downloads." Internet: https://www.magicdraw.com/download, 10 May 2012, [Dec. 22, 2013].

List of Publications

Accepted and Presented:

[1] Vipin Verma and Vinay Arora, "A novel approach for automatic test sequence generation for Java fork/join from activity diagram," Presented at International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), Tamilnadu, IEEE, 2014.

Accepted:

[1] Vipin Verma, Aman Jyoti and Vinay Arora, "Deadlock detection with dependence graph due to wait-notify dependency in multithreaded programs," Second International Conference on Emerging Research in Computing, Information, Communication and Applications (ERCICA), Bangalore, Elsevier, 2014.

[2] Vipin Verma and Vinay Arora, "Test sequence generation for Java7 fork/join using interference dependence," Journal on Today's Ideas – Tomorrow's Technologies, vol. 2, no. 1, June 2014.

Communicated:

[1] Vipin Verma and Vinay Arora, "Tools and techniques for generating test cases for concurrent programs," 7th International Conference on Contemporary Computing (IC3), Noida, IEEE, 2014.