TH-6982

AUTOMATED FSM EXTRACTION FROM MODEL-BASED FORMAL SPECIFICATION

TO 6982

By

SHAFAQ KHALID

(94- FAS/MSSE)

Supervised By

Dr.Aamer Nadeem

Co-supervised By

Mr. Adnan Ashraf

Department of Software Engineering

Faculty of Basic and Applied Sciences

International Islamic University, Islamabad

2010

بسم الله الرحمن الرحيم

In The Name of Allah The Most Beneficent And The Most Merciful

Department of Software Engineering

International Islamic University, Islamabad

Date: ______________

FINAL APPROVAL

It is certified that we have read the project titled "*Automated FSM Extraction from Model Based Formal Specification*" submitted by **Miss Shafaq Khalid Reg. No. 94-FAS/MSSE.** It is our judgment that this research is of sufficient standard to warrant its acceptance by International Islamic University, Islamabad for the degree of MS in Software Engineering.

COMMITTEE

External Examiner:
Dr. Ehsan Ullah Munir
Assistant Professor
Department of Computer Science
COMSATS Institute of IT, WAH Cantt

[illegible]

Internal Examiner:
Mr. Usman Nasir
Assistant Professor
Department of Software Engineering
International Islamic University, Islamabad

Usman

Supervisor:
Dr. Aamer Nadeem
Associate Professor
Department of Computer Science
Mohammad Ali Jinnah University, Islamabad

Aamer

Co-supervisor:
Mr. Adnan Ashraf
Assistant Professor
Department of Software Engineering
International Islamic University, Islamabad

[illegible]

A dissertation submitted to the
Department of Software Engineering,
International Islamic University, Islamabad
as a partial fulfillment of the requirements
for the award of the degree of
MS in Software Engineering.

To My Very Kind Supervisor and My Loving Family

"My Lord have Mercy on them (Parents) both as they did care for me when I was little"

(AL-QURAN 17:24)

DECLARATION

I, hereby declare that "Automated FSM Extraction from Model-based Formal Specification" research work, neither as a whole nor as a part thereof has been copied out from any source. I have completed this research and the accompanied report entirely on the basis of my personal efforts made under the sincere guidance of my supervisor. No portion of the work presented in this report has been submitted in support of any application for any other degree or qualification of this or any other university or institution of learning.

Shafaq Khalid
94-FAS/MSSE

ACKNOWLEDGEMENT

I pay my deepest thanks, my extreme humbleness and deepest gratitude, before very kind and very merciful THE ALMIGHTY ALLAH (SWT) for being there, for blessing me with the wisdom, for the supportive and cooperative people around me, and for granting me the strength and capability to accomplish this project.

I am very thankful to my very kind, very cooperative, encouraging, dynamic and able supervisor, Dr. Aamer Nadeem for taking me under his supervision and providing me the guidance and direction during my research.

I am very thankful to the implementers Ansar Rafique and Sadia Sarwar for implementing the tool for the proposed approach.

I am extremely grateful to my parents for being so patient and cooperative, my sister Sadaf Mustafa, my brother Waqas Khalid, my brother in-law Ghulam Mustafa and my nephew Ahmed Mustafa for being there and for their continuous solace and forbearance.

I also thank my Co-supervisor Mr. Adnan Ashraf for being such a support at International Islamic University. I adore and pray for Salma Imtiaz for guiding and assisting me in approaching the right supervisor. I am also very thankful to my own self for being so consistent and patient in disappointing moments throughout my research thesis. I am also extremely grateful to my friends Munira Ghulam Nabi for her care and concern, Munazzah Jabeen for providing me comfort in worrying situations, Sundus Javed and Sidra Sultan for being so criticizing and rude for completion of my research thesis in time, and the rest of my fellows and friends.

My special thanks are for each of the CSD members at MAJU for accepting me as their own family member even being from another university. Especially I would like to thank Zohaib Haye for his cooperation and support, to brother Bilal Bashir for his positive reinforcement, to Bilal Latif for his feedback in initiating my research idea, to Muhammad Usman and Atif Jillani for their valuable suggestions and guidance.

Last but not the least I am also thankful to the cooperative staff at my work place Federal Investigation Agency (FIA). I also admire the negative reinforcement of some people without whose discouragements, taunts and derisions, I would not have attained the energy and strength as a reaction to complete my research.

Shafaq Khalid

PROJECT IN BRIEF

Project Title:	Automated FSM Extraction from Model-based Formal Specification.
Objective:	To automatically extract FSM from Input and Output Predicates of Object-Z class Specifications.
Undertaken By:	Shafaq Khalid
Supervised By:	Dr. Aamer Nadeem
Co-Supervised By:	Mr. Adnan Ashraf
Starting Date:	December 2008
Completion Date:	December 2009
Operating Systems:	Windows XP
Tool Used:	Object-Z Specification Language, Jdeveloper, LaTex
System Used:	Intel Pentium IV, 3 G. Hz Microprocessor

Shafaq Khalid
94-FAS/MSSE

ABSTRACT

Formal methods have significant importance in making programs to meet requirements for safety, accuracy, security, unambiguity and any other critical property. Formal methods are based on mathematical models, thus make specifications quite clear, unambiguous and concise. Hence, these methods are used by many software testing techniques and provides good opportunity for automated testing and for adding tool support. Many formal specification-based techniques have made use of Finite State Machine (FSM) for the testing purpose. Therefore, FSM extraction from model-based formal specification is of substantial importance. Some techniques for FSM extraction do exist in literature but none of them is fully automated. A major challenge in automatic FSM generation from a class specification is the identification of states and transitions for each operation given in that class. However, extracting states from abstract formal language like Z or Object-Z is not an easy task. Specifications in such formal languages include mixed pre- and postconditions for defining operations. These pre- and postconditions are defined in terms of predicates, expressions or complex data types. This thesis presents an automated approach for the identification of states and transitions of FSM from a class specification. Approach is also demonstrated through an example. A tool for automated generation of finite state machine (AGFSM) is also developed. Therefore, this research as an initial step towards automatic FSM generation provides an important development towards fully automated FSM generation and consequently facilitating automation of specification based testing approaches.

Table of Contents

List of Figures

List of Tables

1

Introduction

1. INTRODUCTION

Formal methods are based on sound mathematical principles and notation. The use of formal methods in specification and design phases brings the highest benefits, because the specification bugs, ambiguities and inconsistencies can be eliminated earlier in the software development life cycle. Use of formal methods also convinces that the specification meets requirements for safety, accuracy, security, un-ambiguity and other critical properties (Smith, 2000).

The use of formal specifications has become vital, not only in the development of high integrity systems, but also for complex systems. Formal specifications are more precise than specifications written in natural languages (Jacky, 1996; Pressman, 2001; Sommerville, 2000). Specifications written in natural language are vague, imprecise, unclear and not machine-processable. Therefore, using formal specification introduces the possibility of the formal and automatic analysis of specification and the source code, for generating a test oracle, for finding appropriate paths through a finite state structure and making tester to be clearer about what it means for a system to pass a test (Hierons et. al., 2009).

Formal specifications are constructed during requirements analysis phase, followed by design, implementation and verification. Formal specifications are widely accepted as they form the basis of program verification, thus providing correctness proof (Huaikou et al., 2000), which is almost non-viable for non-trivial systems. However, to gain confidence in such systems, we need to perform testing based on specification. Although the use of formal specification enhances reliability, it does not eliminate the likelihood of errors (Bowen et al., 2002; Hierons, 1997). Thus, testing is still required. Specification-based testing (Poston, 1996) has several advantages over code-based testing (Offut et al., 2003). Using formal specifications in specification based testing (Donat, 1997) is particularly significant because it provides the opportunity to automate the testing process (Huaikou et al., 2006) and for providing tool support to such approaches (Poston, 1996).

Model-based formal specification languages like Z (Spivey, 1992), B (Abrial, 1996), Object-Z (Smith 2000; Duke et al., 1991), and VDM (Jones, 1996) provide valuable information that can be exploited in specification-based testing.

Finite State Machine (FSM) when used with formal methods, provides significant advantages. FSM being an abstract machine with finite and fixed number of states and transitions has appeared as an important construct in specification-based testing. Much interest has been seen in testing from FSMs as a result of their suitability for modeling or specifying state-based systems (Hierons, 2010). Object-oriented systems can effectively be modeled using FSM, by showing method interactions in a class and testing of class coverage (Huaikou et al., 2006). FSM has also been used by many formal specification-based testing techniques (Huaikou et al., 2006; Murray et al., 1998; Dick et al., 1993; Hierons, 1997) for modeling, implementing and testing software systems.

Significance of FSM extraction from formal specifications can be shown by their wide use for a variety of purposes such as:

a) In facilitating testing process, i.e., test sequencing, test case generation and test case execution (Poston, 1996).
b) As an explicit system behavior representation in order to implement abstract models (Sun et al., 2005).
c) Modeling the interactions between data members (as states) and member functions (as transitions) of a class (Hong et al., 1995).

1.1 Research Problem

Using formal specifications in specification-based testing is known as formal specification-based testing (Donat, 1997). Many formal specification-based testing techniques based on FSM exist in the literature (Sun et al., 2005; Huaikou et al., 2006; Murray et al., 1998; Dick et al., 1993; Hierons, 1997) as discussed in detailed literature survey in chapter 2.

However, generation of FSM from Object-Z specifications is not a simple task due to implicitly given pre- and postconditions in Object-Z specifications unlike VDM (with separate pre- and postconditions). Pre- and postconditions in an operation form the basis to identify the pre- and post-states of an FSM but in Object-Z specifications pre- and postconditions are not explicitly specified unlike VDM-SL with explicitly specified pre- and postconditions. Due to implicitly given pre- and postconditions in Object-Z specifications it is difficult to identify states and transitions from the pre- and postcondition of class operations. Consequently, none of the exiting techniques for FSM extraction from model-based formal specification provides automation for derivation of states and transitions.

States and transitions are the two main components, an FSM is constructed from. Hence, state identification and transition calculation from the Object-Z class specifications are two important tasks to make a class FSM. States of FSM class correspond to the objects and transitions represent the method calls in a class. However, extracting states from abstract formal language Object-Z is not an easy task. Object-Z class specifications include mixed pre- and postconditions that define an operation, where each condition itself, is defined in terms of predicates, expressions or complex data types. Since precondition identifies the pre-states and postcondition identifies the post-states.

Each predicate that defines the pre- and postcondition of an operation participates in identifying pre- and post-state respectively. When the pre- and postcondition is defined in terms of a complex expression consisting of multiple predicates, it becomes even more difficult to identify states for such operations. Each individual predicate defining the pre- and postconditions participate in identifying class state space. Hence, it becomes important to make use of separated predicates which makes a pre- or postcondition. Each predicate can then be analyzed for the variables involved in defining the predicate and the region of values those variables might offer. Since, the variables involved in the predicate and the regions defined by them are needed to derive states and later on in determining the transitions.

1.2 Related Work

Existing techniques include FSM generation from structured languages like Z, VDM-SL and object-oriented formal specifications like Object-Z. However, FSM generated from structured and object-oriented formal specifications are entirely different in their behavior. Object-oriented formal specifications provide FSM of a class in which states are obtained from class objects and transitions show method calls among class methods. The FSM extracted from non object-oriented formal specifications corresponds to FSM of the whole system in which states are extracted from state schemas and transitions represent the method calls. Since, a class is considered as a basic unit of testing in object-oriented systems. We are interested in extracting FSM of an Object-Z class, as Object-Z does not provide FSM of a class (Murray et al., 1998). Hence, it is an interesting problem to extract class FSM to facilitate different testing levels i.e. intra-method testing, inter-method testing and inter-class testing (as cited by Huaikou & Ling, 2006). Class FSM is also significant in explicit system behavior representations for implicitly embedded state and operational constraints in abstract Object-Z specifications (Sun et al., 2006).

Following problems are encountered in the existing techniques for FSM generation:

- None of the existing techniques discuss the identification of states and transitions from input and output predicates used in a specification.
- Manual decision making in derivation of states from the generated test templates by utilizing human perception.
- Manual selection of states from partitions obtained from the specifications.
- Manual transition determination for methods in a class using human judgment.
- Coverage of state space regions offered by the relational predicates for defining a variable in terms of constants and other abstract variables.
- Partial consideration of state identification from logical expressions defining pre- and postconditions.
- Scalability problem that arise by re-writing specifications to obtain partitions.
- Lack of automation support for states and transition identification to make FSM.

Some techniques identify states manually using human intuition and decision making, from the generated set of test templates TTs in TTF (Murray et al., 2000), Test Method Templates TMTs in TCF (Huaikou et al.2006) and by partition analysis in (Dick et al., 1993; Hierons, 1997). Transitions are also calculated manually among the set of manually identified states for generating FSM (Murray et al., 2000; Huaikou et al.2006; Dick et al., 1993; Hierons, 1997). Each state with every other state is analyzed manually, if they satisfy the pre- and postcondition of an operation, are considered as pre- and post-states of a transition.

Sun et al., (2005) extracts FSM from Object-Z class specifications by using history invariants as an additional set of predicates. However, history invariants are no more included in recent versions of Object-Z (Smith, 2000). The work of Hierons, (1997) generates FSM by re-writing the Z specification according to predefine rules. Though, for large systems specifications, it might result in scalability problem. An automated approach by Latif et al., (2008), extracts predicates into input and output predicates from the given Z operation schema. However, this approach does not discuss state identification and transition calculation for FSM using these predicates.

Automated FSM extraction from model-based formal specification is, therefore, an important and interesting problem in formal specification-based testing. However, a major challenge in automatic generation of an FSM from model-based formal specifications like Z & Object-Z is the identification and derivation of valid and disjoint pre- and post-states defined by each predicate that identifies pre- and postconditions of all the operations of a class.

Another challenging task for FSM generation is the identification of transitions among the identified states of all operations without involving human intuition. Since model-based formal specification language like Object-Z can not provide FSM of a class (Murray et al. 1998), hence it is vital to derive FSM from Object-Z class specifications to extort valuable information that can be used in testing process.

1.3 Proposed Work

The proposed approach for FSM extraction is built on the existing work by Latif et al. (2008) for the separation of input and output predicates defining pre- and postconditions of a given Z operation schema. However, in this approach the derivation of states from the separated input and output predicates or from the respective pre- and postconditions is not discussed at all.

This research is an initial step towards automatic FSM generation from model-based formal specification. An approach has been proposed for the extraction of states and transitions of FSM from an Object-Z class. States and transitions of FSM are identified from input and output predicates extracted from a given Object-Z class. Algorithms and tool has been developed to support automation for the proposed approach. The approach is proposed for the Object-Z class specifications that only make use of simple data types, simple predicates with relational and some logical operators. To avoid complexities in automation, complex arithmetic expressions and complex data types like sets, sequences are not included yet.

Therefore, the proposed work is an important development towards automated FSM generation and consequently in the area of formal specification-based testing. The approach is also demonstrated on an example given in chapter 3.

1.4 Organization of Thesis

Organization of the rest of the thesis is as follows. Chapter 2 presents the detailed literature review of the related work. Chapter 3 describes the proposed approach in detail. Chapter 4 discusses the automation and tool support. Chapter 5 evaluates the proposed approach. Chapter 6 presents the conclusion and the future work.

2

Literature Review

2. LITERATURE REVIEW

This chapter discusses the related work in detail. Discussion and analysis for different research works for FSM extraction from model-based formal specification languages such as: Z, Object-Z and VDM etc, are given. A tabular comparison for FSM extraction techniques is also presented at the end of this chapter.

2.1 Dick and Faivre's Approach

Dick and Faivre (1993) presented a testing approach for model-based formal specification languages. The approach is used for generating disjoint partitions from formal specifications and then identification of states using these partitions. The testing approach provided by them is based on FSM. They use FSM for sequencing the test cases. VDM specifications are used to demonstrate the approach.

However, the approach is also practical to any other state-based formal specification language. The approach initially transforms each operation in the formal specification into a proposition in predicate logic. The use of partition analysis reduces the specification to disjunctive normal form (DNF) in order to make disjoint partitions of the proposition. Transformation of a disjunction into disjoint components is shown below:

$$B \vee C \equiv (B \wedge C) \vee (\neg B \wedge C) \vee (B \wedge \neg C)$$

This transformation in DNF results in non-overlapping and disjoint partitions. Each of these partitions depicts an independent proposition and hence can be called as a sub-operation. If there is some possibility to make sub-operations simpler, then these are simplified using propositional predicate logic rules. These sub-operations are then used for the separation and identification of the respective pre- and postconditions. The identified pre- and postconditions are again converted into DNF to obtain equivalence partitions that present non-overlapping partitions. The disjoint partitions obtained as a result of this conversion, represent a state. This state can either be a pre-state or a post-state of that sub-operation. States for all the identified sub operations are derived in a similar manner. The states of FSM are the disjoint before and after states of the sub-operations. Each sub-operation identified, corresponds to a transition between a pair of states say 'S_1' and 'S_2'. A sub operation such as 'Op', having pre-state 'S_1' and post-state 'S_2' represents a transition between these two states. To make FSM, this process is repeated to identify all the transitions. The transitions of the FSM are a set of expressions (sub-operations) that are derived from the partition analysis.

Analysis

The concept of partitioning the formal specification into disjoint partitions for the identification of states was first introduced in this work. The approach was demonstrated on VDM formal specifications, which contains separately specified pre- and postconditions. Unfortunately, this approach cannot be applied to extract states and transitions to make FSM from Object-Z class specifications, because in languages like Z and Object-Z pre- and postconditions are not specified separately. However, this approach does not discuss the identification of states and transitions directly from predicates that define operations other than the partitions. Apart from that no discussion has been provided for the state space coverage for regions defined in relational expressions of pre- and postconditions of an operation. Also, the techniques in paper are all implemented except the generation of Finite State Automata and test values.

2.2 Hierons' Approach

Hierons (1997) presented an approach for testing from Z specifications. FSM is derived from partitions generated by applying partition analysis. This approach begins with the classification of variables that are used in the specification. These variables are classified into one of input, output, global, input state, output state and intermediate variables. Once variables are classified, specification is re-written to predicate logic. By re-writing the specification, it is flattened to categorize the predicates into input and output predicates. The predicates that define the input are denoted as input predicates and corresponds to the precondition. The predicates which are involved in defining the output or the final state are denoted as output predicates and such predicates correspond to the postconditions. Specification is once again re-written to the required form using some rules. This re-writing makes every precondition conjunct with its corresponding postcondition. By doing this, whole specification becomes disjunctions of such pairs (having conjunctions). Formally, this form is presented as,

$$\bigvee_{1 \leq i \leq n} (Xi \wedge Yi)$$

'Xi' and 'Yi' represents the pre- and postconditions respectively. Partitions are then identified from this disjunction of pairs. Each generated partition represents a state; hence states are identified from these partitions, where these disjoint partitions correspond to unique states. Once states are identified, transitions are then determined between those states. The combination of a pre-state and a post-state with an operation that evaluates to true, determines a transition for the FSM.

Let 'S_a' and 'S_b' be the pre- and post-states respectively, for an operation 'Op'. If this combination evaluates to true then a transition exist between 'S_a' to 'S_b' labeled by 'Op'. All the identified states and transitions are then mapped to make an FSM. This FSM is then used for the testing purpose.

Analysis

Hierons used Z specification language in which pre- and postconditions are not explicitly given and need to be separated. This work discusses the identification of pre- and postconditions in a specification as well as the generation of FSM. For rewriting of Z specification an algorithm is given that is used for input domain partitioning. Rewriting the specifications for larger systems can increase the size of the representation producing excessive number of sub domains thus leading to scalability problem.

The approach also lacks the details of automation for the identification of states and transitions. Hence, it provides no tool support due to involvement of human intuition and decision making in deciding the status of variables, states and transitions in the process of FSM generation. However, this approach does not discuss state identification from complex data types and from predicates; rather manually separated partitions are being used. Also, this approach does not handle relational expression in terms of variables and constants in the pre- and postcondition of an operation.

2.3 Sun and Dong's Approach

Sun and Dong (2005; Sun, 2006) presented an approach to extract explicit system behaviors as FSM from abstract Object-Z specifications using history invariants. This work is mainly aimed to generate implementable constructs from high-level designs in Object-Z. The work has specially taken into account the state explosion problem that takes place due to the infinite number of states offered by a class.

However, an FSM should consist of finite number of states. Mapping infinite states to finite number of states is a key challenge in FSM generation. These finite states called as abstract states are generated using predicate abstraction. Abstract predicates are being used to determine abstract states as given (Sun & Dong, 2006):

$$Sa \cong \{s \mid \exists X \subseteq AP \bullet s = \wedge(X \cup \{\neg e \mid e \in P \setminus X\})\}$$

'Sa' represents the set of abstract states, 's' represents an abstract state, 'AP' is the set of all predicates, 'X' is the set of those predicates which must be true in state 's' and 'e' is the predicate that must be false in case of 's'. Using ($\neg e$) in definition of 's' ensures that all states are disjoint. After states are derived, transitions are determined. In addition to states abstraction, each operation is also abstracted to identify the transitions. It is determined that from which state the transition invokes, referred to as pre-state and to which abstract state it can lead, referred to as post-state. To decide on those states that satisfy precondition (pre-states) and those states that satisfy the postconditions (post-states), function W(p) and S(p) are defined respectively.

Formally (Sun & Dong, 2006),

$$W(p)=\{s\in Sa \setminus s\Rightarrow p\}$$

$$S(p)=Sa \setminus \{s\in Sa\setminus s\Rightarrow\neg p\}$$

Pre- and post-states are identified for each operation in the specification. For an operation 'Op', having pre-state 'Sa' and post-state 'Sb', the transition will be from 'Sa' to 'Sb' labeled as 'Op'. Once all the transitions are determined, these are mapped along with the abstract states to form an FSM.

The work is extended further to extract FSM from Object-Z specifications along with history invariants. Where history invariants are used to capture liveness properties, which must be true unlike safety properties. A Büchi automaton is extracted from history invariants. Then this Büchi automaton and the product of earlier developed FSM, a raw FSM is acquired. An algorithm is proposed to prune the FSM ensuring that raw FSM fulfills two crucial properties of the open systems. Guard conditions are then determined for transitions where are found necessary and appropriate. This results in the completion of FSM. This FSM is assumed to be the Object-Z realization with history invariant. The approach is supported by a Java based tool that takes XML form of Object-Z specifications along with the abstract predicates and the approach is partially automated.

Analysis

A special consideration has been given to the state explosion problem in the work of Sun & Dong. State explosion even being a central problem in FSM generation has not been considered by other related works. This approach uses history invariants along with that of Object-Z specifications to generate FSM. However, history invariants are no more included in Object-Z specifications (Smith, 2000). This approach does not consider state identification from arithmetic and relational expressions, when used in pre- and postconditions of operations of Object-Z specifications. State identification from the predicates defining pre- and postconditions of operations is not considered as well.

2.4 Carrington et al. (2000; revised 2003)

A valuable contribution has been added by the researchers at the University of Queensland in the area of formal specification-based testing over last few years. They presented a specification-based testing approach for Object-Z class specifications. The approach demonstrated the usage of Object-Z from test case generation to execution and evaluation. The approach consists of different phases in order to complete the testing processes, which are explained below:

2.4.1 Test Template Framework (TTF)

Test Template Framework (Stocks & Carrington, 1996) is used for test templates generation from an operation's Object-Z specification. TTF provides a stepwise guidance for the generation of abstract test cases. However, originally the TTF was demonstrated on Z specification but the authors claimed its applicability on any other formal specification. That is why; it was later used for the Object-Z specifications. A test template (TT) generated by TTF is basically a Z schema representation of test data. An original specification of operation is also regarded as a TT but at most abstract level it is called Valid Input Space (VIS). To obtain finer TTs from VIS further, a testing strategy is applied on VIS. By applying testing strategy a subset of parent TT is produced including multiple TTs and is more refined than their parent. This process can be repeated unless desired level of test data is attained. Final TTs obtained represent the final test data derived from the specifications. These TTs when are obtained from Output Space, are named as output templates (OT) which are treated as oracles. The process for the generation of TTs is repeated for every operation in the class to obtain a set of TTs for the whole class. Once all TTs are obtained next phase for FSM generation is applied on them.

2.4.2 FSM Generation from TTs

FSM is generated from the TTs obtained by TTF for a class under test. The initial state of the FSM is obtained ~~from the~~ INIT schema of the class where, the rest of the states are obtained from the final set of test templates (TTs) and output templates (OTs). To derive the states of FSM, state templates are generated from TTs and OTs by using schema hiding. Schema hiding, hides the input and output variables from TTs and OTs respectively. This hiding is done only to show state variables in TTs and OTs. State templates (STs) obtained as a result of schema hiding may involve duplicate and over-lapping state templates. These over-lapping and duplicate STs are removed to get disjoint and distinct state templates. These disjoint state templates correspond to the states of FSM. Once states are obtained, transitions are identified between the states. For finding the transitions, operations that map between a pair of states templates are identified. Transitions for all the operations are identified in the similar manner. For each transition, the state template corresponding to the precondition is said to be its source state and the state template corresponding to the postcondition is its target state. When source and target states are identified for all the transitions, FSM is generated from them.

2.4.3 ClassBench Tool

ClassBench framework (Hoffman & Strooper, 1997) generates and executes the test cases for testing purpose. ClassBench conventionally takes the input from implementation. However, this approach utilizes ClassBench to take the inputs that are generated from the specifications. Three main phases of ClassBench tool are given below:

a) Test Graph Generation

FSM is used to generate test graphs which represent a subset of FSM. The testing from test graphs was first introduced by Hoffman and Strooper (1995). A test graph only consists of those states and transitions from FSM, which are to be tested. The choice of states and transitions in the test graph is purely depends on the tester. Since ClassBench generates test suites from the test graph, so it is important to include only those states and transitions in the test graph that need to be tested.

b) Test Oracle Generation

Test oracles are generated using the Object-Z specifications of the class under test. Object-Z specifications have been treated as passive oracle. Therefore, independent oracle class is generated from the specifications. The use of Object-Z specification independently to construct a ClassBench oracle class has been provided by McDonald and Strooper (1998).

c) Test Case Execution and Evaluation

ClassBench tool takes three inputs: generated test graph, test oracle and implementation of the class under test. Once the input is given, this automated tool generates, executes and evaluates the test cases for testing the given class.

2.4.4 Automation

This approach is not fully automated. TTF was first automated by a tool named as TinMan (Murray, Carrington, MacColl, Strooper, 1999). TinMan tool automates the test templates (TTs) generation from the Object-Z specifications by selecting a predicate that satisfies the Valid Input Space (VIS) and a testing strategy from a pre-defined set of testing strategies. This restriction for selection of testing strategy was one of the limitations of TinMan. To overcome this limitation Ashraf and Nadeem (2006) enhanced TinMan to accept the desired testing strategy specified in Object-Z specifications.

Using a type checker for Object-Z specification named as Wizard (Johnston, 1996), a tool was built for the oracle class generation (Carrington et al., 2000). Whereas, the ClassBench tool that generates executes and evaluates test cases is also an automated tool.

Analysis

This approach is a collection of phases to conclude the testing from the specifications. However, each of the sub approach has its own set of limitations and thus corresponds to separate research work. These limitations are also applicable to the Carrington's approach as well. TTF that was formerly used to generate test templates from Z specification was extended for Object-Z Specifications. Since Object-Z is a variant of Z language, hence this extension does not require much exertion. However, certain issues may rise if TTF has to be used for formal specifications other than Z and Object-Z.

Although it is the most automated approach in specification-based testing but still it is not fully automated approach. This approach does not discuss state identification from the pre- and postconditions composed of arithmetic and logical expressions. However, when relational expressions are used, only predicates defined by abstract variables are considered. Identification of states and transitions for FSM is also a manual process. State templates from the derived test templates are selected manually and requires human hunch, involving manual decision making for the selection of non-overlapping set of states. Among the final set of manually identified states, transitions are also identified using human intervention. This manual process and human intuition endows with a hindrance in automating the process of FSM generation from Object-Z class specifications.

2.5 Huaikou and Ling's Approach

An approach to formalize the class testing process from Object-Z specifications was proposed by Huaikou and Ling (2006). They proposed a framework named as Test Class Framework (TCF). According to Binder (2000) there can be different levels of class testing. Harrold and Gregg (as cited by Huaikou & Ling, 2006) suggested testing independent class on three levels (see Binder, 2000) i.e. inter-method, intra-method and inter-class. TCF focuses only on two types of testing levels i.e. intra-method and intra-class testing levels.

Testing each method independently that is encapsulated in the class, is called intra-method testing. TCF testing for intra-method is a slight variation of an earlier framework (Ling & Huaikou, 2000) for Z operation testing.

Since intra-method testing of Object-Z class has similarity with that of testing an operation in Z specification. For intra-method testing in TCF, functional specification of the method under test is denoted by Test Space (TS). Test method templates TMTs are generated according to a defined criteria. TMTs represent a finer level of Object-Z specifications then that of TTs. This process of refinement can be repeated on TMTs until required level of refinement is achieved. To obtain concrete test cases from abstract TMT test cases, TMTs need to be instantiated to specific values. These concrete test cases are named as instant templates (IT). Method Testing Adequacy Function (MTAF) is used optionally to check whether TMTs satisfy certain adequacy criterion.

For the second level of testing i.e. intra-class testing, TCF was first proposed by Ling, Huaikou, & Xuede (2000). It was refined by Huaikou & Ling (2006). In intra-class testing, different valid sequences of method calls need to be tested. This testing of sequences of method calls is achieved by modeling the class under test with a FSM. A state is characterized by a state template (ST). Input and output variables from INIT schema are hidden to form ST corresponding to the initial state. Once all the STs are identified, possible transitions are determined between each pair of STs. For a pair of STs, if such a TMT exits, whose precondition corresponds to one ST and postcondition corresponds to the other ST, then such a TMT becomes a transition among these STs. While the ST that satisfies the precondition of TMT becomes the source state and ST that satisfies the postcondition of TMT becomes the target state. When all the transitions are identified, are mapped along with respective pre- and post-states to form an FSM. After the FSM is produced, different sequences are selected that are needed to be tested. This selection of sequences is made according to some criteria. A Test Class (TC) consists of one such sequence of methods. Hence, a TC represents a full test specification of a test case. Class Testing Adequacy Function (CTAF) is used to formalize the process of generation of TCs from TMTs.

Analysis

Identification of states from the generated set of TMTs is a manual process, and requires human intuition for selecting the disjoint set of states. Transitions among manually selected state pairs are also derived manually for the operations in an Object-Z class.

This approach does not discuss state identification from logical expressions and predicates that define pre- and postconditions of operations. Complex data types are also not handled in this work. However, in relational expressions, only predicates that define regions in terms of constants are considered but the ranges defined by the predicates in terms of abstract variables and constants are not discussed at all. Automation of TCF is not discussed at all and consequently no tool support is provided as well.

Table 2.1 presents the comparison between different FSM extraction techniques from model-based formal specification languages. Table 2.2 shows the abbreviations used in the comparison table. The parameter "Specification language used" represents the model-based formal specification language for demonstrating the particular approach; "Scope/Purpose" indicates the purpose for proposing the approach.

"State Derivation" parameter is divided into five sub-parameters having same parameter values i.e. "Arithmetic Expression" (A.E) parameter depicts the state identification from the arithmetic expressions used in the specifications. This parameter is assigned "Yes" (Y) if the approach is deriving states from such expressions, "No" (N) for not handling the expressions, "Partial" (P) for partially handling the Arithmetic expression. "Logical Expression" (L.E) parameter is used to show whether the approach generates states from logical expressions. "Relational Expression" (R.E) represents whether states are identified from relational expressions containing simple data types like integers, natural numbers etc, for defining a particular variable. This parameter is further divided into three types: i.e. "Defined by Constants", "Defined by Multiple Variables", "Defined by Constants and Variables". Relational Expression that is composed of predicates defining a variable in terms of constants comes under the parameter "Defined by Constants". Predicates in relational expressions which define a variable in terms of constants and other variables come under the parameter "Defined by Constants and Variables". The parameter "Defined by Multiple Variables" includes predicates that define a variable in terms of multiple variables. "Complex Data Types" (sets & sequences) parameter is used to evaluate the approaches that considers such data types. This parameter is assigned "Y" if the approach identifies states for such data types and "N" otherwise.

The need of automation is significant in formal specification-based testing. Therefore, the parameter "Automation/Tool Support" has been given. This parameter has been divided into two sub-parameters: "States Identification" (S.I) and "Transition Calculation" (T.C). These two sub-parameters represent the two actual steps needed for FSM generation. "Y" in "Automation" column shows that the approach provides automation support, "N" means no automation support exists and "P" means some automation support exists but manual work out is also involved.

Table 2.1 Comparison of FSM Extraction Techniques from Formal Specification

Approach	Specification Language Used	Scope/ Purpose	State Derivation							Automation /Tool Support	
			A.E	L.E	R.E			Complex Data Types	IP & OP	S.I	T.C
					Defined by Constant	Defined by Multiple Variables	Defined by Variables & Constants				
Carrington et al. (2000)	Object-Z	Test case Generation & Execution	N	N	N	Y	N	P	N	P	N
Dick & Faivre (1993)	VDM	Test case Generation & Execution	Y	P	N	Y	N	P	N	Y	N
Huaikou & Ling (2006)	Object-Z	Test case Generation & Execution	P	N	Y	N	N	N	N	P	N
Hierons (1997)	Z	Test case Generation & Execution	Y	P	Y	N	N	N	N	N	N
Sun & Dong (2006)	Object-Z with History Invariants	Synthesizing Implementa-ble Constructs	N	P	N	N	N	P	N	P	P

Table 2.2 Abbreviations used

A.E	Arithmetic Expression
L.E	Logical Expression
R.E	Relational Expression
IP & OP	Input Predicate & Output Predicate
S.I	State Identification
T.C	Transition Calculation

2.7 Conclusion

It is apparent from the literature survey that FSM has proved its significance not only in testing approaches but also in determining low level implement-able constructs from abstract formal specifications. Therefore, automated FSM construction from formal specifications has emerged as a necessary and fundamental need in specification-based testing.

None of the approaches for FSM generation from model-based formal specification identifies states and transitions from the input and output predicates that define pre- and postconditions of operations. Relational expressions that are composed of predicates using both constants and variables are also not considered.

Some approaches for FSM extraction generates states manually by quantifying the predicates applying certain testing strategy. The process of refinement to obtain desired level of test data, conversion of the templates into states, removal of duplicate and over-lapping states is a manual process and requires human effort and verdict. The process of determining transitions by mapping an operation to the states satisfying its precondition and postconditions is also manually performed and requires human intuition.

Other approaches generate FSM by simplifying the operations in given specifications according to some criteria in order to generate desired level of disjoint partitions. However, this simplification level of partitions and generation of states and transitions from these partitions requires manual effort and judgment.

3

Proposed Approach

3. THE PROPOSED APPROACH

FSM generation from a model-based formal specification is an imperative problem as discussed in the literature survey. Identification of states and transitions from input and output predicates facilitates in generation of FSM of an Object-Z class. However, as noted in the previous chapter, the existing techniques for FSM construction from a formal specification are not fully automated.

This chapter presents an automated approach proposed for FSM generation from an Object-Z class specification. Section 3.1 gives a brief overview of the proposed approach and the rest of the sections discuss the approach in detail.

3.1 Introduction to the Proposed Approach

This section details the proposed approach for derivation of states and transition calculation of a class FSM from its Object-Z specifications.

In model-based formal specification language such as Object-Z, pre- and postconditions are implicitly expressed, unlike in VDM (with separate pre- and postconditions). Hence, to identify states of FSM from pre- and postconditions, it is required to separate pre- and postconditions and input and output predicates defining them. Since Object-Z can not provide a Class FSM (Murray et al. 1998), therefore generation of FSM from an Object-Z class is a necessary and interesting problem.

However, generation of FSM from Object-Z specifications is not a simple task due to the involvement of expressions, complex data types and abstract predicates fors defining class operations. Predicates that define variables in terms of some constant gives a concrete range of value that it offers. Some predicates might define the value of a variable in terms of other abstract variables with unknown values. Identification of states for such variables is even more difficult task for defining simple yet abstract predicates since the value of variable is unknown. Another important task is to generate states for the pre- and postconditions that are constrained by predicates that define multiple variables.

Our approach is built on an existing work by Latif & Nadeem (2008) for the separation of input and output predicates and extraction of pre- and postconditions from these predicates for a given Z operation schema. This approach is also supported by a tool "TEIOPZ" (Tool for extraction of input and output predicates from Z specification). Abstract view of TEIOPZ is shown in figure 3.1. However, this work does not provide any discussion for the FSM generation from the separated input and output predicates or from the extracted pre- or postconditions constituting these predicates. We have built on this work not only to generate FSM from these separated input and output predicates, but also to accommodate object-oriented features provided by Object-Z.

The proposed approach is an initial attempt towards automatic FSM generation from model-based formal specification. The approach is proposed for the specifications that define operations by using simple data types and simple predicates with relational and some logical operators. To avoid complexities in automation, arithmetic expressions and complex data types like sets, sequences are not included yet.

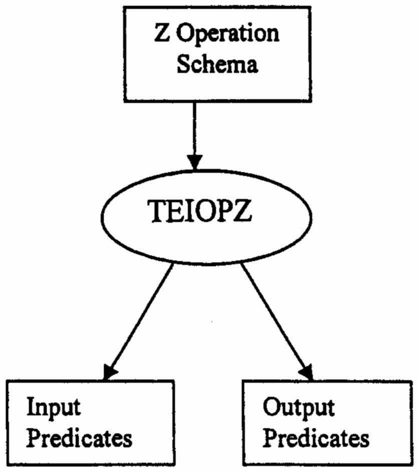


Figure 3.1 Abstract View of TEIOPZ (Latif et al., 2008)

Although TEIOPZ is meant for the separation of Input Predicates denoted as 'IP' and Output Predicates denoted as 'OP' from a given Z operation schema but we have utilized it to separate IP and OP for all operations in an Object-Z class. However, there is a remarkable difference in the Z and Object-Z operations. An Object-Z operation is disabled outside its precondition, unlike Z in which precondition violation gives an undefined result (Murray et al., 2006).

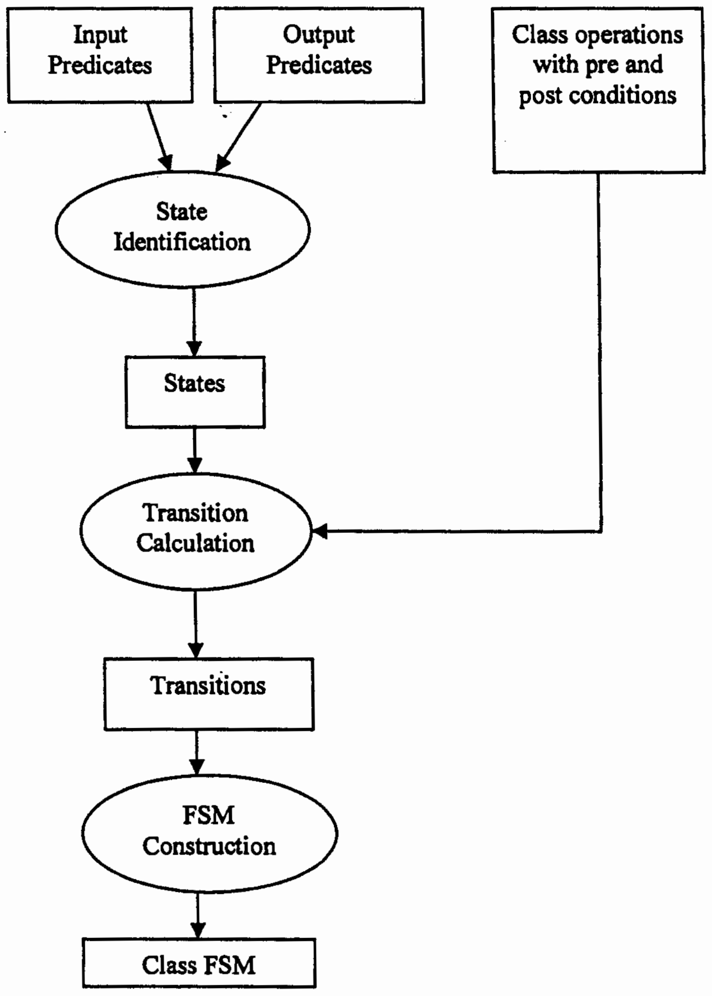


Figure 3.2 Flow chart of the proposed approach

Important steps in FSM generation from Object-Z class specifications are identification of pre- and post-states for all operations in that class followed by the transition calculation among the identified states. These states and transitions then collectively make an FSM. Hence, the proposed approach is partitioned into three main phases: State identification phase, Transition calculation phase, and FSM construction phase. Figure 3.2 presents the flow chart of the proposed approach. The ovals represent the processes and the rectangles represent development artifacts.

3.2 State Identification Phase

In this phase, states are identified for the FSM of an Object-Z class. From the class initial (INIT) schema, and each operation's IP and OP, we derive the states of the class FSM.

Pre-states can be identified from preconditions and the corresponding post-states can be identified from the postconditions. IP and OP that defines precondition (entry-level condition) and post-condition (exit condition) respectively, hence are used in this phase, as a basis to identify states of an Object-Z class.

To identify states from the separated IP and OP this process is composed of three sub processes: Predicate Categorization, Sequenced List Formulation and State Generation as discussed in detail in the following sub-sections.

3.2.1 Predicate Categorization

Predicate categorization is the first sub-process of the state identification phase. It categorizes input and output predicates that are extracted from Object-Z class specifications into defined categories of predicates. Each predicate category corresponds to the predicates with discrete nature and definition. Algorithm for predicate categorization process is shown in figure 3.3. Predicate categorization details are discussed below:

a) Categorization of IP and OP into Pv

This process begins with the identification of IP and OP that are defined by similar variable say 'v' in the specification. These predicates are grouped together into a new predicate list denoted as 'Pv' and this process is repeater unless each predicate in IP and OP becomes a part of its respective 'Pv' list. As a result, 'Pv' list contains all the predicates with similar identifier name. The number of 'Pv' lists obtained from a class specification depends upon the number of variables used in it.

If (IP U OP) represents the set of all predicates say 'P_a' that defines set of all variables say 'V_i' used in the specification then each variable in 'V_i' contains its own set of predicates P_{Vik} defining that particular variable. Given,

$$\{P_a \mid IP\ U\ OP\} \text{ where } \{1 \leq a \leq n\}$$
$$\text{and } \{V_i \mid 1 \leq i \leq m\}$$
$$P_{Vik} = \{p_k \mid p_k \in P_a, P_k \text{ contains } V_i\} \text{ where } \{ k \subseteq \{1 \ldots. n\} \wedge k \neq \{\}\}$$

b) Categorization of Pv into Predicate Categories (PCs)

Once all the Pv lists are obtained for Vi, the predicates in each 'Pv' list are examined further to place these predicates into predicate categories defined to make them appropriate for Sequenced List Formulation process in section 3.2.2. For this predicate categorization a set of rules are proposed which are illustrated below:

Predicate Category 1. For a predicate 'p' in 'Pv' which defines a variable say 'v' on L.H.S of standard infix relational operator such as: less than '<', less than or equal to '≤', greater than '>', greater than or equal to '≥' and constant on its R.H.S are denoted as 'Pvc', where the predicate category is named as 'PCvc', Formally,

$$PCvc = \{Pvc \mid Pvc \in Pv, Pvc \text{ contains } (v \wedge rel \wedge c)\}$$

Where 'c' denotes constant and 'rel' denotes relational operators like: less than '<', less than or equal to '≤', greater than '>', greater than or equal to '≥'.

Discussion: Each predicate category of PCvc contains all those predicates that define similar variable in terms of constants and standard operators. Since standard operators have regions upon which their behavior is expected to be uniform (Hierons, 1997). Hence, each Pvc predicate provides us with the valid region for the variable say 'v_1' defined by it. Thus, all predicates in PCvc category define all possible regions for variable 'v_1' in which its behavior is expected to be uniform for a given class specification. These defined regions are converted into valid sequenced lists in next section 3.2.2 that guarantees generation of valid and disjoint states from it to cover all the state space that might be offered by variable 'v_1' in a class specification as explained in detail in section 3.2.2.

```
Algorithm: CatPredicate
Input:     IP,OP:Set of Predicates;
Output:    PCvc, PCvmv, PCvei: List of Predicates;
Declare:   P: List of all Predicates, rel-oprtr:: infix relational operators used in predicates, PCvc: List of predicates with
           similar variable name defined by constants, PCvmv: List of predicates containing multiple variables, PCvei: List
           of predicates with equal to and not equal to operator.Union(): combines the elements of two sets, append():
           add elements to the list, remove():remove elements from the list
Begin: CatPredicate
1.    P=Union (IP, OP)
2.      For (Every p in P) do
3.          Select all p in P with similar variables on L.H.S of rel-oprtr
4.          Pv=append (Pv, p)
5.          P=remove (P,p)
6.      End For
7.      For (every pv in Pv) do
8.          If pv contains variable on L.H.S and Constant on R.H.S of rel-oprtr then
9.             PCvc = append (PCvc, pv)
10.            Pv=remove (Pv,pv)
11.         End If
12.         If pv in Pv contains variable on L.H.S and R.H.S of rel-oprtr then
13.            PCvmv = append (PCvmv, pv)
14.            Pv=remove (Pv, pv)
15.         End If
16.         If (pv in Pv contains '=' or '≠') then
17.             PCvei = append (PCvei , pv)
18.             Pv=remove (Pv, pv)
19.         End If
20.     End For
21.   Return PCvc, PCvmv, PCvei
End CatPredicate
```

Figure 3.3 Categorize Predicate Algorithm

Predicate Category 2. If a predicate 'p' in 'Pv' defines a variable 'v1' in terms of another variable say 'v2' with the help of standard infix operator such as: less than '<', less than or equal to '≤', greater than '>', greater than or equal to '≥', such a predicate is denoted as 'Pmv', and is added to the predicate category 'PCvmv'.

Formally,

$$PCvmv = \{Pmv \mid Pmv \in Pv,\ Pvc \text{ contains } (v1 \wedge rel \wedge v2)\}$$

Discussion: Each predicate category of PCvmv contains all such predicates that define a variable say 'v_1' in terms of another variable 'v_2' using standard infix operators. In such a case standard operators define abstract regions for 'v_1' whose possible values are unknown. For that reason 'v_1' is not defined in terms of some constant 'c' but in the terms of another variable 'v_2' whose value is unknown yet. Hence each predicate Pmv provides valid yet unknown region for 'v_1' defined by it. Thus, all Pmv in category PCvmv defines all possible undefined and unknown regions for 'v_1' in which its behavior is expected to be uniform for a given class specification. These abstractly defined regions in PCvmv are manipulated further in section 3.2.3 to make valid sequences that guarantees generation of disjoint states from them by covering all the possible state spaces that might be offered by 'v_1'.

Predicate Category 3. If 'Pv' contains a predicate 'p' that defines a variable 'v_1' interms of the standard infix operators such as: equal to '=' and not equal to '≠' and another variable 'v_2' or a constant 'c' then such predicates are added to predicate category 'PCvei'.

Formally,

$$PCvei = \{Pei \mid Pei \in Pv,\ Pei \text{ contains } v1 \wedge (= \vee \neq) \wedge (v2 \vee c)\}$$

Discussion: Each list of category PCvei obtained from Pv contains predicates that equates the variable 'v_1' in a predicate Pei to some constant 'c' or another variable

'v_2'. It also includes the predicates that provides inequality between a variable say 'v_1' and another variable say 'v_2' or a constant 'c'. In each case Pei defines a particular value of 'v_1' either a known value in terms of 'c' or an unknown value in terms of 'v_2'. The value of 'v_1' defined by Pei might cause duplication or over-laping with the values that the regions might hold. To overcome that issue we apply a set of rules to adjust such predicates in the regions and in state generation process for a variable as explained in section 3.2.3.

By the end of this process each 'p' in 'Pv' is placed into categories PCvc or PCvmv or PCvei. This process is repeated unless predicates in all the 'Pv' lists obtained from specifications are categorized in one predicate category or other and all 'Pv' lists become empty. Empty lists ensure that all the predicates in each 'Pv' list, used in a given specification are arranged into the appropriate categories: PCvc, PCvmv and PCvei. Such organization is necessary for making appropriate sequenced lists resultantly as explained in section 3.2.2. Algorithm for Categorize Predicate is shown in figure 3.3.

3.2.2 Sequenced List Formulation

In this process valid sequenced lists are generated from predicate categories obtained previously. Valid sequences are formulated to ensure the generation of valid and non-overlapping states from these lists as explained in section 3.2.3. The algorithm for Sequence List Formulation process is shown in figure 3.4.
Separate criteria are applied for formulation of lists depending on the number of predicates contained in categories PCvc and PCvmv extracted from each Pv List. Some of the predicates that contain constants as in PCvc clearly indicate the state boundaries but for those that contain abstract variables on each side of operator, it is difficult to imagine boundary or range offered by the variable used. Four types of possible alternatives that might occur depending on the type of predicate list are discussed below:

List Formulation Type 1. If the predicates in 'Pv' are categorized into 'PCvc' only, it indicates that all the predicates defining variable 'v' has defined regions.

It is necessary to sort these regions into a valid sequence so as to generate valid states from them. For predicates containing constants like in PCvc, only one valid sequence can be formulated say 'Lc' and that is by sorting its predicates in ascending order on the basis of constants used in all predicates. Reason for generating one sorted list only and not finding its permutations is that, that re-arranging the predicates in PCvc to find other sequences might result in invalid sequences, which can cause invalid state generation.

Formally,

$$Lc=\{PCvc \mid sort(PCvc)\}$$

TH6982

List Formulation Type 2. If the categorization of predicates in 'Pv' results in PCvmv only and number of predicates greater than 1. That means all the predicates defining that variable say 'v' has abstract regions with unknown values. In such a case, it is necessary to make Lists with all possible sequences that may occur denoted as 'Lmv_i'. Since value of variable 'v' can lie in any of the possible combination of the regions. For that reason, we make n factorial (n!) sequences of the predicates in PCvmv.

Formally,

$$Lmv_i = \text{ith permutation of } PCvmv,$$
$$\text{Where } i=\{1, 2, 3 \ldots n!\} \text{ and } n \geq 1$$

List Formulation Type 3. Predicates in 'Pv' if are categorized in categories PCvmv and PCvc then it leads to the following possibilities:

- If the number of predicates in PCvc is greater than or equal to 1 and number of predicates in PCvmv is equal to 1, then (n) number of sequenced lists are formulated denoted as $Lmvc_i$. These lists cover all the possible and valid sequences of the predicates both from PCvmv and PCvc.

  Formally,

$$Lmvc_i = \text{ith permutation of } PCvmv \cup PCvc$$
$$\text{Where } i=\{1, 2, 3 \ldots n\} \text{ and } n \geq 1$$

```
Algorithm: FormSeqLists
Input:     PCvc, PCvmv
Output:    Lists L
Declare:   sort(): Sorts the predicates in List P in ascending order on the basis of constants involved, Pmvct: Contains
           sorted PCvc and PCvmv, append(): adds element to the list, add(): Adds a list to the set of lists
Begin: FormSeqLists
1.        If (#PCvc=0 ) ∧ (#PCvmv =0) then
2.            Error & exit
3.        End If
4.        If (#PCvc>1) ∧ (#PCvmv=0) then
5.            L= PCvc.sort ()
6.            Else If ((#PCvc = 1) ∧ (#PCvmv≥ 1)) OR ((#PCvc = 0) ∧ (#PCvmv>1)) then
7.                    PCvc.sort()
8.                    Pmvc =append (PCvc, PCvmv)
9.                    Make Lists of Pmvconst
10.                   Where # L =n!
11.                   L= Add (L)
12.           End If
13.           Else If ((#PCvc>1) ∧ (#PCvmv=1)) then
14.                   PCvc.sort ()
15.                   Pmvconst =append (PCvc, PCvmv)
16.                   Make Lists of Pmvconst
17.                   Where # L =n
18.                   L= Add (L)
19.           End If
20.           Else If ((#PCvc>1) ∧ (#PCvmv>1)) then
21.                     PCvc.sort ()
22.                     Pmvconst =append (PCvc, PCvmv)
23.                     Make Lists of Pmvconst
24.                     Where # L = n-2*n-1*.....
25.                     L= Add (L)
26.           End If
27.       End If
28. Return L
End FormSeqLists
```

Figure 3.4 Sequenced List Formulator Algorithm

- If the number of predicates in 'PCvc' is equal to 1 and number of predicates in 'PCvmv' is greater than 1, then (n!) number of sequenced lists are formulated denoted as '$Lmvc_i$'.

  Formally,

  $$Lmvc_i = \text{ith permutation of } PCvmv \cup PCvc$$
  $$\text{Where } i = \{1, 2, 3 \ldots . n!\} \text{ and } n \geq 1$$

- If the number of predicates in PCvc is greater than 1 and number of predicates in PCvmv is also greater than 1 and PCvc contains predicates greater than or equal to that of PCvmv. For its sequenced lists, sorted PCvc is appended with PCvmv in such a manner that sorted sequence of

PCvc is not disturbed in any of the new sequence with PCvmv to evade generation of invalid states from them. For such a case possible number of sequenced lists formulated is $(n-1)^2$. Otherwise the number of possible permutations will be (n-2!).

Formally,

$$Lmvc_i = \textit{ith permutation of } PCvmv \cup PCvc$$
$$\textit{Where } i = \{1, 2, 3 \ldots (n-1)^2\} \textit{ and } n \geq 1$$

It is notable here that the sequences are formulated in such a way that they only result in producing valid and disjoint set of states in the state generation process. It is thus essential to formulate lists in which the sorting order is maintained because we are interested in capturing only the valid state space offered by the predicates in PCvc and PCvmv as shown in Formulate Sequenced List algorithm in figure 3.4.

3.2.3 State Generation Process

List of lists 'Lc_i', 'Lmv_i', '$Lmvc_i$' obtained from Sequenced list formulation process ensures all the valid and disjoint sequences of the predicates that define variable 'v'. Let 'L_i' includes collective set of all sequenced lists 'Lc_i', 'Lmv_i', '$Lmvc_i$' obtained for all predicates 'P_i' defining all variables 'v_i' used in the specification. Hence, 'L_i' for a class specification provides all the possible regions of state space that class data members might offer. The State Generation Algorithm is shown in figure 3.5. States are now generated from each of the sequenced list obtained as explained below:

A set of rules is also proposed for state identification among the two consecutive predicates in each sequence list 'L'. These rules are explained below:

Rule 1: For a sequenced list 'L', if the first predicate 'P_i' in it includes standard infix relational operator like: greater than '>' or greater than or equal to '≥', this predicate is negated to convert it to less than or equal to '≤'or less than '<'

respectively. The negated 'P_i' is added to set of states denoted as 'S' as a state say 's'. If not, then P_i without any change, as a state's' is added to 'S'.

Formally,

$$\begin{gathered} \textit{If } P_i \in L_i \textit{ includes} > \textit{or} \geq \textit{then} \\ \sim P_i \in S \\ \textit{else } P_i \in S \end{gathered}$$

Where 'L' denotes a list from all sequenced lists 'L_i' and $i \geq 1$

A state 's' obtained by above rule captures the initial values of state space region for 'v_i' if not defined by the first predicate P_i that contains '>' or '≥'. Other wise in case of no change when P_i contains '<' or '≤', the state obtained by this predicate do provide that initial state space region and hence P_i is declared as 's' without any change.

Rule 2: If the last predicate say 'P_n' in the given List sequence 'L', contains standard operator like: greater than '>' or greater than or equal to '≥', then P_n is added to 'S' as a state's'. Otherwise, if P_n contains'<' or '≤' then negation of P_n is take to inverse its operators to greater than '>' or greater than or equal to '≥' and are added to 'S' as a state's'.

Formally,

$$\begin{gathered} \textit{If } P_n \in L_i \textit{ includes} > \textit{or} \geq \textit{then,} \\ P_n \in S \\ \textit{else} \quad \sim P_n \in S \end{gathered}$$

Where 'L_i' denotes a List from the List of sequences, and i= {1, 2... n}.

A state 's' generated by using this rule covers the last region for state space of variable 'v' if defined by P_n in L_i. If this is not the case then this P_n is converted into a form so as to cover the region containing last values defined by $\sim P_n$ and is added to 'S' as a state 's'.

Rule 3: Two consecutive predicates say P_i and P_{i+1} in list L are compared on the basis of the infix relational operator defining region of each predicate. Comparison begins from the first predicate P_i up till the last predicate say P_n in 'L'. We compare two predicates at a time and apply a set of rules depending on the operator used in each predicate to cover the intermediate region defined by the two predicates as explained below:

- If P_i defines a region for 'v' using '>' or '≥' and P_{i+1} defines a region for 'v' using '<' or '≤', then for capturing the intermediate and valid state space region between these two predicates, only conjunction of the two predicates is taken to generate a state 's' from them.
  Formally,

  If P_i, $P_{i+1} \in L_i$ *and* P_i *includes* $>$ or $\geq$ and P_{i+1} includes $<$ *or* $\leq$, *then*

  $$P_i \wedge P_{i+1} \in S$$

- If P_i defines a region for 'v' using $<$ or $\leq$ and P_{i+1} defines a region for 'v' using $<$ or $\leq$. For generating their intermediate and valid state space region, $\sim P_i$ is conjunct with P_{i+1} to generate a valid and non overlapping state 's' from them.
  Formally,

  If P_i, $P_{i+1} \in L_i$ *and* P_i *includes* $<$ *or* $\leq$ and P_{i+1} includes $<$ or $\leq$ *then,*

  $$\sim P_i \wedge (P_{i+1}) \in S$$

- For P_i defining a region for 'v' using '<' or '≤' and P_{i+1} defining a region for 'v' using '>' or '≥'. For generating their intermediate and valid state space region requires conjunction of $\sim P_i$ with $\sim P_{i+1}$. Resultant state will be a valid and disjoint state 's'.

Formally,

If P_i, $P_{i+1} \in L_i$ and P_i includes $>$ or $\geq$ and P_{i+1} includes $<$ or $\leq$ *then,*

$$\sim P_i \wedge \sim (P_{i+1}) \in S$$

- If P_i defines a region for 'v' using '>' or '≥' and P_{i+1} defines a region for 'v' using '>' or '≥'. The intermediate and valid state space region between these two predicates will be captured by taking conjunction of P_i with $\sim P_{i+1}$. Resultant state will be a valid non-overlapping state's'.

Formally,

If P_i, $P_{i+1} \in L_i$ and P_i includes $>$ or $\geq$ and P_{i+1} includes $>$ or $\geq$ then

$$P_i \wedge \sim P_{i+1} \in S$$

Applying this rule ensures that intermediate regions defined by the predicates in list L are converted into valid, disjoint and non-overlapping state space regions, being offered by the predicates. This state space generated by Rule 3 will include regions other than those covered by the first and last states generated by Rule 1 and 2.

State generation process ensures that all the possible, disjoint, valid and non-overlapping state space regions are covered and assigned to respective states. These regions are defined for each list 'L' formulated for a variable 'v'. Hence, the states generated by this process will be covering all possible state space regions defined by all Lists L_i for each variable 'v_i' used in the specification.

If a predicate category PCvei is empty for variable 'v' then 'S' generated includes the final set of states. Otherwise, if PCvei is not empty then the predicates in PCvei must be over-lapping with, some value of any of the state space region covered by a state's'. To overcome that problem following rules are applied to make the states disjoint.

Rule 1: If a predicate Pei in predicate category PCvei defines a variable 'v_1' on L.H.S in terms of equal to operator '=', another variable 'v_2' or constant 'c' on

R.H.S, than predicate Pei is added to S as a state 's'. We need to negate this value offered by 'v_l' from particular state space region including it. For this reason, ~Pei is conjunct with the state's' which contains its value.

Formally,

If Pei $\in$ PCvei and includes '=' and Pei $\subseteq$ s, then

Pei $\in$ S and

~Pei$\wedge$s (replacing previous s) $\in$S

```
Algorithm StateGen
Input:      Lists L
Output:     SetofStates S
Declare:    P: List of Predicates, PCvei: List of predicates containing equality and non-equal to operator,
            GetAllPredinList(): Gets all predicates included in each List L, add(): adds a state to set of states
Begin StateGen
1.    State S= null
2.      For (every l in L) do
3.      P= L.getallpredinList()
4.              If ($p_i$ in P contains > or ≥) then
5.                S.Add (~P)
6.                  Else S.Add (P)
7.              End If
8.              If ($p_n$ in P contains < or ≤) then
9.                S.Add(~P)
10.                 Else S.Add (P)
11.             End If
12.     For (every p in P) do
13.             If (p contains < or ≤) $\wedge$ (p++ contains < or ≤) then
14.               S.Add (~p $\wedge$ p++)
15.                  Else If (p contains < or ≤) $\wedge$ (p++ contains > or ≥) then
16.                        S.Add(~p $\wedge$ ~p++)
17.                  End If
18.                  Else If (p contains > or ≥) $\wedge$ (p++ contains < or ≤ ) then
19.                        S.Add(p $\wedge$ p++)
20.                  End If
21.                  Else If (p contains > or ≥) $\wedge$ (p++ contains > or ≥) then
22.                        S.Add (p$\wedge$ ~p++)
23.                  End If
24.                  End If
25.     Repeat until $p_n$ in P
26.     End For
27.     For (every p in PCvei) do
28.       Check each PCvei with every s in S
29.             If (P contains '=') $\wedge$ (P $\subseteq$ of S )
30.               Then S= (S $\wedge$~ PCvei)
31.             End If
32.     End For
33.               Add to S
34.             End for
35.   Return S
End StateGen
```

Figure 3.5 State Generation Algorithm

Rule 2: If a predicate Pei in predicate category PCvei containing a variable 'v_1' on L.H.S of not equal to operator '≠', with another variable 'v_2' or constant 'c' on L.H.S, then we only conjunct this Pei without any change with the respective region covered by 'v_1' from the set of states 'S'. For this reason we conjunct Pei with the state 's' with which it makes a subset to.

Formally,

If Pei ∈ PCvei and includes '≠' and the value in Pei ⊆ s, then

Pei∧s (replacing previous s) ∈S

Considering input predicates IP and output predicates OP inform of predicate categories PCvei, PCvmv and PCvei for each variable 'v_i' in the specification, formulates those list sequences that gives more equivalence classes, that is, a more refined partition of the class's state space. Since the derived states contain non-overlapping disjoint states, hence providing maximal partition (there is no overlap of states and together the states completely cover the class's state space). Also, states generation from IP and OP ensures that resultant states will contain both pre- and post-states of the class FSM respectively.

3.3 Transition Calculation Phase

In state identification phase, valid, disjoint and non-overlapping set of states 'S' was generated for a class FSM from extracted input and output predicates. To determine FSM of a class, transitions are also needed along with the states. Hence, in this phase transitions are calculated for all the operations in a given class.

For calculating transitions, an explicit representation of initial schema, operations of class with respective pre- and postconditions, inform of a text file is provided. Since, TEIOPZ is developed for extraction of input and output predicates for single operation schema. TEIOPZ, when is used for extracting predicates of whole class with more than one operation does not provide any relationship among each of the operation and its respective pre- and postcondition and with that of the predicates that define these conditions. However, extension of TEIOPZ for extraction of predicates from object-oriented class specifications may resolve this issue.

A valid transition 't' for an operation 'opr' in FSM occurs if there exist a pre-state 'Spre' in 'S', satisfying its precondition and a post-state 'Spost' in 'S' satisfying its postcondition. As discussed earlier in section 3.1 a pre- and postcondition is defined by IP and OP respectively. Hence, transitions can be calculated for each 'opr' by identifying states that satisfy each IP used to define the precondition. This will result in identification of all pre-states. Similarly, comparing states with each OP defining postcondition provides all post-states for that operation.

Transitions for all the operations can be calculated in a similar manner. Such transitions are labeled with their respective pre- and post-states and are represented in FSM by an arc labeled by the operation name pointing from pre-state to post-state.

Transitions are determined by applying the boundary value analysis on the given set of states to find the valid pre- and post-states that satisfy pre- and postconditions for all the operation in the given Object-Z class. Hence, to find pre- and post-states for each operation it is necessary to identify those states that satisfy each predicate in its pre- and postcondition.

3.3.1 Identification of Pre-states for an Operation

The class information provided by the explicitly provided file along with the set of states 'S' identified from IP and OP (in section 3.2) will help in calculating transitions for each operation. For the identification of pre-states for an operation, each input predicate containing 'v' in the precondition of that operation is compared with every state 's' derived for variable 'v' in 'S'. The state whose space becomes the subset of the region defined by that predicate is denoted as pre-state for operation 'opr'.

Formally,
For a given operation 'opr',

$$IP \times S \rightarrow Spre$$
$$If\ s \subseteq IP,\ and\ s \in S\ then$$
$$s \in Spre\ (opr)$$

3.3.2 Identification of Post-states for an Operation

Similarly post-state can be identified for an operation. Each output predicate containing variable 'v' in the postcondition is matched with every state's' in S. The state whose state space makes the subset of the region defined by that output predicate is denoted as pre- state for operation 'opr'.

Formally,
For a given operation 'opr',

$$OP \times S \rightarrow Spost$$
$$If\ s \subseteq OP,\ and\ s \in S\ then$$
$$s \in Spost(opr)$$

Pre- and post-states for all operations of the Object-Z class are determined in the similar manner.

A transition 't' can be represented as:

$$Spre \rightarrow opr \rightarrow Spost$$

Where Spre and Spost are the pre- and post-states for operation 'opr'.

The set of transitions 'T_i' is calculated for all operations of the given class. Calculated T_i, along with respective pre- and post-states will help in making FSM of a given Object-Z Class.

```
Algorithm TransitionCalc
Input:        S, OPR, IP, OP
Output:       TS: Transition Set
Declare:      GetPrecondition(): Gets the precondition of a particular operation OPR from File,
              GetPostCondition(): Gets the postcondition of a particular operation OPR from File,
              GetInputpredicates(): Gets the set of input predicates included in the precondition of a particular
              operation, GetoutputPredicates(): Gets the set of output predicates included in the postcondition of a
              particular operation
Begin: TransitionCalc
1.        Add ClassOprationsPreandPost.txt
2.        For (every Opr in class) do
3.            Pre=OPR.GetPrecondition()
4.            Post= OPR.GetPostCondition()
5.            IP=Pre.GetInputpredicates()
6.            OP= Post.GetoutputPredicates()
7.        For (every IP of OPR) do
8.            Check each IP with all s in S
9.                If S⊆IP then
10.               Add s to SPre
11.               Spre=S
12.               End If
13.       End For
14.       For (every opr of OPR) do
15.           Check each OP with all s in S
16.           If s⊆ OP then
17.               Add s to Spost
18.               Spost = S
19.           End If
20.       End For
21.     End For
22.   Return Spre, Spost, OPR
End TransitionCalc
```

Figure 3.6 Transition Calculation Algorithm

3.4 FSM Construction Phase

In Object-Z an operation schema represents a transition from one subset of state space to another, with associated input and output variables. After all transitions 'T' are obtained the number of variables involved in the pre- and post-states that satisfying the pre- and postconditions of an operation are checked. Figure 3.7 shows the algorithm for FSM construction.

If the pre- and post-states of transitions of an operation are defined by single variable then the pre- and post-states are mapped along with the transitions. Such calculated transitions along with the respective pre- and post-states thus make final Class FSM.

3.4.1 State Combinations

If multiple variables are involved in pre-states of transitions of an operation then the identified pre-states need to be combined. The number of these state combinations depends upon the number of pre-states identified for each IP involved in defining precondition.

Similarly, if multiple variables are involved in post-states of transitions of an operation then the identified post-states are combined. The number of these state combinations depends upon the number of post-states identified for each output predicate involved in defining the post condition.

Let opr_i be the set of operations defined in a class, and IP_i be the number of input predicates that define opr's precondition. OPi be the number of output predicates defining opr's postcondition. $Spre_i$ be the set of pre-states identified for each IP in IP_i used in opr_i's precondition and $Spost_i$ be the set of post-states identified for each OP in OP_i used in opr_i's postcondition. Formally, State combinations denoted as 'SC' are,

$$\{opr_i \text{ in } Class \mid i>0\} \text{ and}$$
$$\{IP_j \in opr_i(pre) \mid j>1\}$$
$$\{OP_j \in opr_i(post) \mid j>1\}$$
$$\{Spre_i \subseteq IP_j \text{ and } Spre_i \in S \mid i>0\} \text{ and}$$
$$\{Spost_i \subseteq OP_j \text{ and } Spost_i \in S \mid i>0\}$$

Formally, State combinations for pre-states of an operation are,

$$SCpre_i = \{Spre_i \in Opr_i(pre) \mid \prod_{j=1}^{n} Spre(IP_j)\}$$

State combinations for post-states of an operation,

$$SCpost_i = \{Spost_i \in Opr_i(post) \mid \prod_{j=1}^{n} Spost(OP_j)\}$$

This process will be repeated unless state combinations for all the pre- and post-states of all the operations of the Object-Z class are obtained. The initial state of the FSM is read from the input file provided explicitly and is obtained from initial schema of the Object-Z class. This file contains explicit representation of the relationships of all the operations with their respective pre- and postconditions. To identify the transitions for FSM this file plays a vital role for keeping track of the separated predicates with their respective operations in the original class specifications. Once transitions are calculated, these are mapped along with their respective pre- and post-state combinations obtained for all the operations in an Object-Z class. To reduce the complexity of the resulting FSM, state minimization process is applied. Due to this minimization the number of states and transitions is reduced resulting in a simplified FSM.

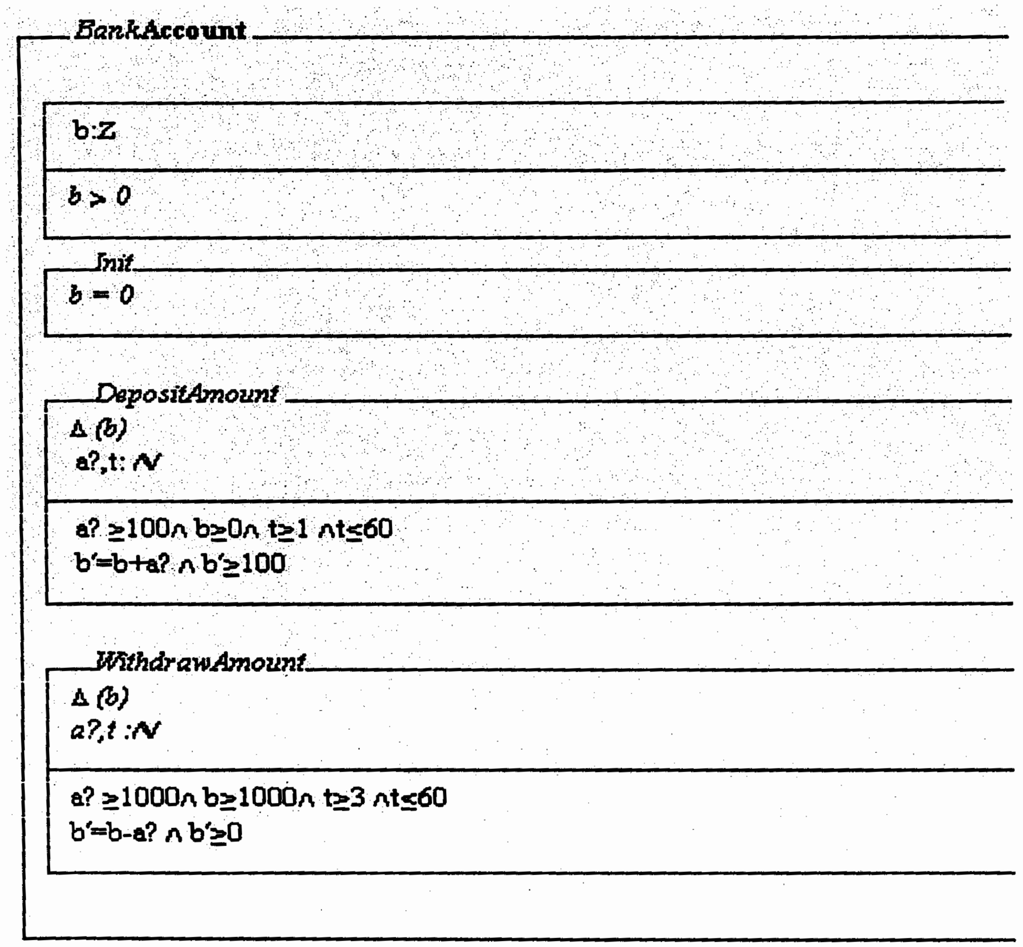
```
Algorithm: FSMConstruction
Input: Spre, Spost, OPR
Output: TransitionSet TS
Declare: GetSpre(): Gets all the prestates of a particular operation , GetSpost(): Gets all the poststates of a particular
         operation, Sinv : Initial state invariant, SC: State Combinations
Begin: FSMConstruction
1.      include ClassOprationsPreandPost.txt
2.      Make Sinv of initial schema Si
3.      For (each opr in OPR) do
4.        If Spre(i)∈ V_k and i>1 then
5.          SC(i)=#Spre(V_1)* #Spre(V_2)* .......*#Spre(V_n)
6.            If Spost(i) ∈ V(i) then
7.              SC(i)=#Spost(V_1)* #Spost(V_2)* .......*#Spost(V_n)
8.              For (every Spre in SC(i)) do
9.                SC(i) ∈ SCpre(opr)
10.               For (every Spost in SC(i)) do
11.                 SC(i) ∈ SCpost(opr)
12.                 TS ts="";
13.                 TS= ts + OPR.GetSCpre() + "--------------->" + OPR +"----------->" + OPR.GetSCpost;()
14.                 TS.Add (ts)
15.               End For
16.             End for
17.           End if
18.       End if
19.     End for
20.     Else
21.       For (every S in Spre) do
22.         For (every S in Spost) do
23.           TS ts="";
24.           TS= ts + OPR.GetSpre() + "---------------->" + OPR +"----------->" + OPR.GetSpost()
25.           TS.Add (ts)
26.         End for
27.       End For
28. Return TS
End FSMConstruction
```

Figure 3.7 FSM Construction Algorithm

3.5 Introduction to the Example

The proposed approach for the generation of FSM is demonstrated on Object-Z specifications of a 'Bank Account' class.

A simple type of bank account has been taken as an example to demonstrate the proposed approach. This type of bank account is opened for a term period 't' of 60 months with no initial balance 'b'. After one month of term period of opening of account the user is allowed to deposit money, but the amount deposited 'a' should be greater than 100$ with in the specified term period. The deposited amount can be withdrawn after the term period of 3 months of opening of account. User can only with draw amount greater than 1000$ only when the balance in the account is greater than 1000$.



BankAccount

$b : \mathbb{Z}$

$b > 0$

Init

$b = 0$

DepositAmount

$\Delta (b)$

$a?, t : \mathbb{N}$

$a? \geq 100 \wedge b \geq 0 \wedge t \geq 1 \wedge t \leq 60$

$b' = b + a? \wedge b' \geq 100$

WithdrawAmount

$\Delta (b)$

$a?, t : \mathbb{N}$

$a? \geq 1000 \wedge b \geq 1000 \wedge t \geq 3 \wedge t \leq 60$

$b' = b - a? \wedge b' \geq 0$

Figure 3.8 Object-Z class specification for Bank Account

3.5.1 Object-Z Representation

The requirement specification of a Bank account class is represented by a class schema named as 'BankAccount' in Figure 3.8. It represents a bank account with INIT schema that initializes the state variable 'b=0'. A state variable 'b' represents the account balance, a? represents input variable being used and 't' represents the intermediate variable. Two operations represented by operation schema 'Deposit Amount' and 'Withdraw Amount' from an account, shows the constraints on the amount deposited and with drawn from the account respectively.

3.5.2 Extraction of Finite State Machine (FSM)

The step by step demonstration of FSM extraction from Object-Z class with necessary explanation is provided below:

Phase 1: Extraction of Input and Output predicates

For FSM generation initially Input and Output predicates from Bank Account Class are extracted using TEIOPZ, resultantly:

IP= {b= 0, a? geq 100, b geq 0, t geq 1,t leq60, a? geq1000, bgeq1000, t geq3,t leq60}

OP= {b'=b+a?, b'=b-a?, b'≥100, b'≥0}

Phase 2: Automatic Generation of FSM

FSM is generated from Bank Account Class, by applying the steps of the proposed approach as explained below:

Step 1: Predicate Categorization

In this step IP and OP are categorized into lists 'Pv' containing predicates with similar variables, the predicates in each 'Pv' is further categorized into predicate categories having infix relational operators containing constants, multiple variables and those containing '=' and '≠', denoted as PCvc, PCvmv and PCvei respectively.

Applying Predicate categorization process results:

Initially,

$P = IP \cup OP$

P = {b= 0, a?geq 100, b geq 0, b′geq100, b′geq 0, t geq 1, t leq60, a? geq1000,
b geq1000, t geq3, t leq60, b′=b+a?, b′=b-a?}

After the categorization of predicates in respective 'Pv' lists,

Pv_1= Pb= {b= 0, b geq1000, b geq0, b′=b+a?, b′=b-a?, b′geq100, b′geq 0}

Pv_2= Pt= {t geq 1,t leq60,t geq3,t leq60}

Pv_3= Pa= {a?geq 1000, a? geq100}

Since the Latex source of the Object-Z class has been taken as input, hence for simplicity these notations are replaced by the respective standard relational operators.

Pv Lists:

Pv_1= Pb= {b= 0, b ≥1000, b ≥0, b′=b+a?, b′=b-a?, b′≥100, b′≥0}

Pv_2= Pt= {t ≥ 1, t ≤60, t ≥3, t ≤ 60}

Pv_3= Pa= {a? ≥1000, a? ≥100}

From each Pv list generated predicates defining variable 'v' interms of constant 'c' are stored in to respective PCvc, PCvei, and Pmvar lists.

After categorizing the predicates in Pb in to predicate categories:

PCbc ={ b ≥1000, b ≥0, b′≥100, b′≥0}

PCbei = {b= 100, b′= b+a?, b′=b-a?}

PCbmv ={}

After categorizing the predicates in Pt in to predicate categories:

PCtc ={ t ≥ 1,t ≤60,t ≥3,t ≤ 60}

PCtei ={}

PCtmv ={}

After categorizing the predicates in Pa in to predicate categories:

PCac $=\{ a?\geq 1000, a? \geq 100\}$

PCaei $=\{\}$

PCamv $= \{\}$

Step 2: Sequenced List Formulation

In this step the lists are now formulated according to a predefined sequence to maintain validity among the resultant states, as shown below:

Applying List Formulation Type 1,

If # PCvc $> 1 \wedge$ PCvmv $= 0$, then

After sorting PCbc, PCtc, PCac in ascending order

(Removing symbols and duplicate predicates)

PCbc $= \{b \geq 0, b \geq 100, b \geq 1000\} \Rightarrow L_b$

PCtc $=\{ t \geq 1, t \geq 3, t \leq 60, t \leq 60\} \Rightarrow L_t$

PCac $= \{a \geq 100, a \geq 1000\} \Rightarrow L_a$

Step 3: State Generation

Once required lists are formulated that states are generated from each of the sequenced lists by applying a set of rules, shown below:

Taking Lists Lb, Lt, La

$L_b = \{b \geq 0, b \geq 100, b \geq 1000\}$

$L_t = \{t \geq 1, t \geq 3, t \leq 60\}$

$L_a = \{a \geq 100, a \geq 1000\}$

Applying state generation process on L0,

$L_b = \{b \geq 0, b \geq 100, b \geq 1000\}$

$S_1 = b<0$

$S_2 = b \geq 0 \wedge b<100$

$S_3 = b\geq 100 \wedge b<1000$

$S_4 = b\geq 1000$

$L_t = \{t\geq 1, t\geq 3, t\leq 60\}$

$S_5 = t<1$

$S_6 = t\geq 1 \wedge t<3$

$S_7 = t\geq 3 \wedge t\leq 60$

$S_8 = t>60$

$L_a = \{a\geq 100, a\geq 1000\}$

$S_9 = a<100$

$S_{10} = a\geq 100 \wedge a<1000$

$S_{11} = a\geq 1000$

Checking S with PCbei, PCtei, and PCaei,

Final set of states generated:

Table 3.1 States generated

State Name	State Invariant
S_0	$b=0$
S_1	$b<0$
S_2	$b\geq 0 \wedge b<100 \wedge b\neq 0$
S_3	$b\geq 100 \wedge b<1000$
S_4	$b\geq 1000$
S_5	$t<1$
S_6	$t\geq 1 \wedge t<3$
S_7	$t\geq 3 \wedge t\leq 60$
S_8	$t>60$
S_9	$a<100$
S_{10}	$a\geq 100 \wedge a<1000$
S_{11}	$a\geq 1000$

Step 4: Transition Calculation

For transition calculation the file containing explicit representation of class operations with respective pre- and postcondition is given below:

Initial state: $b=0$

Operation 1: Deposit Amount

Precondition: $a\geq 100$, $t\geq 1$, $t\leq 60$, $b\geq 0$
Postcondition: $b=b+a$, $b\geq 100$

Operation 2: Withdraw Amount

Precondition: $a\geq 1000$, $t\geq 3$, $t\leq 60$, $b\geq 1000$
Postcondition: $b=b-a$, $b\geq 0$

Figure 3.9 Representation of Operations of Bank Account Class

Final Result:
Initial State: $b=0$

Operation 1: Deposit Amount

Prestates: $b\geq 0\wedge b<100\wedge b\neq 0$, $b\geq 100\wedge b<1000$, $b\geq 1000$, $t\geq 1\wedge t<3$, $t\geq 3\wedge t\leq 60$, $a\geq 100\wedge a<1000$, $a\geq 1000$

Poststates: $b\geq 100\wedge b<1000$, $b\geq 1000$

Operation 2: Withdraw Amount

Prestates: $b\geq 1000$, $t\geq 3\wedge t<60$, $a\geq 1000$

Poststates: $b\geq 0\wedge b<100\wedge b\neq 0$, $b\geq 100\wedge b<1000$, $b\geq 1000$

Figure 3.10 Class Operations with Respective Pre- and Post-states

Step 5: State Combinations

The pre- and postconditions of operations of Bank Account class are defined by predicates with multiple variables. Hence, for identifying states that satisfies the whole pre- and postconditions, combinations of states identified in figure 3.10 for precondition of deposit operation are made. This step is necessary to show all the state combinations that satisfy the pre- and postconditions of each operation in Bank Acount class.

SC= 3 *2 * 2= 12

Table 3.2 State Combination

SC_1	$b\geq 0 \wedge b<100 \wedge b\neq 0$	$t\geq 1 \wedge t<3$	$a\geq 100 \wedge a<1000$
SC_2	$b\geq 0 \wedge b<100 \wedge b\neq 0$	$t\geq 1 \wedge t<3$	$a\geq 1000$
SC_3	$b\geq 0 \wedge b<100 \wedge b\neq 0$	$t\geq 3 \wedge t\leq 60$	$a\geq 100 \wedge a<1000$
SC_4	$b\geq 0 \wedge b<100 \wedge b\neq 0$	$t\geq 3 \wedge t\leq 60$	$a\geq 1000$
SC_5	$b\geq 100 \wedge b<1000$	$t\geq 1 \wedge t<3$	$a\geq 100 \wedge a<1000$
SC_6	$b\geq 100 \wedge b<1000$	$t\geq 1 \wedge t<3$	$a\geq 1000$
SC_7	$b\geq 100 \wedge b<1000$	$t\geq 3 \wedge t\leq 60$	$a\geq 100 \wedge a<1000$
SC_8	$b\geq 100 \wedge b<1000$	$t\geq 3 \wedge t\leq 60$	$a\geq 1000$
SC_9	$b\geq 1000$	$t\geq 1 \wedge t<3$	$a\geq 100 \wedge a<1000$
SC_{10}	$b\geq 1000$	$t\geq 1 \wedge t<3$	$a\geq 1000$
SC_{11}	$b\geq 1000$	$t\geq 3 \wedge t\leq 60$	$a\geq 100 \wedge a<1000$
SC_{12}	$b\geq 1000$	$t\geq 3 \wedge t\leq 60$	$a\geq 1000$

Step 6: States Minimization

The possible combinations of states are minimized now to make lesser number of states and transitions to make a simplified FSM, as shown below:

Table 3.3 State Minimization table

State	Deposit (Operation 1)	Withdraw(Operation 2)
SC_1	$SC_5, SC_6, SC_7, SC_8, SC_9, SC_{10}, SC_{11}, SC_{12}$	--------------
SC_2	$SC_5, SC_6, SC_7, SC_8, SC_9, SC_{10}, SC_{11}, SC_{12}$	--------------
SC_3	$SC_5, SC_6, SC_7, SC_8, SC_9, SC_{10}, SC_{11}, SC_{12}$	--------------
SC_4	$SC_5, SC_6, SC_7, SC_8, SC_9, SC_{10}, SC_{11}, SC_{12}$	--------------
SC_5	$SC_5, SC_6, SC_7, SC_8, SC_9, SC_{10}, SC_{11}, SC_{12}$	--------------
SC_6	$SC_5, SC_6, SC_7, SC_8, SC_9, SC_{10}, SC_{11}, SC_{12}$	--------------
SC_7	$SC_5, SC_6, SC_7, SC_8, SC_9, SC_{10}, SC_{11}, SC_{12}$	--------------
SC_8	$SC_5, SC_6, SC_7, SC_8, SC_9, SC_{10}, SC_{11}, SC_{12}$	--------------
SC_9	$SC_5, SC_6, SC_7, SC_8, SC_9, SC_{10}, SC_{11}, SC_{12}$	--------------
SC_{10}	$SC_5, SC_6, SC_7, SC_8, SC_9, SC_{10}, SC_{11}, SC_{12}$	--------------
SC_{11}	$SC_5, SC_6, SC_7, SC_8, SC_9, SC_{10}, SC_{11}, SC_{12}$	--------------
SC_{12}	[illegible]	[illegible]

In this step the source states that have similar target states combinations are disjunction into one new state and renaming the rest of the states give:

$$SC_1 \vee SC_2 \vee SC_3 \vee SC_4 \vee SC_5 \vee SC_6 \vee SC_7 \vee SC_8 \vee SC_9 \vee SC_{10} \vee SC_{11} \Rightarrow S_{13}$$
$$SC_{12} \Rightarrow S_{14}$$
$$SC_0 \Rightarrow S_0$$

The Resulting FSM is:

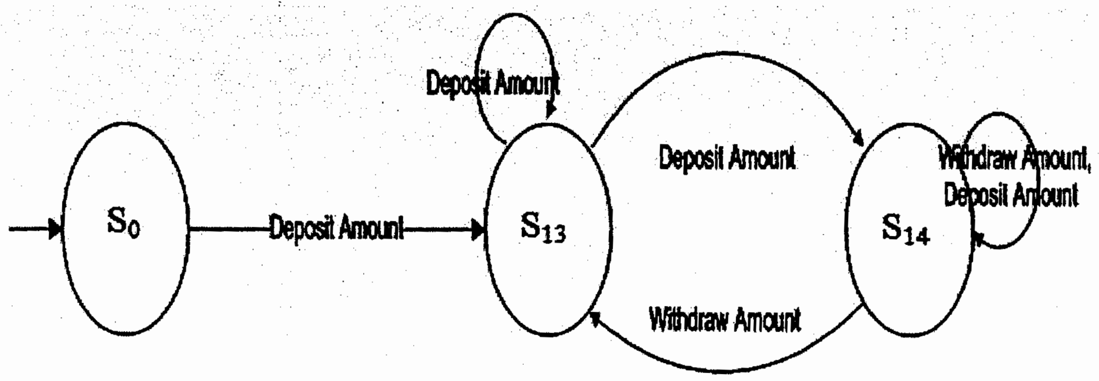


Figure 3.11 FSM of Bank Account class

3.6 Hypothetical Example

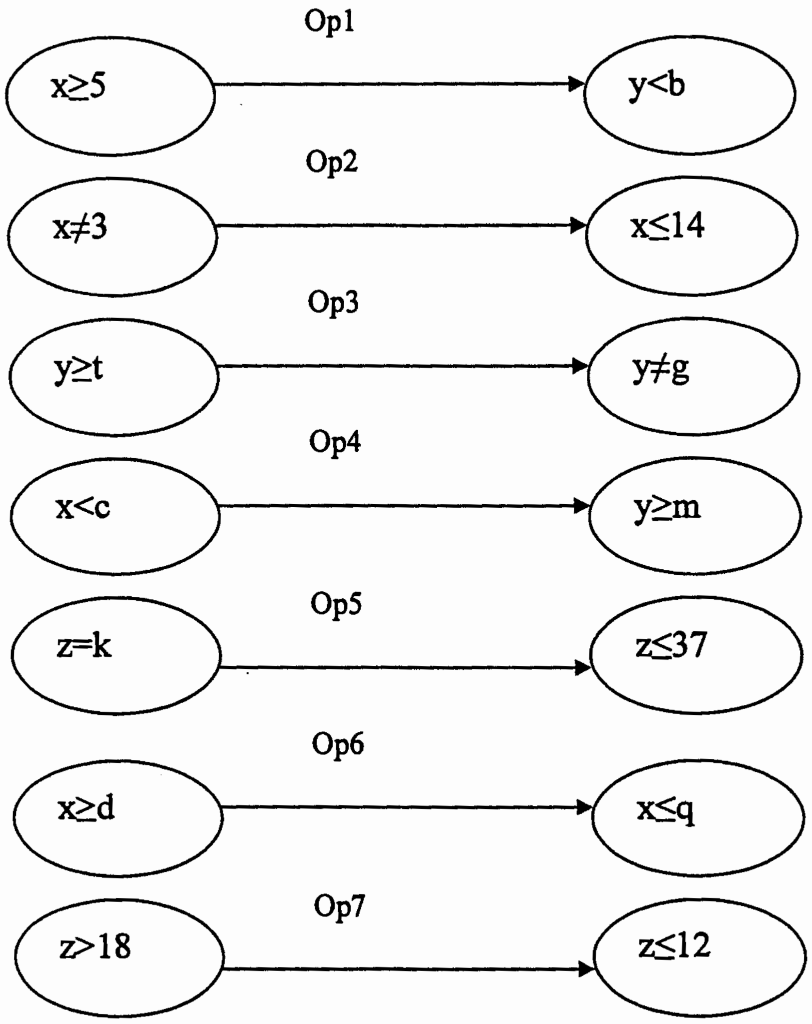


Figure 3.12 Hypothetical Example

Extracted set of input predicates 'IP' and output predicates 'OP' using TEIOPZ:

IP: $\{x \geq 5, x \neq 3, y \geq t, x < c, z = k, x \geq d, z > 18\}$
OP: $\{y < 25, x \leq 14, y \neq g, y \geq m, z \leq 37, x \leq q, z \leq 12\}$

Step 1: Categorization of Predicates into 'Pv' lists,

Px = $\{x \geq 5, x \neq 3, x \leq 14, x < c, x \geq d, x \leq q\}$
Py = $\{y < b, y \geq t, y \neq g, y \geq m\}$
Pz = $\{z = k, z \leq 37, z > 18, z \leq 12\}$

Step 2: Categorization of Predicates into lists PCvc, PCvmv, and PCvei

PCxc = $\{x \leq 14, x \geq 5\}$
PCxmv = $\{x<c, x \geq d, x \leq q\}$
PCxei = $\{x \neq 3\}$

PCyc = {}
PCymv = $\{y<b, y \geq t, y \geq m\}$
PCyei= $\{y \neq g\}$

PCzc = $\{z \leq 37, z>18, z \leq 12\}$
PCzmv= {}
PCzei= $\{z=k\}$

Step 3: Sorting the extracted PCvc lists if any,

Sort PCxc, PCyc and PCzc

PCxc= $\{x \geq 5, x \leq 14\}$
PCyc = { }
PCzc= $\{z \leq 12, z>18, z \leq 37\}$

Step 4: Combine PCxmv and PCxc if any resulatant PCxmvc will be,

PCxmvc= $\{x \geq 5, x \leq 14, x<c, x \geq d, x \leq q\}$
Combine PCyc and PCymv,
PCymvc = $\{y<b, y \geq t, y \geq m\}$
Combine PCzc and PCzmv,
PCzmvc= $\{z \leq 37, z>18, z \leq 12\}$

Step 5: Make sequenced list formulations, if both PCvc and PCvmv are non-empty,

No. of Sequenced lists for PCxmvc will be n-1*n-2*n-3...
For PCxmvc
4*3*2*1=24 Lists

Lmxc$_1$= $\{x \geq 5, x \leq 14, x \leq q, x<c, x \geq d\}$
Lmxc$_2$= $\{x \geq 5, x \leq q, x \leq 14, x<c, x \geq d\}$
Lmxc$_3$= $\{x \leq q, x \geq 5, x \leq 14, x<c, x \geq d\}$
Lmxc$_4$= $\{x \geq 5, x \leq q, x<c, x \geq d, x \leq 14\}$

Lmxc$_5$= $\{x \geq 5, x \leq 14, x \leq q, x \geq d, x<c\}$
Lmxc$_6$= $\{x \geq 5, x \leq q, x \leq 14, x \geq d, x<c\}$
Lmxc$_7$= $\{x \leq q, x \geq 5, x \leq 14, x \geq d, x<c\}$
Lmxc$_8$= $\{x \geq 5, x \leq q, x \geq d, x<c, x \leq 14\}$
Lmxc$_9$= $\{x \geq 5, x \leq 14, x \geq d, x<c, x \leq q\}$

$Lmxc_{10}= \{x\geq 5, x\geq d, x\leq 14, x<c, x\leq q\}$
$Lmxc_{11}= \{x\geq d, x\geq 5, x\leq 14, x<c, x\leq q\}$
$Lmxc_{12}= \{x\geq 5, x\geq d, x<c, x\leq q, x\leq 14\}$

$Lmxc_{13}= \{x\geq 5, x\leq 14, x\geq d, x\leq q, x<c\}$
$Lmxc_{14}= \{x\geq 5, x\geq d, x\leq 14, x\leq q, x<c\}$
$Lmxc_{15}= \{x\geq d, x\geq 5, x\leq 14, x\leq q, x<c\}$
$Lmxc_{16}= \{x\geq 5, x\geq d, x\leq q, x<c, x\leq 14\}$

$Lmxc_{17}= \{x\geq 5, x\leq 14, x<c, x\geq d, x\leq q\}$
$Lmxc_{18}= \{x\geq 5, x<c, x\leq 14, x\geq d, x\leq q\}$
$Lmxc_{19}= \{x<c, x\geq 5, x\leq 14, x\geq d, x\leq q\}$
$Lmxc_{20}= \{x\geq 5, x<c, x\geq d, x\leq q, x\leq 14\}$

$Lmxc_{21}= \{x\geq 5, x\leq 14, x<c, x\leq q, x\geq d\}$
$Lmxc_{22}= \{x\geq 5, x<c, x\leq 14, x\leq q, x\geq d\}$
$Lmxc_{23}= \{x<c, x\geq 5, x\leq 14, x\leq q, x\geq d\}$
$Lmxc_{24}= \{x\geq 5, x<c, x\leq q, x\geq d, x\leq 14\}$

For PCymvc
No. of Sequenced lists for PCymvc will be like n*n-1*n-2*n-3……., if only PCymv is non-empty,

3*2*1=6 sets

$Lmyc_1 = \{y<b, y\geq t, y\geq m\}$
$Lmyc_2 = \{y\geq t, y<b, y\geq m\}$
$Lmyc_3 =\{y<b, y\geq m, y\geq t\}$
$Lmyc_4= \{y\geq m, y<b, y\geq t\}$
$Lmyc_5 = \{y\geq t, y\geq m, y<b\}$
$Lmyc_6 = \{y\geq m, y\geq t, y<b\}$

There is no PCzmv so no need to make its sequenced list formulations.

Step 6: Making states for all the sequenced lists formulated by applying the state generation process,

For $Lmxc_1=\{x\geq 5, x\leq 14, x\leq q, x<c, x\geq d\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \land (x\leq 14) \rightarrow (5\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \land (x\leq q) \rightarrow (14<x\leq q) \rightarrow S$
$\sim(x\leq q) \land (x<c) \rightarrow (q<x<c) \rightarrow S$
$\sim(x<c) \land \sim(x\geq d) \rightarrow (c\leq x<d) \rightarrow S$
$(x\geq d) \rightarrow S$

For $Lmxc_2=\{x\geq 5, x\leq q ,x\leq 14, x<c, x\geq d\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \land (x\leq q) \rightarrow (5\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \land (x\leq 14) \rightarrow (q<x\leq 14) \rightarrow S$
$\sim(x\leq 14) \land (x<c) \rightarrow (14\leq x<c) \rightarrow S$
$\sim(x<c) \land \sim(x\geq d) \rightarrow (c\leq x<d) \rightarrow S$
$(x\geq d) \rightarrow S$

For $Lmxc_3 = \{x\leq q, x\geq 5, x\leq 14, x<c, x\geq d\}$

$(x\leq q) \rightarrow S2$
$\sim(x\leq q) \wedge \sim(x\geq 5) \rightarrow (q<x<5) \rightarrow S$
$(x\geq 5) \wedge (x\leq 14) \rightarrow (5\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge (x<c) \rightarrow (14<x<c) \rightarrow S$
$\sim(x<c) \wedge \sim(x\geq d) \rightarrow (c\leq x<d) \rightarrow S$
$(x\geq d) \rightarrow S$

For $Lmxc_4 = \{x\geq 5, x\leq q, x<c, x\geq d, x\leq 14\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge (x\leq q) \rightarrow (5\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \wedge (x<c) \rightarrow (q<x<d) \rightarrow S$
$\sim(x<c) \wedge \sim(x\geq d) \rightarrow (c\leq x<d) \rightarrow S$
$(x\geq d) \wedge (x\leq 14) \rightarrow (d\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \rightarrow (x>14) \rightarrow S$

$Lmxc_5 = \{x\geq 5, x\leq 14, x\leq q, x\geq d, x<c\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge (x\leq 14) \rightarrow (5\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge (x\leq q) \rightarrow (14<x\leq q) \rightarrow S$
$\sim(x\leq q) \wedge \sim(x\geq d) \rightarrow (q<x<d) \rightarrow S$
$(x\geq d) \wedge (x<c) \rightarrow (d\leq x<c) \rightarrow S$
$\sim(x<c) \rightarrow (x\geq c) \rightarrow S$

$Lmxc_6 = \{x\geq 5, x\leq q, x\leq 14, x\geq d, x<c\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge (x\leq q) \rightarrow (5\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \wedge (x\leq 14) \rightarrow (q<x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge \sim(x\geq d) \rightarrow (14<x<d) \rightarrow S$
$(x\geq d) \wedge (x<c) \rightarrow (d\leq x<c) \rightarrow S$
$\sim(x<c) \rightarrow (x\geq c) \rightarrow S$

$Lmxc_7 = \{x\leq q, x\geq 5, x\leq 14, x\geq d, x<c\}$

$(x\leq q) \rightarrow S$
$\sim(x\leq q) \wedge \sim(x\geq 5) \rightarrow (q<x<5) \rightarrow S$
$(x\geq 5) \wedge (x\leq 14) \rightarrow (5\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge \sim(x\geq d) \rightarrow (14<x<d) \rightarrow S$
$(x\geq d) \wedge (x<c) \rightarrow (d\leq x<c) \rightarrow S$
$\sim(x<c) \rightarrow (x\geq c) \rightarrow S$

$Lmxc_8 = \{x\geq 5, x\leq q, x\geq d, x<c, x\leq 14\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge (x\leq q) \rightarrow (5\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \wedge \sim(x\geq d) \rightarrow (q<x<d) \rightarrow S$
$(x\geq d) \wedge (x<c) \rightarrow (d\leq x<c) \rightarrow S$
$\sim(x<c) \wedge (x\leq 14) \rightarrow (c\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \rightarrow (x>14) \rightarrow S$

$Lmxc_9 = \{x\geq 5, x\leq 14, x\geq d, x<c, x\leq q\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge (x\leq 14) \rightarrow (5\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge \sim(x\geq d) \rightarrow (14<x<d) \rightarrow S$
$(x\geq d) \wedge (x<c) \rightarrow (d\leq x<c) \rightarrow S$
$\sim(x<c) \wedge (x\leq q) \rightarrow (c\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \rightarrow (x>q) \rightarrow S$

$Lmxc_{10} = \{x\geq 5, x\geq d, x\leq 14, x<c, x\leq q\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge \sim(x\geq d) \rightarrow (5\leq x<d) \rightarrow S$
$(x\geq d) \wedge (x\leq 14) \rightarrow (d\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge (x<c) \rightarrow (14<x<c) \rightarrow S$
$\sim(x<c) \wedge (x\leq q) \rightarrow (c\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \rightarrow (x>q) \rightarrow S$

$Lmxc_{11} = \{x\geq d, x\geq 5, x\leq 14, x<c, x\leq q\}$

$\sim(x\geq d) \rightarrow (x<d) \rightarrow S$
$(x\geq d) \wedge \sim(x\geq 5) \rightarrow (d\leq x<5) \rightarrow S$
$(x\geq 5) \wedge (x\leq 14) \rightarrow (5\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge (x<c) \rightarrow (14<x<c) \rightarrow S$
$\sim(x<c) \wedge (x\leq q) \rightarrow (c\leq x\leq q) \rightarrow S2$
$\sim(x\leq q) \rightarrow (x>q) \rightarrow S$

$Lmxc_{12} = \{x\geq 5, x\geq d, x<c, x\leq q, x\leq 14\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge \sim(x\geq d) \rightarrow (5\leq x<d) \rightarrow S$
$(x\geq d) \wedge (x<c) \rightarrow (d\leq x<c) \rightarrow S$
$\sim(x<c) \wedge (x\leq q) \rightarrow (c\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \wedge (x\leq 14) \rightarrow (q<x\leq 14) \rightarrow S$
$\sim(x\leq 14) \rightarrow (x>14) \rightarrow S$

$Lmxc_{13}=\{x\geq 5, x\leq 14, x\geq d, x\leq q, x<c\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge (x\leq 14) \rightarrow (5\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge \sim(x\geq d) \rightarrow (14<x<d) \rightarrow S$
$(x\geq d) \wedge (x\leq q) \rightarrow (d\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \wedge (x<c) \rightarrow (q<x<c) \rightarrow S$
$\sim(x<c) \rightarrow (x\geq c) \rightarrow S$

$Lmxc_{14}=\{x\geq 5, x\geq d, x\leq 14, x\leq q, x<c\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge \sim(x\geq d) \rightarrow (5\leq x<d) \rightarrow S$
$(x\geq d) \wedge (x\leq 14) \rightarrow (d\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge (x\leq q) \rightarrow (14<x\leq q) \rightarrow S$
$\sim(x\leq q \wedge (x<c) \rightarrow (q<x<c) \rightarrow S$
$\sim(x<c) \rightarrow (x\geq c) \rightarrow S$

$Lmxc_{15}=\{x\geq d, x\geq 5, x\leq 14, x\leq q, x<c\}$

$\sim(x\geq d) \rightarrow (x<d) \rightarrow S$
$(x\geq d) \wedge \sim(x\geq 5) \rightarrow (d\leq x<5) \rightarrow S$
$(x\geq 5) \wedge (x\leq 14) \rightarrow (5\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge (x\leq q) \rightarrow (14<x\leq q \rightarrow S$
$\sim(x\leq q) \wedge (x<c) \rightarrow (q<x<c) \rightarrow S$
$\sim(x<c) \rightarrow (x\geq c) \rightarrow S$

$Lmxc_{16}=\{x\geq 5, x\geq d, x\leq q, x<c, x\leq 14\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge \sim(x\geq d) \rightarrow (5\leq x<d) \rightarrow S$
$(x\geq d) \wedge (x\leq q) \rightarrow (d\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \wedge (x<c) \rightarrow (q<x<c) \rightarrow S$
$\sim(x<c) \wedge (x\leq 14) \rightarrow (c\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \rightarrow (x>14) \rightarrow S$

$Lmxc_{17}=\{x\geq 5, x\leq 14, x<c, x\geq d, x\leq q\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge (x\leq 14) \rightarrow (5\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge (x<c) \rightarrow (14<x<c) \rightarrow S$
$\sim(x<c) \wedge \sim(x\geq d \rightarrow (c\leq x<d) \rightarrow S$
$(x\geq d) \wedge (x\leq q) \rightarrow (d\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \rightarrow (x>q) \rightarrow S$

$Lmxc_{18}=\{x\geq 5, x<c, x\leq 14, x\geq d, x\leq q\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge (x<c) \rightarrow (5\leq x<c) \rightarrow S$
$\sim(x<c) \wedge (x\leq 14) \rightarrow (c\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge \sim(x\geq d) \rightarrow (14<x<d) \rightarrow S$
$(x\geq d) \wedge (x\leq q) \rightarrow (d\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \rightarrow (x>q) \rightarrow S$

$Lmxc_{19}=\{x<c, x\geq 5, x\leq 14, x\geq d, x\leq q\}$

$(x<c) \rightarrow S$
$\sim(x<c) \wedge \sim(x\geq 5) \rightarrow (c\leq x<5) \rightarrow S$
$(x\geq 5) \wedge (x\leq 14) \rightarrow (5\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge \sim(x\geq d) \rightarrow (14<x<d) \rightarrow S$
$(x\geq d) \wedge (x\leq q) \rightarrow (d\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \rightarrow (x>q) \rightarrow S$

$Lmxc_{20}=\{x\geq 5, x<c, x\geq d, x\leq q, x\leq 14\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge (x<c) \rightarrow (5\leq x<c) \rightarrow S$
$\sim(x<c) \wedge \sim(x\geq d) \rightarrow (c\leq x<d) \rightarrow S$
$(x\geq d) \wedge (x\leq q) \rightarrow (d\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \wedge (x\leq 14) \rightarrow (q<x\leq 14) \rightarrow S$
$\sim(x\leq 14) \rightarrow (x>14) \rightarrow S$

$Lmxc_{21}=\{x\geq 5, x\leq 14, x<c, x\leq q, x\geq d\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge (x\leq 14) \rightarrow (5\leq x\leq 14) \rightarrow S$
$\sim(x\leq 14) \wedge (x<c) \rightarrow (14<x\leq c) \rightarrow S$
$\sim(x<c) \wedge (x\leq q) \rightarrow (c<x\leq q) \rightarrow S$
$\sim(x\leq q) \wedge \sim(x\geq d) \rightarrow (q<x<d) \rightarrow S$
$(x\geq d) \rightarrow S$

$Lmxc_{22}=\{x\geq 5, x<c, x\leq 14, x\leq q, x\geq d\}$

$\sim(x\geq 5) \rightarrow (x<5) \rightarrow S$
$(x\geq 5) \wedge (x<c) \rightarrow (5\leq x<c) \rightarrow S$
$\sim(x<c) \wedge (x\leq 14) \rightarrow (c\leq x\leq 14) \rightarrow S$
$\sim x\leq 14) \wedge (x\leq q) \rightarrow (14<x\leq q) \rightarrow S$
$\sim(x\leq q) \wedge \sim(x\geq d) \rightarrow (q<x<d) \rightarrow S$
$(x\geq d) \rightarrow S$

$Lmxc_{23}$= {$x<c, x\geq5, x\leq14, x\leq q, x\geq d$}

$(x<c) \rightarrow S$
$\sim(x<c) \wedge \sim(x\geq5) \rightarrow (c\leq x<5) \rightarrow S$
$(x\geq5) \wedge (x\leq14) \rightarrow (5\leq x\leq14) \rightarrow S$
$\sim(x\leq14) \wedge (x\leq q) \rightarrow (14<x<q) \rightarrow S$
$\sim(x\leq q) \wedge \sim(x\geq d) \rightarrow (q<x<d) \rightarrow S$
$(x\geq d) \rightarrow S$

$Lmxc_{24}$= {$x\geq5, x<c, x\leq q, x\geq d, x\leq14$}

$\sim(x\geq5) \rightarrow (x<5) \rightarrow S$
$(x\geq5) \wedge (x<c) \rightarrow (5\leq x<c) \rightarrow S$
$\sim(x<c) \wedge (x\leq q) \rightarrow (c\leq x\leq q) \rightarrow S$
$\sim(x\leq q) \wedge \sim(x\geq d) \rightarrow (q<x<d) \rightarrow S$
$(x\geq d) \wedge (x\leq14) \rightarrow (d\leq x\leq14) \rightarrow S$
$\sim(x\leq14) \rightarrow (x>14) \rightarrow S$

3.6.1 Extracted Set of Non-overlapping States

Table 3.4 Removing the Duplicate State Invariants

$Lmxc_1$	$Lmxc_2$	$Lmxc_3$	$Lmxc_4$	$Lmxc_5$	$Lmxc_6$	$Lmxc_7$	$Lmxc_8$
$x<5$		$x\leq q$					
$5\leq x\leq14$	$5\leq x\leq q$	$q<x<5$					
$14<x\leq q$	$q<x\leq14$		$q<x<d$				
$q<x<c$	$14\leq x<c$				$14<x<d$		
$c\leq x<d$			$d\leq x\leq14$	$d\leq x<c$			
$x\geq d$			■	$x\geq c$			
$Lmxc_9$	**$Lmxc_{10}$**	**$Lmxc_{11}$**	**$Lmxc_{12}$**	**$Lmxc_{13}$**	**$Lmxc_{14}$**	**$Lmxc_{15}$**	**$Lmxc_{16}$**
		■					
	$5\leq x<d$	$d\leq x<5$					
				$d\leq x\leq q$			
	$14<x<c$						$c\leq x\leq14$
$c\leq x\leq q$							
■							
$Lmxc_{17}$	**$Lmxc_{18}$**	**$Lmxc_{19}$**	**$Lmxc_{20}$**	**$Lmxc_{21}$**	**$Lmxc_{22}$**	**$Lmxc_{23}$**	**$Lmxc_{24}$**
		$x<c$					
$5\leq x<c$		$c\leq x<5$					

For PCymvc:
$PCymvc = \{y<b, y\geq t, y\geq m\}$

We have,

$Lmyc_1 = \{y<b, y\geq t, y\geq m\}$
$(y<b) \rightarrow S$
$(y<b) \wedge \sim(y\geq t) \rightarrow (b\leq y<t) \rightarrow S$
$(y\geq t) \wedge \sim(y\geq m) \rightarrow (t\leq y<m) \rightarrow S$
$(y\geq m) \rightarrow S$

$Lmyc_2 = \{y\geq t, y<b, y\geq m\}$
$\sim (y\geq t) \rightarrow (y<t) \rightarrow S$
$(y\geq t) \wedge (y<b) \rightarrow (t\leq y\leq b) \rightarrow S$
$\sim(y<b) \wedge \sim(y\geq m) \rightarrow (b\leq y<m) \rightarrow S$
$(y\geq m) \rightarrow S$

$Lmyc_3 = \{y<b, y\geq m, y\geq t\}$
$(y<b) \rightarrow S$
$\sim(y<b) \wedge \sim(y\geq m) \rightarrow (b\leq y<m) \rightarrow S$
$(y\geq m) \wedge \sim(y\geq t) \rightarrow (m\leq y<t) \rightarrow S$
$(y\geq t) \rightarrow S$

$Lmyc_4 = \{y\geq m, y<b, y\geq t\}$
$\sim(y\geq m) \rightarrow (y<m) \rightarrow S$
$(y\geq m) \wedge (y<b) \rightarrow (m\leq y<b) \rightarrow S$
$\sim(y<b) \wedge \sim(y\geq t) \rightarrow (b\leq y<t) \rightarrow S$
$(y\geq t) \rightarrow S$

$Lmyc_5 = \{y\geq t, y\geq m, y<b\}$
$\sim(y\geq t) \rightarrow y<t \rightarrow S$
$(y\geq t) \wedge \sim(y\geq m) \rightarrow (t\leq y<m) \rightarrow S$
$\sim(y<m) \wedge \sim(y\geq b) \rightarrow (m\leq y<b) \rightarrow S$
$\sim(y<b) \rightarrow (y\geq b) \rightarrow S$

$Lmyc_6 = \{y\geq m, y\geq t, y<b\}$
$\sim (y\geq m) \rightarrow y<m \rightarrow S$
$(y\geq m) \wedge \sim (y\geq t) \rightarrow m\leq y<t \rightarrow S$
$(y\geq m) \wedge (y<b) \rightarrow m\leq y<b \rightarrow S$
$\sim(y<b) \rightarrow (y\geq b) \rightarrow S$

Table 3.5 Set of States Generated From $Lmyc_i$

$Lmyc_1$	$Lmyc_2$	$Lmyc_3$	$Lmyc_4$	$Lmyc_5$	$Lmyc_6$
$y<b$	$y<t$		$y<m$		
$b\leq y<t$	$t\leq y\leq b$		$m\leq y<b$		
$t\leq y<m$	$b\leq y<m$	$m\leq y<t$			
$y\geq m$		$y\geq t$		$y\geq b$	

For PCzmvc:

$PCzc = \{z\leq 12, z>18, z\leq 37\}$
$(z\leq 12) \rightarrow S$
$\sim(z\leq 12) \wedge \sim(z>18) \rightarrow 12<z\leq 18 \rightarrow S$
$\sim(z\leq 18) \wedge \sim(z>37) \rightarrow 18<z\leq 37 \rightarrow S$
$\sim(z\leq 37) \rightarrow z>37 \rightarrow S$

Table 3.6 Set of States Generated From PCzmvc

$Lmzc_1$
$z \leq 12$
$12 < z \leq 18$
$18 < z \leq 37$
$z > 37$

Step 7: Making final set of states by adjusting PCxei if any

PCxei={x=3}
Make x=3 ➔S (and add x≠3 where it lies among predicates)

Applying boundary analysis on the states generated from $Lmxc_i$

For PCyei= {y=g} adjusting it with corresponding states in $Lmyc_i$

Making (y=g) ➔ S
For PCzei= {z≠k} adjusting it with corresponding states in $Lmzc_i$

Step 8: Making final set of states

Table 3.7 Final Set of States for 'Px'

States	$Lmxc_1$	State	$Lmxc_2$	States	$Lmxc_4$	States	$Lmxc_6$	States	$Lmxc_8$
S_1	$x<5 \wedge x \neq 3$	S_7	$5 \leq x \leq q$	S_{13}	$d \leq x \leq 14$	S_{19}	[illegible]	S_{25}	$5 \leq x < c$
S_2	$5 \leq x \leq 14$	S_8	$q < x \leq 14$	S_{14}	$x > 14$	S_{20}	$5 \leq x < d$	S_{26}	[illegible] $\wedge$ $x \neq 3$
S_3	$14 < x \leq q$	S_9	$14 \leq x < c$	S_{15}	$d \leq x < c$	S_{21}	$x<d$ $\wedge$ $x \neq 3$	S_{27}	$c \leq x < 5 \wedge x \neq 3$
S_4	$q < x < c$	S_{10}	$x < q \wedge x \neq 3$	S_{16}	$x > c$	S_{22}	$d \leq x < 5 \wedge x \neq 3$		
S_5	$c \leq x < d$	S_{11}	$q < x < 5 \wedge x \neq 3$	S_{17}	$14 < x < d$	S_{23}	$d \leq x \leq q$		
S_6	$x \geq d$	S_{12}	$q < x < d$	S_{18}	$c \leq x \leq q$	S_{24}	$c \leq x \leq 14$		

Table 3.8 Final Set of States for 'Py'

States	Invariants	States	Invariants
S_{28}	$y<b \wedge y\neq g$	S_{34}	$b\leq y<m \wedge y\neq g$
S_{29}	$b\leq y<t \wedge y\neq g$	S_{35}	$m\leq y<t \wedge y\neq g$
S_{30}	$t\leq y<m \wedge y\neq g$	S_{36}	$y\geq t \wedge y\neq g$
S_{31}	$y\geq m \wedge y\neq g$	S_{37}	$y<m \wedge y\neq g$
S_{32}	$y<t \wedge y\neq g$	S_{38}	$m\leq y<b \wedge y\neq g$
S_{33}	$t\leq y\leq b \wedge y\neq g$	S_{39}	$y\geq b\wedge y\neq g$

Table 3.9 Final Set of States for 'Pz'

States	Lmzci
S_{40}	$z\leq 12 \wedge z\neq k$
S_{41}	$12<z\leq 18 \wedge z\neq k$
S_{42}	$18<z\leq 37 \wedge z\neq k$
S_{43}	$z>37\wedge z\neq k$
S_{44}	$z=k$

Step 9: Identifying Pre- and Post-states for Each Pre- and Postcondition

The identified pre- and post-states for pre- and postcondition of an operation are further minimized to get lesser number of states and transitions. The source states having the similar destination states are combined to form new states.

$S_1 \rightarrow S_{45}$
$S_2 \vee S_3 \vee S_4 \vee S_8 \vee S_{15} \rightarrow S_{46}$
$S_5 \vee S_{12} \vee S_{13} \vee S_{17} \vee S_{20} \rightarrow S_{47}$
$S_6 \vee S_{14} \vee S_{16} \vee S_{18} \vee S_{19} \vee S_{23} \vee S_{24} \rightarrow S_{48}$
$S_7 \vee S_9 \rightarrow S_{49}$
$S_{10} \vee S_{11} \vee S_{21} \vee S_{27} \rightarrow S_{50}$
$S_{22} \rightarrow S_{51}$
$S_{25} \vee S26 \rightarrow S52$
$S_{30} \vee S_{33} \vee S_{36} \rightarrow S_{53}$
$S_{42} \vee S_{43} \rightarrow S_{54}$
$S_{44} \rightarrow S_{55}$

$S_{28} \vee S_{29} \vee S_{31} \vee S_{32} \vee S_{34} \vee S_{35} \vee S_{37} \vee S_{38} \vee S_{39} \vee S_{40} \vee S_{41} \rightarrow S_{56}$

3.6.2 State Transition Table

Table 3.10 Final Set of States after State Minimization

States	Op_1	Op_2	OP_3	Op_4	Op_5	Op_6	Op_7
S_{45}				S_{56}			
S_{46}	S_{53}, S_{56}	S_{45}, S_{46}, S_{47}, S_{48}, S_{49}, S_{50}, S_{51}, S_{52}		S_{56}		S_{46}, S_{47}, S_{48}, S_{49}, S_{50}, S_{51}, S_{52}	
S_{47}	S_{53}, S_{56}	S_{45}, S_{46}, S_{47}, S_{48}, S_{49}, S_{50}, S_{51}, S_{52}					
S_{48}	S_{53}, S_{56}	S_{45}, S_{46}, S_{47}, S_{48}, S_{49}, S_{50}, S_{51}, S_{52}				S_{46}, S_{47}, S_{48}, S_{49}, S_{50}, S_{51}, S_{52}	
S_{49}	S_{53}, S_{56}	S_{45}, S_{46}, S_{47}, S_{48}, S_{49}, S_{50}, S_{51}, S_{52}		S_{56}			
S_{50}		S_{45}, S_{46}, S_{47}, S_{48}, S_{49}, S_{50}, S_{51}, S_{52}					
S_{51}		S_{45}, S_{46}, S_{47}, S_{48}, S_{49}, S_{50}, S_{51}, S_{52}				S_{46}, S_{47}, S_{48}, S_{49}, S_{50}, S_{51}, S_{52}	
S_{52}		S_{45}, S_{46}, S_{47}, S_{48}, S_{49}, S_{50}, S_{51}, S_{52}		S_{56}			
S_{53}			S_{53}, S_{56}				
S_{54}							S_{56}
S_{55}					S_{54}, S_{56}		
S_{56}							

4

Automation & Tool Support

4. AUTOMATION AND TOOL SUPPORT

This chapter describes the tool named as "Automatic Generator of Finite State Machine" (AGFSM), which automates the proposed approach for generation of FSM from an Object-Z class. This chapter presents automation details including tool architecture, responsibilities of different tool components and screen shots of GUIs (Graphical User Interface).

4.1 Tool Architecture

AGFSM is a tool for extraction of states and transitions of FSM of an Object Z class. Abstract view of AGFSM is shown in figure 4.1. It takes input and output predicates extracted using an existing tool TEIOPZ (Latif et al., 2008) from the LaTeX source of the given Object-Z class specification. On the basis of these extracted input and output predicates, it identifies the states invariants and displays them in the graphical user interface (GUI). Then, it calculates transitions using identified states and an explicitly given input file showing relationship among class's operations and their respective pre and postconditions and finally displays them in the GUI.

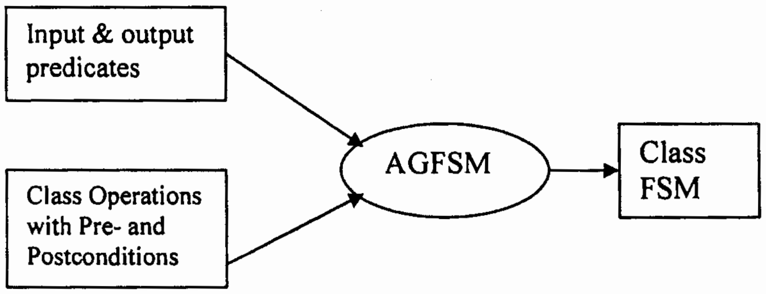

Input & output predicates
Class Operations with Pre- and Postconditions
AGFSM
Class FSM

Figure 4.1 Abstract View of the tool

The tool achieves final output using four main components, i.e., Predicate Categorizer, Sequence List Formulator, State Identifier, and Transition Calculator as shown in figure 4.2.

4.1.1 Predicate Categorizer

The input and output predicates are scanned to identify the predicates with similar variables. This list of predicates with similar variables is further categorized on the basis of constants and variables used in the relational predicates. The predicate categories for each variable used in the given Object-Z class specification are extracted by this component. Categorization of predicates is discussed in detail in section 3.2.1.

4.1.2 Sequenced List Formulator

This component takes the predicate categories as input and transforms them into sequenced lists. It formulates the permuted lists in a valid sequence on the basis of number of predicates present in the each predicate category containing constants and multiple variables such as PCvc and PCvmv respectively. Details of sequenced list formulation from these predicate categories are given in section 3.2.2. These sequenced lists formulated are used as input in the state generation process, hence ensuring validity of resultant states.

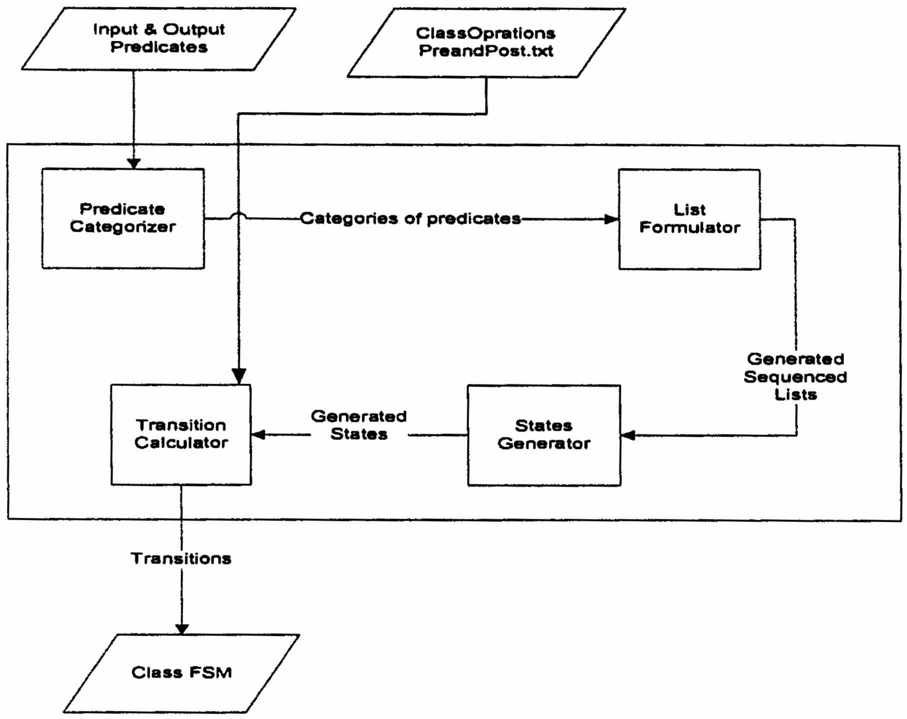


Figure 4.2 Schematic Diagram of the AGFSM

4.1.3 States Generator

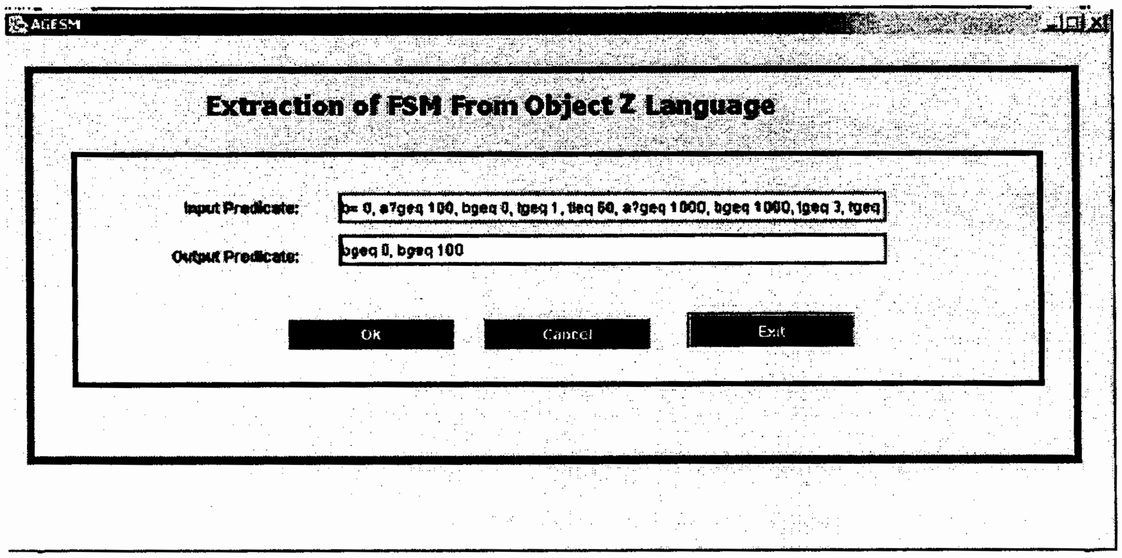
Sequenced lists are taken as input in this component. This component applies a set of rules on the predicates in each sequenced list formulated as discussed in detail in section 3.2.3. Valid set of states are generated as a result. These states are generated to cover all the input and output state space provided by each predicate for a variable in each sequenced list.

4.1.4 Transition Calculator

The states generated from the state generator component, along with the explicit file are provided as input to the transition calculator component. This component calculates the valid set of transitions for every operation of the class with respective pre- and post-states identified, details are provided in section 3.3. These states are mapped along with the transitions to make final Class FSM.

4.2 Snapshots of AGFSM Tool

AGFSM has a very interactive GUI. The main interface of AGFSM is shown in figure 4.3. It contains two input text boxes: 'Input predicates', 'Output predicates' and three buttons: 'Ok', 'Cancel' and 'Exit'.

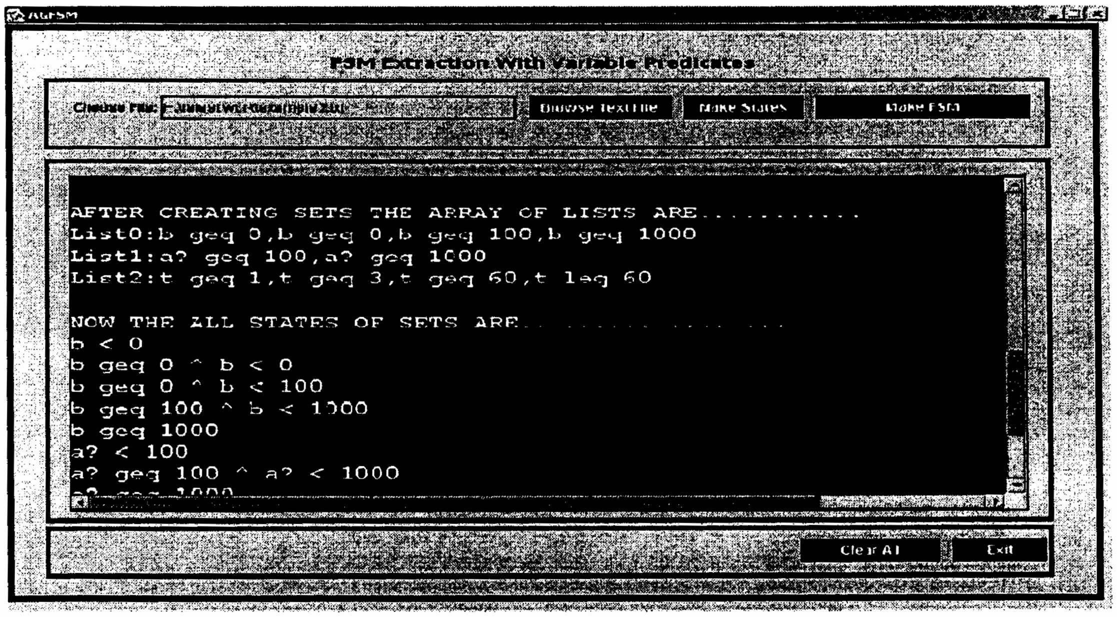


Input Predicate: b= 0, a?geq 100, bgeq 0, tgeq 1, tleq 50, a?geq 1000, bgeq 1000, tgeq 3, tgeq

Output Predicate: bgeq 0, bgeq 100

Figure 4.3 Main GUI- AGFSM Showing Input and Output Predicates Of Bank Account Class

A user can enter input predicates and output predicates of all operations of the Object-Z class. On pressing the 'Ok' button, next interface appears showing the formulated sequenced lists as shown in figure 4.4.



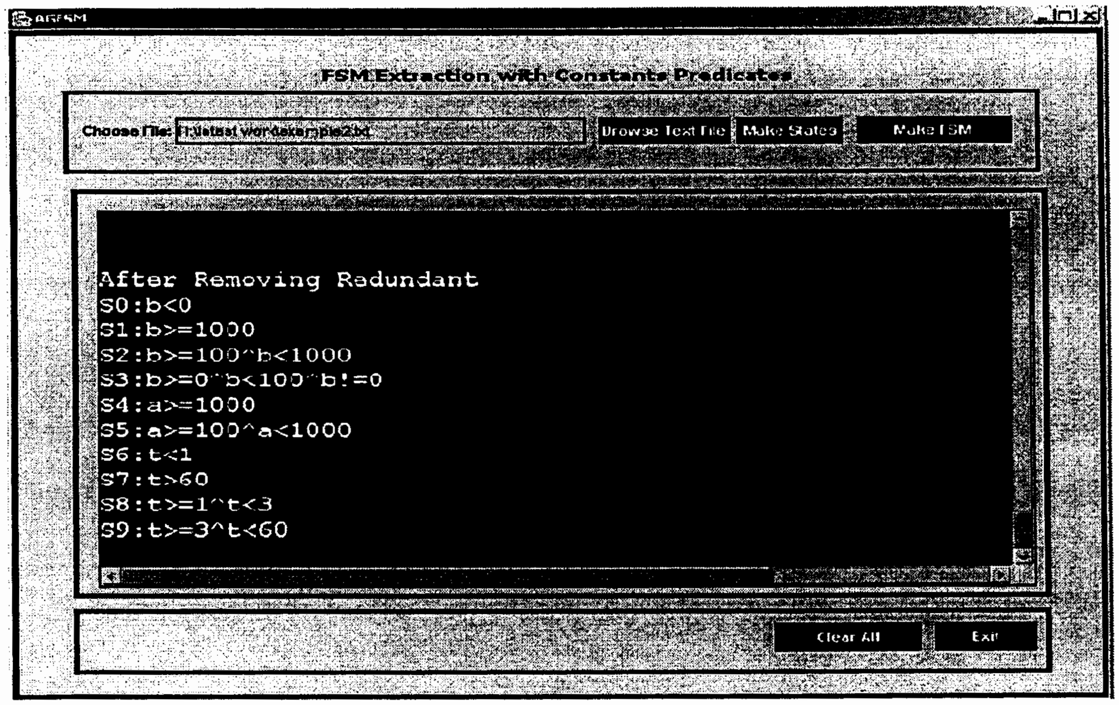
```
AFTER CREATING SETS THE ARRAY OF LISTS ARE...........
List0:b geq 0,b geq 0,b geq 100,b geq 1000
List1:a? geq 100,a? geq 1000
List2:t geq 1,t geq 3,t geq 60,t leq 60

NOW THE ALL STATES OF SETS ARE...............
b < 0
b geq 0 ^ b < 0
b geq 0 ^ b < 100
b geq 100 ^ b < 1000
b geq 1000
a? < 100
a? geq 100 ^ a? < 1000
```

Figure 4.4 Sequenced List Formulation For Bank Account Class

Next interface contains five buttons: 'Make States', 'Browse Text File' and 'Make FSM', 'Clear All' and 'Exit'. Pressing 'Make State' button will identify set of all valid states from each of permuted lists as shown in figure 4.5.

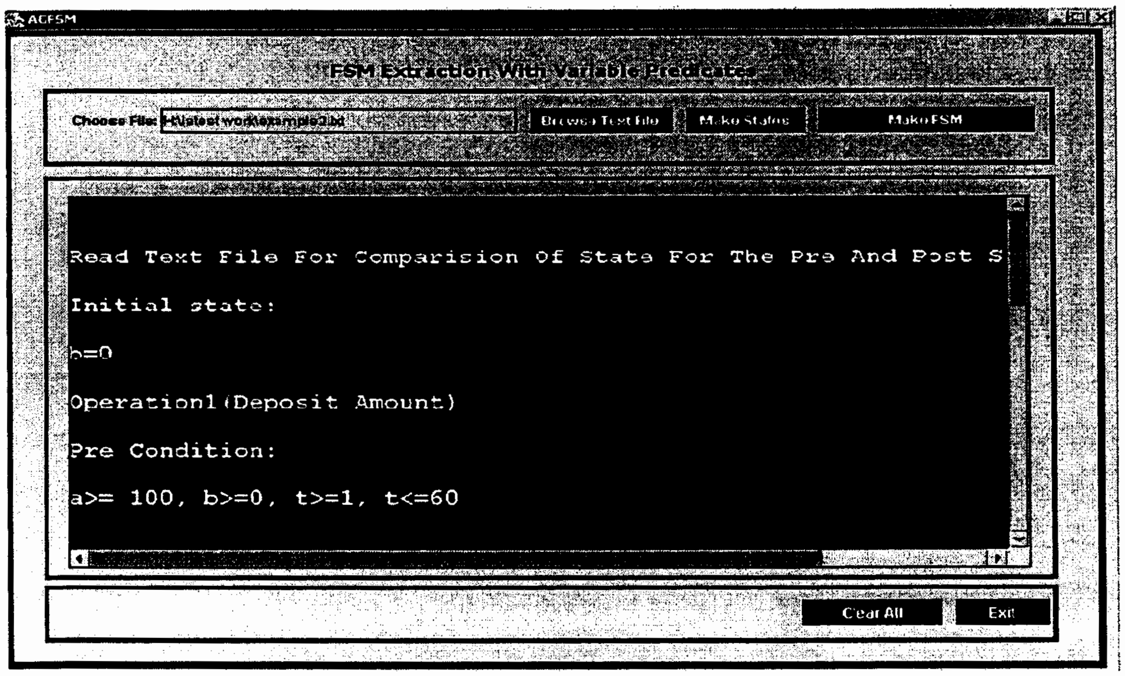
Once the states are identified, the "Browse Text File" button is pressed to browse and select the input file. After the input file is selected, its contents are displayed in the GUI to confirm that the file is successfully selected and read as shown in figure 4.6.

The "Make FSM" button will calculate the transitions by matching the identified states with preconditions and the postconditions provided by the input text file of Object Z class. The states that become the subsets of the preconditions and postconditions become the pre- state and post-state of that particular operation. Transitions for all the operations of class are calculated and shown with the respective pre- and post-states as shown in figure 4.7.



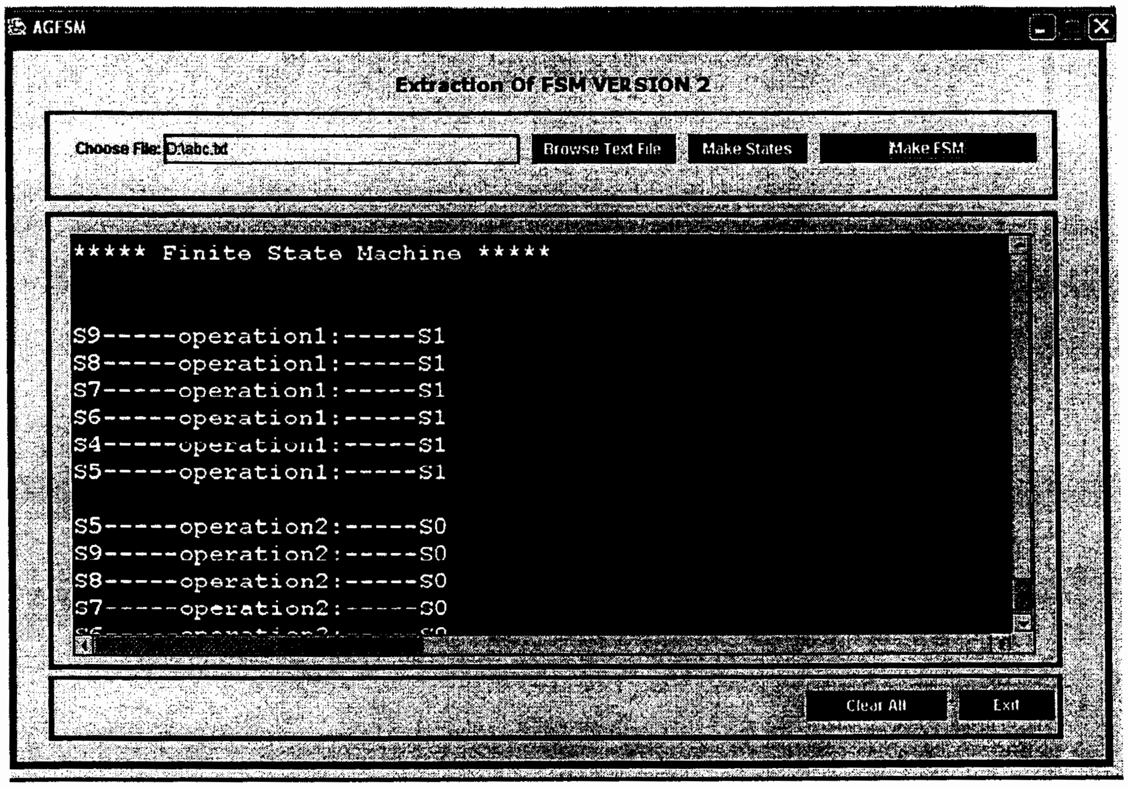
```
After Removing Radundant
S0:b<0
S1:b>=1000
S2:b>=100^b<1000
S3:b>=0^b<100^b!=0
S4:a>=1000
S5:a>=100^a<1000
S6:t<1
S7:t>60
S8:t>=1^t<3
S9:t>=3^t<60
```

Figure 4.5 States Invariants Identified For Bank Account Class



```
Read Text File For Comparision Of State For The Pre And Post S

Initial state:

b=0

Operation1(Deposit Amount)

Pre Condition:

a>= 100, b>=0, t>=1, t<=60
```

Figure 4.6 Reading Text File for Representing Operations of Bank Account Class



```
***** Finite State Machine *****


S9-----operation1:-----S1
S8-----operation1:-----S1
S7-----operation1:-----S1
S6-----operation1:-----S1
S4-----operation1:-----S1
S5-----operation1:-----S1

S5-----operation2:-----S0
S9-----operation2:-----S0
S8-----operation2:-----S0
S7-----operation2:-----S0
```

Figure 4.7 Transitions Calculated With Respective Pre- and Post-States For All Operations of Bank Account Class

5

Evaluation

5. EVALUATION

This chapter evaluates the proposed approach for FSM generation from Object-Z class specifications as well as the developed tool 'AGFSM' as shown in table 5.1. Section 5.1, 5.2, 5.3, 5.4 and 5.5 evaluates the approach against five different parameters.

5.1 Scope/Purpose

The proposed approach is mainly aimed to extract FSM from model-based formal specification. It is obvious from the literature survey that FSM generation has been significant mainly for: Testing, development of implement-able constructs and as a realization of Object-Z specifications.

This work is targeted towards extraction of FSM for testing purpose as the work of Dick & Faivre (1993), Hierons (1997), Carrington (2000), Huaikou & Ling (2005), as well as for developing implement-able constructs as the work of Sun and Dong (2006).

5.2 Specification Language Used

The proposed approach is based on the Object-Z specification language. Other related works are also typically notation specific. However, the approach may be easily extended for other model based formal specifications such as Z.

5.3 State Identification

State identification is a necessary step for the extraction of FSM because the states along with operations help identifying the transitions of FSM. The state generation process in Carrington et al., (2000) and Huaikou et al. (2006) is manual and the refinement of states determined later is based on decision of a tester. Even the removal of resultant duplicate and overlapping states to get disjoint states depends on human intuition. Apart from that, these approaches do not consider the pre and post-conditions that are defined by more than one predicates. Also, no strategy is provided to extract states for a particular variable of simple data type, if defined by more than one relational predicates.

History invariants that are additional set of predicates are used for the extraction of states in the work of Sun et al., (2006). However, these history invariants are not commonly used in the recent versions of Object-Z. The proposed approach only uses the predicates extracted from the given Object-Z Class specification without involving history invariants. This approach too does not discuss coverage of state space defined by multiple predicates for a particular variable. The approaches that extract states from disjoint partitions (Dick et al.,1993, Hierons, 1997), depend on human verdict to decide the simplification level of partitions obtained and to identify distinct and non-overlapping states from them.

A set of distinct and non-overlapping states is extracted from the predicates by applying a sequence of steps. This extraction of states involves no human intervention to decide about the level of refinement of states and for the removal of duplicate and overlapping states. The resultant states not only provide abstraction but also are reachable in the resulting FSM. State Identification parameter has been evaluated on the basis of following sub parameters:

5.3.1 Logical Expression (L.E)

The proposed approach extracts states and transitions from logical expressions including conjunctions used in the Object-Z class specifications.

5.3.2 Relational Expression (R.E)

The approach and AGFSM provides successful extraction of states and transitions from the relational expressions including: less than '<', less than equal to '≤', greater than '>', greater than equal to ' ≥', equal to '=', not equal to '≠'.

This parameter is further evaluated on the basis of three sub-parameters:

a) Defined by Constants

The approach provides extraction of states and transitions from the predicates involved in relational expressions that are used to define a variable in terms of constant values only.

b) Defined by Multiple Variables

Extraction of states and transitions from the predicates in relational expressions which are used to define a variable in terms of other abstract variables is also provided by the proposed approach.

c) Defined by Constants and Variables

States and transitions are extracted from the predicates in relational expressions used to define a variable in terms of constant values and other abstract variables.

5.3.3 Input and Output Predicates

Latif et al. (2008) proposed an approach to separate input and output predicates from a given Z operation schema. Its ultimate goal was to develop states from the extracted pre-post conditions. However, it does not discuss the extraction of states and transitions from the separated predicates. Hence, the work by Latif et al., (2008) has not been included in the evaluation table for states and transition identification of FSM.

The proposed approach particularly addresses the generation of states and transitions for FSM from the extracted input and output predicates of a given Object Z class in detail. An approach is given to identify states from input and output predicates. In addition, the proposed approach also calculates transitions from the identified states and the pre- and postconditions of operations of a class.

5.4 Transition Calculation (T.C)

It is another significant step after the identification of states for making FSM. However, in the existing techniques for FSM generation, the process of determining transitions by mapping an operation to the states satisfying its precondition and post conditions is also manual. However, this work is not only detailed and mechanical but also requires human intuition in deciding about the pre and post states for an operation. The proposed approach provides a systematic way to identify the pre-states and post-states for a particular operation. The states that subsets the regions offered by predicates defining the pre-and post-conditions of an operation becomes the pre-and post-states respectively.

5.5 Automation / Tool Support

Limited tool support has been provided by the formal methods. The tool support that is developed for the proposed approach, AGFSM is a fully automated tool. Hence, the proposed approach is an important development towards the automatic FSM extraction from Object-Z specifications and consequently in specification-based testing techniques as automation in specification based testing is scarce (Offut et al., 2003).

The work of Latif & Nadeem (2008) for the extraction of pre and post conditions for FSM extraction from Z language and did not discuss FSM generation from the separated input and output predicates and consequently, no automation details were discussed as well. The proposed approach is built upon their work. AGFSM automated the extraction of states and transitions of FSM from the input and output predicates from Object-Z specification. Hence, it is a significant improvement towards the automated FSM extraction and automated testing.

Carrington et al., (2000) does not provide any automation details for identification of states from test templates and transition calculation to make an FSM. Huaikou, (2006) generates an FSM from a class using Test Class Framework (TCF), but it too does not provide any automation details for FSM generation from templates generated through TCF. Hence, AGFSM is an important development towards automatic FSM extraction.

Table 5.1 Comparison of FSM Extraction Techniques from Formal Specification

Approach	Specification Language Used	Scope / Purpose	State Derivation					Automation / Tool Support	
			L.E	R.E			IP & OP	S.I	T.C
				Defined by Constant	Defined by Multiple Variables	Defined by Variables and Constants			
Carrington et al. (2000)	Object-Z	Test Case Generation & Execution	N	N	Y	N	N	P	N
Dick & Faivre (1993)	VDM	Test Case Generation & Execution	P	N	Y	N	N	Y	N
Huaikou & Ling (2006)	Object-Z	Test Case Generation & Execution	N	Y	N	N	N	P	N
Hierons (1997)	Z	Test Case Generation & Execution	P	Y	N	N	N	N	N
Sun & Dong (2006)	Object-Z With History Invariants	Synthesizing Implementable Constructs	P	N	N	N	N	P	P
Our Approach	Object-Z	FSM Construction	P	Y	Y	Y	Y	Y	Y

Table 5.2 Abbreviations Used

L.E	Logical Expression
R.E	Relational Expression
IP & OP	Input Predicate and Output Predicate
S.I	State Identification
T.C	Transition Calculation

6

Conclusion & Future Work

6. CONCLUSION & FUTURE WORK

FSM has appeared as an important construct particularly in facilitating testing process i.e. test sequencing, test case generation and test case execution in formal specification-based testing. FSM has also been driven as explicit system behaviors to generate implement-able constructs from abstract high level specifications.

However, FSM extraction from model-based formal specifications is not an easy task. Model-based formal specifications like Object-Z do not explicitly specify pre- and postconditions in class operations. None of the existing techniques for FSM extraction identify states and transitions from input and output predicates that define pre- and postconditions of operation in a class. State generation from the predicates in relational expressions that define a variable in terms of constants and variables has also not considered earlier.

This research is an initial step towards automatic FSM generation from model-oriented formal specification. The proposed approach successfully extracts states and transitions of FSM from an Object-Z class specification. The developed tool, AGFSM, fully automates the proposed approach. The approach is proposed for the Object-Z class specifications including simple data types, simple predicates with relational and some logical operators. However, complex data types like sets, sequences are not considered in research.

As automated testing area is a growing research area. Therefore, this research is an important contribution towards the automation of software testing techniques using FSM based on Object-Z specifications.

6.1 Future Work

Fully automated testing based on automatically extracted FSM can be an important future direction of this research work. Another future direction can be to develop (link) abstract models implementation technique based on our approach.

Other future directions to this research can be to automatically generate FSM for multiple Object-Z classes covering the inter-class relationships like inheritance, generalization and association etc. Also, the approach takes an explicit input representation of class's INIT schema, state invariants and each of its operation's pre- and postcondition for transition calculation, other than class's extracted input and output predicates from TEIOPZ. This is due to the reason that TEIOPZ is mainly developed for single Z operation and provides no relationship between a class's operations and their respective pre- and postconditions and the input and output predicates involved in them. Hence, extension of TEIOPZ for Object-Z class can eliminate the need for explicit representation needed other than Object-Z input and output predicates and can be an important future direction. Moreover, the work can also be explored, whether this approach can be generalized for other model-based formal specification languages.

References

Abrial, J., (1996) The B–Book: Assigning Programs to Meanings, Cambridge University Press.

Bowen, J., Bogdanv, K., Clark, J., Harman, M., Hierons, R., Krause, P.(2002) *FORTEST:* Formal Methods and Testing. In Proceedings of the 26th Annual International Computer Software and Applications Software COMPSAC'02 Washington, DC:IEEE Computer Society, pp.91-104.

Binder, V., R. (2000). Testing object-oriented systems: models, patterns, and tools. Boston, MA: Addison-Wesley. ISBN 0-201-8093-9

Carrington, D., MacColl, I., McDonald, J., Murray, L., & Strooper, P. (2000). From Object-Z specifications to ClassBench test suites. *Software Testing, Verification and Reliability, 10*(2), 111-137.

Carrington, D., MacColl, I., McDonald, J., Murray, L., & Strooper, P. (2003). From Object-Z specifications to ClassBench test suites. Retrieved October 22, 2007, from http://www.itee.uq.edu.au/~comp4600/sbt-paper.pdf

Dick, J., & Faivre, A. (1993). Automating the generation and sequencing of test cases from model-based specifications. In J. Woodcock, & P. G. Larsen (Eds.), *Lecture Notes in Computer Science 670* [Proceedings of the First international Symposium of Formal Methods Europe on industrial-Strength Formal Methods FME '93] (pp. 268-284). Berlin, *Germany*: Springer-Verlag.

Donat, R., M. (1997). Automating Formal Specification-Based Testing. In M. Bidoit & M. Dauchetin (Eds.), *Lecture Notes in Computer Science 1214* [Proceedings of the 7th international Joint Conference CAAP/FASE on theory and Practice of

Software Development TAPSOFT'97] (pp. 833-847). Berlin, *Germany*: Springer-Verlag.

Duke, R., King, K., Rose, G. et al. (1991*)*. The Object Z specification language: version 1 [R].Technical Report No. 91 – 01, Queensland, Australia.

Hierons, M., R. (1997). Testing from a Z specification. *Software Testing, Verification and Reliability, 7*(1), 19–33.

Hierons, M.R. et al. (2009), Using formal specifications to support testing, *ACM Computing Surveys, vol. 41, no. 2.*

Hierons M..R, (2010) Canonical finite state machines for distributed systems, *Theoretical Computer Science, 411 (2010), pp. 566–580*

Hoffman, M., D., & Strooper, A., P. (1995). The testgraph methodology: Automated testing of collection classes. *Journal of Object-Oriented Programming, 8*(7), 35-41.

Hoffman, M., D., & Strooper, A., P. (1997). ClassBench: A methodology and framework for automated class testing. *Software Practice and Experience, 27*(5), 573-597.

Huaikou, M., & Ling, L. (2006). An approach to formalizing specification-based class testing. *Journal of Shanghai University, 10*(1), 25-32.

Huaikou, M., & Ling, L. (2000). A test class framework for generating test cases from Z specifications. In *Proceedings of the 6th IEEE International Conference on Complex Computer Systems ICECCS'00* (pp. 164-171). Washington, DC: IEEE Computer Society.

Huaikou, M., Ling, L., Chuanjiang, Y., Jijun, M., & Li, L. (2001). Z User Studio: An Integrated Support Tool for Z Specifications. In *Proceedings of the Eighth Asia-Pacific Software Engineering Conference APSEC'01* (pp. 437-444). Washington, DC: IEEE Computer Society.

Jacky, J. (1996). The way of Z: Practical Programming with Formal Methods. New York: Cambridge University Press. ISBN 0-521-55976-6

Jones, B., C. (1996), Systematic Software Development Using VDM, Upper Saddle River, NJ: Prentice-Hall. ISBN 0-13-880733-7

Johnston, W. (1996). A type checker for Object-Z. Technical Report 96-24, Software Verification Research Center, The University of Queensland, Australia.

Korel, B. (1990). Automated software test data generation. *IEEE Transactions on Software Engineering, 16*(8), 870-879.

Latif, B., Nadeem, A. (2008), Automatic Extraction of Pre- and Postconditions from Z Specifications, *Studies in Computational Intelligence Series, Vol. 150, ISBN 978-3-540-70774-5,* Springer-Verlag.

Ling, L., Huaikou, M., & Xuede, Z. (2002). A framework for specification-based class testing. In *Proceedings of the 8th IEEE International Conference on Complex Computer Systems ICECCS'02* (pp. 153-162). Washington, DC: IEEE Computer Society.

McDonald, J., & Strooper, P. (1998). Translating Object-Z specifications to passive test oracles. In *Proceedings of the 30th Annual International Computer Software and Applications Software COMPSAC'06* (pp. 101-104). Washington, DC: IEEE Computer Society.

Murray, L., Carrington, D., MacColl, I., & Strooper, P. (1997). Extending Test Templates with inheritance. In *Proceedings of the Australian Software Engineering Conference ASWEC'97* (pp. 80-87). Washington, DC: IEEE Computer Society.

Murray, L., Carrington, D., MacColl, I., McDonald, J., & Strooper, P. (1998). Formal derivation of finite state machines for class testing. In J. P. Bowen, A. Fett, & M. G. Hinchey (Eds.), *Lecture Notes in Computer Science 1493* [Proceedings of the 11th international Conference of Z Users on the Z Formal Specification Notation ZUM'98] (pp. 42-59). Berlin, *Germany*: Springer-Verlag.

Murray, L., Carrington, D., MacColl, I., & Strooper, P. (1999). TinMan — A Test Derivation and Management Tool for Specification-Based Class Testing. In *Proceedings of the 32nd International Conference on Technology of Object-Oriented Languages* (pp. 222-233). Washington, DC: IEEE Computer Society.

Offut, J., Liu, S., Abdurazik, A., & Ammann, P. (2003). Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, *13*, 25-53.

Poston R. M. (1996). Automating Specification Based Sofware Testing [M]. IEEE Computer Society Press, Los Alamitos, California.

Pressman, S., R. (2001). Software Engineering: A Practitioner's Approach (5th ed.). New York: McGraw-Hill. ISBN 0072496681

Smith, G. (2000). The Object-Z specification language, Boston, MA: Kluwer. ISBN 0-7923-8684-1

Sommerville, I. (2000). Software Engineering (5th ed.). Harlow, *England*: Addison-Wesley. ISBN 0-201-42765-6

Spivey, M., J. (1992). The Z Notation: A Reference Manual. New York: Prentice- Hall. ISBN 0-13-978529-9

Spivey, M., J. (2000). *The fUZZ manual* (2nd ed.). Oxford, *England*: The Spivey Partnership. Retrieved May 27, 2008, from http://spivey.oriel.ox.ac.uk/mike/fuzz/fuzzman.pdf

Stocks, P., & Carrington, D. (1996). A framework for specification-based testing. *IEEE Transactions on Software Engineering, 22*(11), 777-793.

Sun, J., & Dong, S., J. (2006). Design synthesis from interaction and state-based specifications. *IEEE Transactions on* Software *Engineering, 32*(2), 349-364.

Sun, J., & Dong, S., J. (2005). Extracting FSMs from Object-Z specifications with history invariants. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems ICECCS'05* (pp. 96-105). Washington, DC: IEEE Computer Society.

Sun, J. (2006). *Complementary Formalisms - Synthesis, Verification and Visualization.* Unpublished doctoral dissertation, Department of Computer Science, School of Computing, National University of Singapore.