Testing Functional Requirements using B Model Specifications

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

Master of Engineering
in
Software Engineering

By:
Amit Gupta
(80731002)

Under the supervision of
Dr. Rajesh Bhatia
Asst. Professor & Head, CSED,
Thapar University, Patiala.

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

JUNE 2009

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Testing Functional Requirements using B Model Specifications"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Dr. Rajesh Kumar Bhatia and refers other researchers' works which are duly listed in the reference section.

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

(Amit Gupta)

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

9/07/09

(Dr. Rajesh Kumar Bhatia)
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by

9/7/09

(Dr. Rajesh Kumar Bhatia)
Assistant Professor & Head
Computer Science & Engineering Department
Thapar University,
Patiala.

13/7/09

(Dr. R.K. Sharma)
Dean (Academic Affairs)
Thapar University,
Patiala.

i

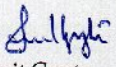
Acknowledgment

I wish to express my deep gratitude to Dr. Rajesh K. Bhatia, Head, Assistant Professor, Computer Science & Engineering Department, TU, Patiala for providing his uncanny guidance and support throughout the thesis.

I am thankful to Dr. (Mrs). Seema Bawa, Computer Science & Engineering Department, TU, Patiala, for the motivation and inspiration that triggered me for the thesis work. I would also like to thank all the staff members who were always there at the need of the hour and provided with all the help and facilities, which I required for the completion of the thesis.

Last but not the least, I express my heartfelt thanks to my parents and all of my friends for encouraging me and providing me useful information during my work.

Finally, my special thanks go to authors whose works I have consulted and quoted in this work.



Amit Gupta
80731002
M.E.(Software Engineering)-2nd year
Computer Science & Engineering Department
Thapar University
Patiala -147004

ii

Abstract

Testing is very important part of software development. Almost 80% software fails because of the improper or inefficient testing. Testing is performed by different types of strategies. Generally testing is performed on code, but if the software can be tested in the earlier phases then most of the errors can be eliminated and can be stopped from propagating to next phase. Thus there is a need to explore testing possibilities in earlier phases.

The proposed work presents a novel requirement based testing approach that can fix errors in initial phase. Formal Specification languages play a vital role in software testing. Formal models provide a precise specification of the system, and can be used as a vehicle for driving the development process. To perform requirement based testing, we need a formal language that can deal with the requirement specification efficiently.

Many researchers have proposed various approaches to generate test cases from formal specifications. These approaches include test case generation from various state based languages like Z, VDM and B specifications. We investigate, why B specification is more appropriate than Z and VDM for test case generation in proposed work. We also compare test case generation from above specification languages and find limitations of each approach.

In this thesis work we proposed a technique that can provide better coverage of requirements as compared to other approaches. For maximizing the coverage of requirements in our model, we annotate our specifications with requirement identifiers, which help in later stages to detect which requirements are covered and which are yet to be tested. Test cases are generated by extracting invariants and postconditions from our specification, and are transformed in a generalized form. Using test selection criteria, we can cover all parts of our model and generates test cases for each of our test objective.

Keywords: Software Testing, Formal Languages, Z, VDM, B Specifications.

Abbreviations

SUT	System under Test
RBT	Requirement based testing
tcs	Test case specifications
TTF	Test Template Framework
SRS	Software requirement specification
SDS	Software Design Specification
FSL	Formal Specification Languages
VDM	Vienna Development Method
invar	Invariant
init	Initialization

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

IEEE definition of software testing is executing the program with intent of finding errors. Testing is the most critical phase in the software development life cycle [1, 2, 7]. The testing phase is the final filter for all errors of omission and commission. Almost 50%-70% of the software projects fail due to the improper or inefficient testing. Almost 50% of the software production development cost is expended in software testing. It uses resources and in return, it adds nothing to the product in terms of its functionality. Software testing remains the primary concern used to gain consumer's confidence in the software. Ideally, testing of software guarantees the absence of errors in the software, but in reality it only discloses the presence of software errors, but never guarantees their absence. Even, systematic testing cannot prove absolutely the absence of errors, which are detected by discovering their effect. One objective of software testing is to find errors and program structure faults.

Testing software is far more complex than exercising a program to see if it works. Software testing is an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate. Quality is not an absolute term; it is value to some person. With that in mind, testing can never completely establish the correctness of arbitrary computer software; testing furnishes a criticism or comparison that compares the state and behavior of the product against a specification.

A common practice of software testing is performed by an independent group of testers after the functionality is developed and before it is shipped to the customer. This practice often results in the testing phase being used as buffer to compensate for project delays, thereby compromising the time devoted to testing. Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes. Each review, inspection, audit, walk-through and group code read, all is in reality a form of test.

1.1 Testing Objectives

Testing is very important part of software development process. It takes half of the time of the software development. The main objective of testing is to prove that the software product as a minimum meets a set of pre-established acceptance criteria under a prescribed set of environmental circumstances. Testing is performed on the requirements, code and design with different methodologies. But still failure rate of the software project is about 80%. The reason for failure is because the testing performed on the software is not sufficient to detect various types of errors in the software. Generally practitioners consider code testing as most important. So testing only code and ignoring all other artifacts may result in catastrophic type of failure.

Requirement phase is very important part because whole design and coding part depend on this phase. But generally testing of software requirement is avoided or done partially. Because requirement is not executable part, so testing of a requirement becomes very cumbersome. Our objective is to prove that the requirements specification from which the software was designed is correct. Correctness means that functional, performance, and timing requirements match acceptance criteria. Software testing is further complicated by the fact that system acceptance criteria usually involve hardware, procedures, and operators so that acceptance tests involve more than just the software.

Following are the objectives that software testing follows:

- A good test case is one that has a high probability of finding an as-yet undiscovered error.
- A successful test is one that uncovers an as-yet undiscovered error.
- Testing cannot show the absence of defects, it can only show that software errors are present.

1.2 Testing Principles

Following are principles of Software Testing [2]:

- *All tests should be traceable to customer requirements*. The objective of system testing is to uncover errors. It follows that the most severe defects are those that cause the program to fail to meet its requirements.
- *Tests should be planned long before testing begins*. Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.
- *Testing should begin "in the small" and progress toward testing "in the large".* The first test planned and executed generally focus on individual program modules. As testing progresses, testing shifts focus in an attempt to find errors in integrated clusters of modules and ultimately in the entire system.
- *Exhaustive testing is not possible.* The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing.
- *To be most effective, testing should be conducted by an independent third party.* By "most effective" , means testing that has the highest probability of finding errors. For this reason, the software engineer who created the system is not the best person to conduct all tests for the software.

1.3 Characteristics of a "Good" Test

Following are the characteristics of a good test [2]:

- *A good test has a high probability of finding an error.* To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.
- *A good test is not redundant*. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose.
- *A good test should be "best of breed".* In a group of tests that have a similar intent, time and resource limitations may militate for the execution of only a subset of these tests.

- *A good test should be neither too simple nor too complex*. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

1.4 Testing Process

The IEEE829 standard [3] describes a framework within which the entire testing process can be managed. The framework allows easy communication between members of a testing project, organizes the testing process, and outlines the documents that should be made part of any compliant testing process. Figure 1.1 shows all the activities of software testing according to IEEE829.

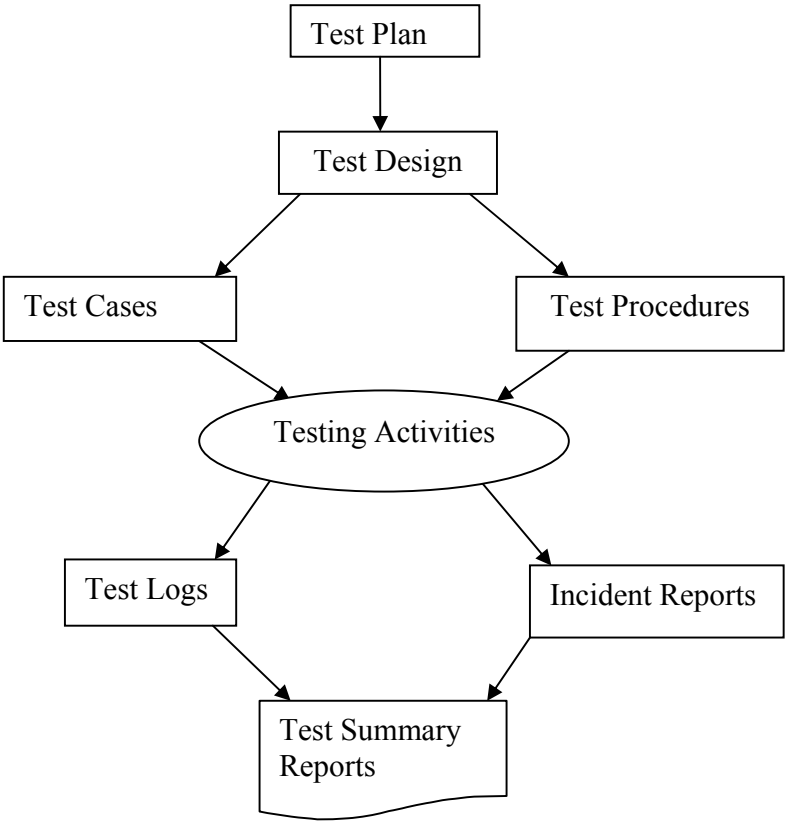


Figure 1.1: Testing activities

1.4.1 Test Plan

Test plan describes the scope, approach, resources, and schedule of testing activities in a given project, and identifies the items to be tested, the features of those items to be tested, the individual testing tasks that are to be performed, and personnel responsible for those tasks, along with the risks associated with the plan.

1.4.2 Test Design

Test design, or test specification as it is sometimes known, further refines the testing approach, and identifies the test cases and test procedures, and test item pass/fail criteria.

1.4.3 Test Cases

Individual test cases document the values that will be used as input to individual tests, together with the associated expected outputs. A test case identifies any constraints on the test procedure resulting from the use of the test case.

1.4.4 Test Procedures

Test procedure describes the exact steps required to operate the system and execute test cases in order to implement the test design. Test procedure is kept separate from the test design as it is followed step by step.

1.4.5 Test Logs

Test logs are used to record what occurred during execution of a test or set of tests. Test logs may either be manually created as tests are executed, or automatically by the system as testing processes.

1.4.6 Incident Reports

Incident reports are used to provide a description of any events that occur during testing that require further investigation.

1.4.7 Test Summary Report

Test summary report summarizes the testing activities associated with one or more test designs.

1.5 Software Testing Approaches

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing, a set of steps into which we can place specific test case design techniques and testing methods, should be defined for the software process. A number of software testing approaches have been proposed in the literature.

1.5.1 Top-Down Approach

The top-down approach is based on establishing the top-level control structure first. In top-down strategy, testing starts from the top of the hierarchy, and then incrementally adds modules that it calls and tests the new combined system. This approach of testing requires stubs to be written. A **stub** is a dummy routine that simulates a module [2]. In the top-down approach, a module (or a collection) cannot be tested in isolation because they invoke some other modules. To allow the modules to be tested before their subordinates have been coded, stubs simulate the behavior of the subordinates.

1.5.2 Bottom-up Approach

The bottom-up approach starts from the bottom of the hierarchy. First the modules at the very bottom, which have no subordinates, are tested [5]. Then these modules are combined with higher-level modules for testing. At any stage all the subordinate modules exist and have been tested earlier. To perform bottom-up testing, drivers are needed to set up the appropriate environment and invoke the module. It is the job of the **driver** to invoke the module under testing with the different set of test cases.

1.6 Software Testing Techniques

Testing techniques refers to the way the software artifacts are examined during the software development life cycle.

1.6.1 Static Testing

The term *static testing* refers to testing the software requirement specification (SRS), software design specification (SDS) and other non-executable items through requirement analysis, audits, desk checks, inspections, walk-through, etc [1]. Static testing is employed to verify the correctness of requirements, designs, and code before execution of test cases. Static testing will also audit newly developed or reused code for adherence to established standards.

A successful static test of a software module depends upon several things going right [8]:

1. A correct allocation of requirements to the software components.
2. A correct partitioning and sub allocation of software requirements to the module.
3. Successful (correct) module design.
4. A successful translation of the intermediate code (pseudo-code, POL, etc.) into programming language statements.

The purposes of the static testing are:

- Validating the requirement specifications.
- Looking for omissions, inconsistencies, redundancies in all documents and source code.
- To ensure that the documents of design and coding conform to the specification.

1.6.2 Dynamic Testing

Dynamic testing is a term that describes the development of test cases and test procedures, the execution of test cases, and the structure and use of test logs and anomaly or incident reports. There are two popular ways to perform dynamic testing, namely, black box testing and white (glass) box testing. Either of these two methods requires a set of well-developed and well-structured test cases. Dynamic testing cannot prove the absolute correctness of a software product unless it is performed in

an exhaustive manner [4]. An exhaustive test requires a set of test cases that guarantees the following: explicitly exercises every possible, module path and every possible combination of paths with every possible module input and every possible combination of module inputs.

1.7 Types of Testing

Any engineered product can be tested in one of two ways:(1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function; (2) knowing the internal workings of a product, tests can be conducted to ensure that internal operations are performed according to specifications and all internal components have been adequately exercised. The first test approach is called black-box testing and the second, white-box testing.

1.7.1 Black Box or Functional Testing

Black box testing is the testing of a piece of software without regard to its underlying implementation [9]. Specifically, it dictates that test cases for a piece of software are to be generated based solely on an examination of the specification (external description) for that piece of software. The goal of black box testing is to demonstrate that the software being tested does not adhere to its external specification.

The objective is to search for interface errors, function or process errors, performance shortcomings, start-up/shutdown errors, and errors in local (module) databases by selecting appropriate inputs and external conditions and monitoring outputs.

Black box testing attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Performance errors

- Initialization and termination errors

Black Box Testing Techniques includes:

- Equivalence Partitioning
- Boundary Value Analysis
- Comparison Testing

1.7.2 White Box Testing

White box testing also known as *glass box testing* .It is a test case design method that uses the control structure of the procedural design to derive test cases [10]. Using white box testing methods, the software engineer can derive test cases that:

1. Guarantee that all *independent paths* within a module have been exercised at least once.
2. Exercise all logical decisions on their true and false sides.
3. Execute all loops at their boundaries and within their operational bounds.
4. Exercise internal data structures to assure their validity.

Thus white box testing is the testing of the underlying implementation of a piece of software (e.g., source code) without regard to the specification (external description) for that piece of software [6]. The goal of white box testing of source code is to identify such items as (unintentional) infinite loops, paths through the code which should be allowed, but which cannot be executed and dead (unreachable) code.

White Box Testing Techniques includes:

- Control flow testing.
- Data flow testing.
- Basic path testing.

1.8 Requirement Based Testing

The requirements-based testing process addresses two major issues: first, validating that the requirements are correct, complete, unambiguous, and logically consistent; and second, designing a necessary and sufficient, from a black box perspective, set of test cases from those requirements to ensure that the design and code fully meet those requirements. In designing tests two issues need to be overcome: reducing the immensely large number of potential tests down a reasonable size set and ensuring that the tests got the right answer for the right reason. The requirement testing process does not assume, going in, that we will see good Requirements Specifications. That is very rarely the case. The requirement testing process will drive out ambiguity and drive down the level of detail [11].

The overall requirement testing strategy is to integrate testing throughout the development life cycle and to focus on the quality of the Requirements Specification. This leads to early defect detection which has been shown to be much less expensive than finding defects during integration testing or later. The requirement testing process also has a focus on defect prevention, not just defect detection. Demonstrating test coverage of requirements is important in regulated business such as health care, safety-critical software etc. Effective test coverage requires not only a well organized testing effort, but also requires testable requirements. Requirements are often reported as not suitable for testing, because they are, for instance, incomplete or too ambiguous.

1.8.1 Reasons for Requirement Testing

There are numerous economic reasons to improve the quality of software by testing the requirements:

- Reducing the costs to detect and remediate defects.
- Reducing the time it takes to deliver the software and
- Improving the probability of successfully installing the right solution.

1.8.2 Relative Cost to Fix an Error

The reason for integrating testing earlier into the life cycle has simple economic reasons. First, that the majority of defects have their root cause in poorly defined requirements as shown in Figure 1.2.

Distribution of Bugs **Distribution of Efforts to fix Bugs**

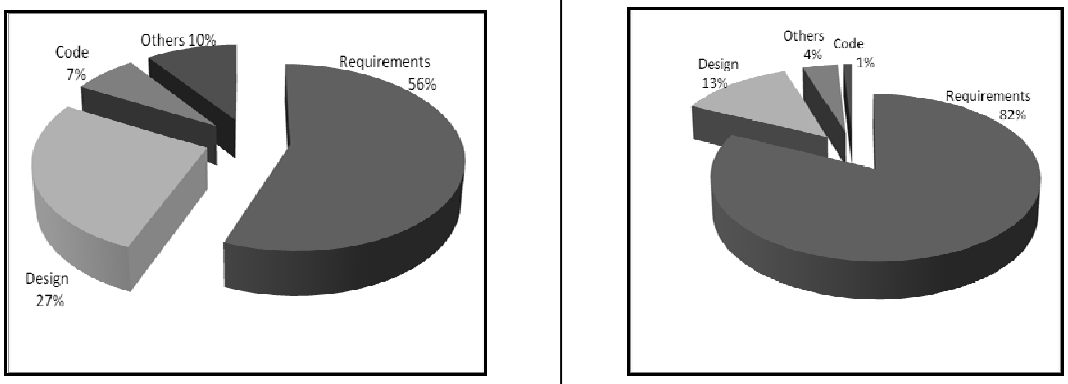


Figure 1.2. Distribution of Defects.

And second that the cost of fixing an error is cheaper, the earliest it is found. If a defect was introduced while coding, we just fix the code and re-compile it. However, if a defect has its roots in poor requirements and is not discovered until integration testing then you must re-do the requirements, re-do the design, re-do the code, re-do the tests, and re-do the user documentation. It is all this "re-do" work that sends projects over budget and over schedule. Let's say that a defect introduced during the requirements phase is found during the requirements phase, and the cost of finding that defect is 1X. If that same defect is not found until integration testing or production it will cost hundreds or even thousands of times more. Table1.1 shows relative cost to fix an error during different phases.

Table1.1 Relative Cost to Fix an Error.

Phase in which Defect is Found	**Cost Ratio**
Requirements	1
Design	3-6
Coding	10
Unit/Integration Testing	15-40
System/Acceptance Testing	30-70
Production	40-1000

1.9 Formal Specification and Software Testing

Traditionally formal methods and software testing have been seen as rivals. That is, they largely failed to inform one another and there was very little interaction between the two communities. In recent years, however, a new consensus has developed. Under this consensus, these approaches are seen as complementary [12, 13]. The use of a formal specification or model eliminates ambiguity and thus reduces the chance of errors being introduced during software development. Where a formal specification exists, both the source code and the specification may be seen as formal objects that can be analyzed and manipulated. A formal specification could be analyzed in order to explore the consequences of this specification and potentially find mistakes [14]. If this is done then we have greater confidence that we are testing the *system under test (SUT)* against the *actual* requirements.

The use of a formal specification introduces the possibility of the formal and potentially, automatic analysis of the relationship between the specification and the source code. This is often assumed to take the form of a proof, but such a proof cannot guarantee operational correctness. Software testing is an important and, traditionally, extremely expensive part of the software development process, with the importance and cost depending on the nature and criticality of the system. Studies have suggested that testing often forms more than 50% of the total development cost, and hence dominates the overall production cost. Where formal specifications and

models exist, these may be used as the basis for automating parts of the testing process and this can lead to more efficient and effective testing [16]. It may transpire that its support for test automation is one of the most significant benefits of formal model building. The links between testing and formal methods do, however, go well beyond generating tests from a formal specification.

1.9.1 Formal Specification Languages

Formal specification languages are mathematically based languages whose purpose is to aid the construction of systems and software. Often backed by tool support, they can be used to both describe a system and also then to analyze its behavior, possibly verifying key properties of interest. The engineering rationale behind formal methods is that time spent on specification and design will be repaid by a higher quality of product, this contributing to the commercial rationale of trying to reduce the cost of rework later on. The primary idea behind a formal method is that there is benefit in writing a precise *specification* of a system, and formal methods use a formal or mathematical syntax to do so. A specification of a system might cover one or more of a number of aspects, including its functional behavior, its structure or architecture, or even cover aspects of non-functional behavior such as timing or performance criteria.

A precise specification of a system can be used in a number of ways. First, it can be used as the process by which a proper understanding of the system can be articulated, thereby revealing errors or aspects of incompleteness. The specification can also be analyzed or it can be verified correct against properties of interest. A specification can also be used as a vehicle for driving the development process, either through refining the specification toward code or by direct code generation. Of course, a key aspect of the development process is testing, and a specification can also be used to support the testing process.

1.9.2 Model-Based Languages

There are a number of different ways to write a precise specification. One approach is to build a *model* of the intended behavior, and languages such as Z,

VDM, and B do so by describing the states the system could be in together with operations that change the state. The states of the system are typically described using sets, sequences, relations, and functions, and operations are described by predicates given in terms of pre and postconditions.

1.9.3 Finite State-Based Languages

Model-based languages such as Z, VDM, and B can describe arbitrarily general systems, and have potentially infinite state. Finite state-based languages define their state from a finite set of values, which are often presented graphically with state transitions representing changes of state akin to operations in a notation such as Z. Examples of such languages include finite state machines (FSMs) [18,19], SDL, Statecharts, and X-machines. FSM-based test techniques have been considered when testing from such specifications. Much of the work on testing software from an FSM has been motivated by protocol conformance testing, since FSMs are suitable for specifying the control structure of a communications protocol. However, more recently, FSM based testing has been used within an approach called *model-based testing* in which a model is produced in order to drive testing.

1.9.4 Process Algebra State-Based Languages

Concurrency can be given a very elegant algebraic treatment, and *process algebras* describe a system as a number of communicating concurrent processes. Examples include CSP [20], CCS, and LOTOS. Finite-state based languages such as Statecharts and SDL can also be used to describe a system as a set of communicating concurrent processes. However, process algebras have a rich theory that provides alternative notions of conformance described in terms of implementation relations. The implementation relations capture several types of observations that can be made, in addition to traces (sequences of inputs and outputs), and different properties of the environment. CSP, for example, describes a system as a collection of communicating *processes* running concurrently and synchronizing on *events*.

1.9.5 Hybrid Languages

Many systems are built with a combination of analog and digital components. In order to specify and verify such systems it is necessary to use a specification language that encompasses both discrete and continuous mathematics. There has been recent interest in these *hybrid languages*, such as CHARON [22]. A simple example of a nonlinear hybrid system is that of a *temperature controller*. The temperature of a room is controlled through a thermostat which continuously senses the temperature and turns the heater on and off.

1.9.6 Algebraic Languages

Process algebras are amenable to algebraic manipulation; however, there are also languages which describe a system solely in terms of its algebraic properties. These algebraic specification languages describe the behavior of a system in terms of *axioms* that characterize its desired properties. Examples of algebraic specification languages include OBJ [23] and the Common Algebraic Specification Language (CASL) [21]. In mathematical terms algebra (or an *algebraic system*) consists of (1) a set of symbols denoting values of some type, referred to as the *carrier set* of the algebra; and (2) a set of operations on the carrier set.

1.10 Formal Specification based Software Testing

Software testing is mostly about *empirically* checking correctness. Formal methods, on the other hand, have traditionally been about *formally* verifying the correctness of software [15]. A major thrust of formal methods is the introduction of system models early in the lifecycle, against which the software can be proven through the use of appropriate mathematics. By using formal methods and testing together, we can reduce the cost of development by applying testing techniques much earlier in the lifecycle while defects are relatively inexpensive to correct. We might also be able to automate more of the testing process by:

- generating functional test cases from the specification of a system;
- deriving provably correct oracles for checking the results of tests.

This means that formal methods and testing will always be two complementary techniques for the reduction of errors in computer based systems.

1.11 Thesis Organization

Chapter 2 is the review and analysis of works done related to formal specification based languages. In this chapter we will also investigate test case generation from formal specification languages like Z, VDM and B.

Chapter 3 discovers the gap between the techniques discussed in Chapter 2. In this Chapter Problem Definition is formed for the thesis work.

Chapter 4 discusses the solution of the problem formed in Chapter 3. It also discusses the methodology with examples and experimental results are shown.

Chapter 5 concludes the thesis work and also discusses the future scope of the thesis.

Chapter 2: Literature Review

In this chapter we will investigate different kinds of Formal state-based Languages like Z, VDM, and B. We explored test case generation from these formal specification languages through subsections 2.1 to 2.7.

Formal methods allow, applying mathematics (set theory and logic) to specify, design and implement software. The resulting code can be proved to be consistent with the original specification. Formal specifications use mathematical notation to describe in a precise way the properties which an information system must have, without constraining the way in which these properties are achieved. Formal Specifications describe what the system must do without saying how it is to be done. This abstraction makes formal specifications useful in the process of developing a computer system; because they allow questions about what the system should does in a concise manner [17].

A formal specification can serve as a single, reliable reference point for those who investigate the customer's needs, those who implement programs to satisfy those needs, those who test the results, and those who write instruction manuals for the system. Because it is independent of the program code, a formal specification of a system can be completed early in its development. Although it might need to be changed as the design team gains in understanding and the perceived needs of the customer evolve, it can be a valuable means of promoting a common understanding among all those concerned with the system.

2.1 Z Specification

Z Language applies typed sets, relations and functions within the context of first-order predicate logic to build Schemas, a means of structuring a formal specification. The main ingredient of Z is a way of decomposing a specification into small pieces called schemas. Z specifications are organized as a set of schemas- a box like structure that introduces variables and specifies the relationship between these variables [24, 25]. A schema is essentially the formal specification analog of the

programming language component. In the same way that components are used to structure a system, schemas are used to structure a formal specification. By splitting the specification into schemas, we can present it piece by piece. A Schema describes the stored data that a system accesses and alters. In the context of Z, this is called "state". In Z, schemas are used to describe both static and dynamic aspects of a system.

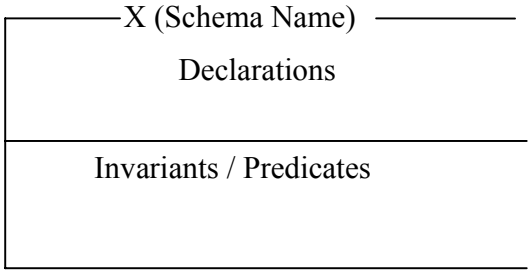
The static aspects include:

- the states it can occupy;
- the invariant relationships that are maintained as the system moves from state to state.

The dynamic aspects include:

- the operations that are possible;
- the relationship between their inputs and outputs;
- the changes of state that happen.

Z *schema,* describe a specification's state space and operations. A schema groups variable declarations with a list of predicates that constrain the possible value of a variable. In Z, the schema X is defined by the form given below, where X is the schema name.



The declaration gives the type of the function or constant, while the predicate gives it value. The schema consists of two parts. The part above the central line represents the variables of the state, while the part below the central line describes the data invariant. Whenever the schema representing the data invariant and state is used

in another schema it is preceded by the Δ symbol. As the last sentence implies, schemas can be used to describe operations, the inclusion of ΔX results in all variables that make up the state being available for the another schema and ensures that the data invariant will hold before and after the operation has been executed. In Z, an input variable that is read from and does not form part of the state is terminated by a question mark (?), and the output variable in schema is terminated by (!).

2.2 Test case Generation from Z Specification

Test Case Generation based on formal model-based specifications has been considered in the literature in the context of the Z, VDM and B notation. In this section we study various test case generation approaches used earlier for Z specification.

In a research by Amla and Ammann et al. in [28], have proposed, test case generation by partitioning the input space of the Z specification. Partition testing is a standard strategy in which the input space is partitioned into sub-domains, and test cases are drawn from each subdomain. The technique for test case generation are based on the principle of partitioning the input domain into equivalence classes on the basis of the specification These equivalence classes are subdomains of the input space upon which it is assumed that the behavior of the system under test is uniform and thus the techniques relate to *uniformity hypothesis*. A small sample of data is then selected from each class: if the uniformity hypothesis holds then it is sufficient to choose one datum in each class.

Hierons et al. in [29] gives an approach for testing from a Z specification by classifying the test domains. The idea is based on considering various combinations after partitioning of the input and output sets (domains) and states into subsets (subdomains), which typically appear in the declaration part of a specification, and conditions contained in the predicates of the specification. This approach is structured, but less rigorous as simple partitions of the input space are constructed by examining the obvious divisions of the inputs.

Simon Burton in [30] also proposed test case generation from partitioning of input space of an operation into equivalence classes, but he extended the approach by formally specifying the test heuristics, in order to improve test case generation. Testing heuristics are formally specified and an automated theorem prover, CADiZ, is used to apply the heuristics to specifications and generate test data from the resulting test cases. The formal approach to specifying and applying the testing heuristics allows properties of the heuristics and resulting test cases to be guaranteed. In his research he shows how the framework supports both partitioning and fault-based heuristics. It was shown how the formal specification of both partitioning and fault-based heuristics can be used as a mechanism for both comparing the relative fault finding abilities of different testing heuristics and assessing the testability of a given specification. A set of abstract test cases is produced from the specification based on various partitioning heuristics and fault hypotheses. The test cases are themselves represented as operations in Z. This allows test data to be generated using conventional constraint solving techniques and provides a mechanism for deriving tests based on more than one heuristic (heuristics can be repeatedly applied to refine the resulting test cases). An abstract test case is defined by constraining the input space of the specification to particular subdomains considered interesting for testing purposes. This is achieved by conjoining *constraining predicates* to the predicate part of the schema operation. Abstract test cases are derived based on common testing practice (such as boundary value analysis or disjunction partitioning) or experience of faults detected in previous builds and systems. Heuristics can be used to partition the input space into disjoint equivalence classes. It is then assumed that selecting data from each class will increase the probability of detecting an error. The constraining predicates define which subdomain is being considered in each test case. Abstract test cases can also be constructed by simply constraining the input space to those values that are considered likely to detect specific errors (fault-based testing). In this case, the constraining predicate will represent the necessary condition for detecting the hypothesized fault.

As an example, the following schema defines the operation *SquareRoot* which calculates the integer square root (*r!*) of its input (*n?*). Where an exact root is not

available, the greatest integer value is chosen that when multiplied by itself does not exceed the input. The variable j is an intermediate value used in the calculation. If black-box testing were being performed, the input variable (n?) in the schema would relate to the input of the implementation under test and the output variable (r!) would relate to the observable result of the test. Intermediate variables would be hidden from the test environment.

SquareRoot

$$n? : \mathbb{Z}$$
$$r! : \mathbb{Z}$$

$$n? \geq 0 \land$$
$$r! * r! \leq n? \land$$
$$\neg(\exists j: \mathbb{Z} \mid j > r! \bullet j * j \leq n?)$$

Partitioning heuristics can be based on either the signature (variable type declarations) or the predicate part of the specification. In the latter case, partitioning information is typically derived from input predicates, predicates that do not constrain output variables. Input predicates may involve intermediate variables but only those that are not used in the definition, either directly or indirectly, of output variables in other predicates. The only predicate that satisfies this criterion is $n? \geq 0$. A boundary value analysis partitioning heuristic can be applied to produce the following constraining predicates: $n? = 0$, $n? = 1$ and $n? > 1$. The resulting test cases would be as shown below. The constraining predicate is highlighted in each case.

Partition1

$$n? : \mathbb{Z}$$
$$r! : \mathbb{Z}$$

$$\boxed{n? = 0 \land}$$
$$n? \geq 0 \land$$
$$r! * r! \leq n? \land$$
$$\neg(\exists j : \mathbb{Z} \mid j > r! \bullet j * j \leq n?)$$

Partition2

$n? : Z$

$r! : Z$

$n? = 1 \wedge$

$n? \geq 0 \wedge$

$r! * r! \leq n? \wedge$

$\neg(\exists j : Z \mid j > r! \bullet j * j \leq n?)$

Partition3

$n? : Z$

$r! : Z$

$n? > 1 \wedge$

$n? \geq 0 \wedge$

$r! * r! \leq n? \wedge$

$\neg(\exists j : Z \mid j > r! \bullet j * j \leq n?)$

Partitioning heuristics can be represented as theorems in Z that describe the equivalence between the original specification and the resulting abstract test cases. The following template is used for defining partitioning heuristics where the input space defined by P is partitioned into the subdomains $P_1 \ldots\ldots P_n$ and Vars (P) is a function that returns declarations of all variables referenced in the predicate to be partitioned, P. Table 2.1 shows formalised partitioning heuristics for boundary values. This template does not require the subdomains to be disjoint, only complete (this allows multiple testing criteria to be combined within the same heuristic). The partitioning heuristic template is thus:

$$\models \forall \text{ Vars}(P) \bullet P \Leftrightarrow P_1 \vee \ldots\ldots \vee P_n$$

To generate abstract test cases using the partitioning heuristics, the parameters of the heuristic are instantiated with the operands of the predicate to be partitioned. The resulting terms of the disjunct on the right hand side of the equivalence are then used as the constraining predicates.

Table2.1: Formalized Partitioning Heuristics.

Group 1: Typed expressions as operands to the partitioning heuristic	
1: Boundary value analysis applied to $<$	$\forall A, B : Z \bullet A < B \Leftrightarrow A = B-1 \vee A < B-1$
2: Boundary value analysis applied to $>$	$\forall A, B : Z \bullet A > B \Leftrightarrow A = B+1 \vee A > B+1$
3: Boundary value analysis applied to $\leq$	$\forall A, B : Z \bullet A \leq B \Leftrightarrow A = B \vee A = B-1 \vee A < B-1$
4: Boundary value analysis applied to $\geq$	$\forall A, B : Z \bullet A \geq B \Leftrightarrow A=B \vee A = B+1 \vee A > B+1$

Group 2: Predicates as operands to the partitioning heuristic	
5: Disjunction partitioning	$\forall A,B : Boolean \bullet A \vee B \Leftrightarrow (A \wedge B) \vee (\neg A \wedge B) \vee (A \wedge \neg B)$
6: Implication partitioning	$\forall A,B : Boolean \bullet A \Rightarrow B \Leftrightarrow \neg A \vee (A \wedge B)$

2.3 VDM Specification

VDM (Vienna Development method) owes its existence to the IBM laboratory in Vienna. VDM is a formal method for the description and development of computer systems. Its formal descriptions or specifications use mathematical notation to

provide a precise statement of the intended function of a system [26]. Such descriptions are built in terms of models of an underlying state with a collection of operations. VDM specification uses logic of propositions for specifying the rule of inferences, without any additional hypothesis. VDM specifies the propositional logic to substitute for arbitrary expressions in a clear and precise manner.

The Vienna group creates an operational semantics approach capable of defining the whole of PL/I language. This meta language forms the part of VDL (Vienna Definition Language). The attempt to use VDL definition in requirement specification and design, showed that the operational semantic approach used in the VDL could complicate formal reasoning in an unnecessary way, which makes the process more complex, as compare to other formal specification languages.

2.4 Test case Generation from VDM Specification

Dick and Faivre in [31] demonstrate that, by rewriting the precondition and postcondition of a VDM specification to Disjunctive Normal Form (DNF), much of the partitioning process can be automated. They also describe a technique for extracting a finite automaton (FA) from the specification; the FA can be used to drive test execution. It consists of, generating abstract finite state automata from the formal model by partition analysis of the state space and operations. and selecting test cases as path in the FSA using various test selection criteria like all-transitions or all-transition-pairs.

The four steps in test case generation from VDM specification are:

- Extraction of definitions by collecting all parts (precondition, postcondition, invariants);
- Unfolding of all definitions (in case of recursive definitions the unfolding is limited to some predefined limited number);
- Transformation of the definition to DNF to get the disjoint sub-domains; and
- Further simplification of each sub-domain.

2.5 B machine Specification

B is a formal method for specifying, refining and implementing software. B is a formal method that allows us to produce proof obligations that demonstrate correctness of the design and the refinement. The main idea of B is to start with abstract model of the system under development and gradually add details by building a sequence of more concrete models. A B development process creates a number of proof obligations, which guarantee the correctness of B specifications. Proof obligations can then be proven by automatic or interactive prover. The goal of B is to obtain a proved model [27].

A B machine is conceptually not very different from an object (a single instance of a class) or a module. That is, it defines some private state plus several public operations that manipulate and access that state. The B abstract machine notation is a textual notation. The structure of the machine is a series of clauses; the common B clauses are summarized in table 2.2. The full B notation also allows parameterized machines and several kinds of import. We now briefly describe the purpose of each clause.

The MACHINE clause contains the name of the machine. This is similar to the name of a class in Java. The SETS clause is for defining data types that the machine will use. We give the name of each data type and if it is a finite enumerated type, we can list all its members.

Table 2.2 Structure of B machine.

MACHINE
name
SETS
basic_types
CONSTANTS
constant_names
PROPERTIES

```
        properties_of_constants
VARIABLES
        variable_names
INVARIANT
        properties_of_variables
INITIALISATION
        initial_state
DEFINITIONS
        macros
OPERATIONS
        operations
END
```

The CONSTANTS clause contains the names of any constants used in the machine. The PROPERTIES clause specifies the values of those constants. The VARIABLES clause specifies all the data variables of the machine, and The INVARIANT clause specifies the allowable range of each variable, plus any constraints *between* the variables. To be more precise, for each variable v in the VARIABLES clause, the INVARIANT must contain a conjunct of the form $v \in S$ or $v \subseteq S$. But it typically also specifies relationships between variables. The INITIALISATION clause is a set of parallel assignments, giving a value to each state variable.

The OPERATIONS clause defines all the operations of the machine. Some of these will be *query* operations, which return a value but do not change the state variables. Other operations will be *update* operations, which do change the state variables. The behavior of each operation is specified by a *precondition*, which says what input values are allowed, and a *substitution*, which is a set of parallel assignments. These assignments must update all the output variables of the operation and may also update some of the state variables.

To enhance expressiveness, there are several conditional constructs, like IF-THEN-ELSE and CASE statements, which allow us to specify conditional updates. However, no looping or sequential composition is allowed in B machines, and a given

variable cannot be updated twice in parallel. These restrictions mean that every execution of an operation updates each variable at most once. This makes the operations easier to update the state variables and for test case generation.

2.6 The B Toolkit

The B notation includes a large toolkit of data structures for use within machines, including sets, relations, functions, sequences, bags or multisets and trees. These data structures make it possible to represent complex information and the data structures of the application at an abstract level. The goal of using these high-level data structures is to write a clear, simple, and abstract behavior model. In this chapter we explore a brief introduction to the B notation for logic, sets, relations, and functions.

2.6.1 Logic

B has the Boolean operators that most programming languages have, such as *conjunction*, *disjunction*, and *negation*, plus equality and inequality on all types of data, and the usual integer arithmetic operators. It also provides logical *implication* and *equivalence* operators. The main logical operators are summarized in Table 2.3.

Table 2.3 Logical Operators in B

ASCII	Typeset	Name	True when.
P & Q	$P \wedge Q$	conjunction	P and Q are both true
P or Q	$P \vee Q$	disjunction	At least one of P, Q is true
not(P)	$\cong P$	negation	P is false
P => Q	$P \Rightarrow Q$	implication	$(\cong P) \vee Q$
P <=> Q	$P \Leftrightarrow Q$	equivalence	$(P \wedge Q) \vee (\cong P \wedge \cong Q)$
E = F	$E = F$	equality	*E* and *F* have the same value
E /= F	$E \neq F$	inequality	*E* and *F* have different values

When we have a large data structure, or one that varies in size, and we want to say something about all the elements, then we can use the *universal quantifier*. This is written as !x.(x:S => P), or $\forall x.(x \in S \Rightarrow P)$ when typeset, and is true when the predicate *P* is true for *every* $x \in S$. Similarly, if we want to search a data structure, or say that it is possible to find a solution to some problem, we can use the *existential quantifier*. This is written as #x.(x:S & P), or $\exists x.(x \in S \wedge P)$ when typeset, and is true when there is at least one member $x \in S$ that makes the predicate *P* true.

2.6.2 Sets

Sets are collections of elements of the same type, without repetition. They are not ordered. They are written using braces, { }. For example, the set of prime numbers less than 10 can be written {2, 3, 5, 7} or {2, 5, 7, 3} or even {2, 2, 3, 5, 7, 7}. These are all equivalent to {2, 3, 5, 7}.

We call the number of elements in a set its *cardinality*, written card(S). Elements of a set can be numbers, enumerated values, for example, the set of primary colors {red, yellow, blue}, or even sets. Set operators include the familiar operators such as union, written \/ in the ASCII form of B, or $\cup$ when typeset, intersection (/\ or $\cap$), and subtraction (-).

In addition to equality and inequality, there are several Boolean operators on sets to allow us to check their members. The *membership* operator is written E: S, or $E \in S$ when typeset, and is true when the value of *E* is one of the elements of *S*. The *non-membership* operator, written E /: S, or $E \notin S$ when typeset, is the logical negation of membership, so it is true when the value of *E* is *not* one of the members of *S*.

The *subset* operator is written S <: T, or $S \subseteq T$ when typeset, and is true when *every* member of *S* is also a member of *T*. That is, $S \subseteq T \Leftrightarrow \forall x.(x \in S \Rightarrow x \in T)$

Table 2.4 Set Operators in B

Operator	Meaning
{}	empty set
{E, F}	set enumeration
card(S)	cardinality

S * T	cartesian product
S \/ T	set union
S /\ T	set intersection
S - T	set difference
E: S	member of
E /: S	not member of
S <: T	subset of
S /<: T	not subset of
S <<: T	strict subset of
S /<<: T	not strict subset of

2.6.3 Relations

We need to define various kinds of relationships between two collections. For this, we use *relations*, which are sets of pairs. A *pair* is written c |-> d, or $c \mapsto d$ in typeset format. This pair associates the elements *c* and *d*. Note that *c* and *d* may have different types. For example, the pair $3 \mapsto red$ links the integer 3 with the color red. The first element of a pair is called its *antecedent*, and the second element is called its *image*. So in the pair $3 \mapsto red$, 3 is the antecedent and *red* is the image of the pair.

A set of pairs between two sets is a *relation*. The set of possible relations from a set A to a set B is written A <-> B, or $A \leftrightarrow B$ when typeset.

The *domain* of a relation *R*, written dom(*R*), is the set of all the antecedents of *R*. The *range* of a relation *R*, written ran(*R*), is the set of all the image elements of *R*.

The *domain restriction* operator, written S <| R, or $S \lhd R$ when typeset, selects from the relation *R* those pairs whose antecedent belongs to the set *S*. A variant of domain restriction, called *domain antirestriction* or *domain subtraction*, is useful when you want to select the pairs whose antecedents are *not* in *S*. This is written S <<| T, when typeset. The *range restriction* operator, written R |> S, or $R \rhd S$ when typeset, is used to select from the relation *R* those pairs whose image belongs to the set *S*. And of course, there is also a *range antirestriction* (or *range subtraction*) operator, R |>> S.

Table 2.5 Relational Operators in B

Operator	Meaning
S<->T	relation
E|->F	maplet
dom(r)	domain of relation
ran(r)	range of relation
id(S)	identity relation
S<|r	domain restriction
S<<|r	domain subtraction
r|>S	range restriction
r|>>S	range subtraction
r~	Inverse of relation
r[S]	relational image
r1<+ r2	right overriding

2.6.4 Functions

Functions are just a special case of relations. They are the many-to-one and "one-to-one" relations—that is, the relations in whom each element of the domain is mapped to at most one image element. Functions are very common in B machines because they provide a powerful way to express a relationship between two sets. They can be used to model arrays, maps, permutations, and many other functional relationships.

We use the two most common kinds of function, which are partial functions and total functions. The difference is that a *total function* maps every element of its input type to some value, whereas a *partial function* may leave some elements undefined. Since functions are relations, they are sets of pairs, so we can use all the set and relation operators to define and manipulate functions. For example, the *dom* operator tells us all the input values of the function, and the relational override <+ operator allows us to update the function.

Table 2.6 Functional Operators in B

Operator	Meaning
S+->T	partial function
S-->T	total function
%x.(x:T|E)	lambda function with input x and output E
f(E)	function application

2.7 Test case Generation from B Specification

In a research by Lionel Van Aertryck and Marc Benveniste et al. in [35], they proposed a method known as CASTING, a Computer Assisted Software Test Engineering method for test case generation from B formal specifications. They suggests a layered architecture consists of the "administrator layer", which provides configuration features, and the "user layer", which properly assists test designers in generating test suites from a given input text. Test generation is decomposed into two main stages in CASTING: the extraction of test case specifications, tcs, and the generation of test suites from tcs.

Extraction stage: The starting point of the process is the *test intention* that the tester may have in mind. An example of test intention for a B specification can be expressed as follow "Find test data for each case in the disjunctive normal form of the precondition of each operation" [32]. The extraction process applies a given strategy to the input formalism yielding a set of *tcs*. A *tcs* is a triple (E; C_i, C_o) where:

- E is the name of a functional entry point description of the unit under test (an operation in B, a function in C),
- C_i is a set of constraints on its input variables and state variables before execution,
- C_o is a set of constraints on its output variable and state variables after execution.

Constraints are expressed in first order predicate logic. Each triple (E, C_i, C_o) can be seen as a test case in symbolic form.

Generation stage: The generation process derives a collection of test suites from the set of *tcs*. A test suite is a sequence of the unit under test together with the predicates that must be satisfied by the resulting output. In order to build valid test suites, a symbolic graph is generated from the *tcs*. The paths in this graph are symbolic representations of the possible test suites. Then, a selection of paths is done and values are given to the variables to obtain the test suites. Due to the particular algorithm used to generate and select the test suites, this collection of test suites fulfils the following requirements:

- Each *tcs* must be covered by the derived test suites.
- Each test suite is an authorised (according to the input document) sequence of functional entry points.

This stage is carried out through a call to a constraint solver with an appropriate solving strategy in order to verify that tcs are covered by test cases.

In [36], Manoranjan Satpathy et al. adapt the abstraction refinement techniques of Clarke et al. [37] for systematic generation of model based test cases. They approximate a formal model by a more abstract finite state machine. From the finite model, they obtain probable test cases through model checking, and then a guided symbolic execution is performed over the given formal model to check if this is a real test case. In case of a failure of test case, the finite abstract model is refined and the cycle is repeated to check until the test cases are able to detect faults.

A combination of the CEGAR approach [37] and the partition refinement technique [38] will be used to obtain model based test cases for the B models. This approach replaces all operations in the given B model by operation instances; obviously it does not alter the behavior of the B machine. And the testing criterion is to obtain test cases along with expected outputs for each operation instance. The B model so generated will be treated as the concrete model in the CEGAR approach that they intend to use. So they will obtain a model which would be an abstraction of the concrete model. Then the CEGAR approach will be applied, and when there is a need to refine the abstract model, this will be done by partition refinement. The concrete counterexamples, so obtained, will be the test cases.

Algorithm for Customization of Abstract model:

```
customize abstract(A, OP(X)) {
//A: Abstract model; L0 is the lone state
create a new state STOP; pred(STOP) = true;
remove the self loop with label OP(X) from A;
add a new transition (L0, OP(X), STOP) to A;
add loops to STOP for each operation in original A;
}
```

Every node in the abstract state stands for a set of states in the concrete model, and all the concrete states mapping to an abstract state constitute one equivalence class. Each state in the abstract state is identified with a predicate. They use the following notations for describing model and its refinement. If S is a state (or a node) in the abstract model, pred(S) corresponds to the state predicate. For a B operation invocation OP(X), pre(OP(X)) denotes the operation precondition, and body(OP(X)) denotes the substitution corresponding to the operation call. The variable X sitting in place of the actual parameter of OP( ) can be instantiated; for multiple parameters, there are multiple variables. If P is a state predicate in the abstract model, then the weakest precondition due to the application of OP(X) is denoted by wp(P, body(OP(X))). An edge in the abstract model with the label (s, OP(X), t) would mean that the invocation of a call to OP(X) at source state s has led control to the target state t.

In research by Nikolai Kosmatov et al. in [39, 40], proposed BZ-TESTING-TOOLS (BZ-TT) for generating tests from formal specifications. They proposed model-based coverage criteria, based on formalizing boundary-value testing heuristics. This criteria form a hierarchy of data oriented coverage criteria, and can be applied to any formal notation that uses variables and values for representing system states.

The BZ-TT method consists of testing all the possible behaviors of the system when it is in a *boundary state* of its subdomains. This test generation process is fully supported by the BZ-TT tool-set. The behavior of the system on the points of a subdomain is defined by a predicate which is called an *effect predicate*. One effect predicate defines one behavior of the specification. An effect predicate contains two

parts: the conditions on input and state variables defining the subdomain (before part), and the predicates defining the system behavior on this subdomain (after part). The precondition of the operation is included into the first part of the effect predicate. In the BZ-TT approach, the first step consists in partitioning each operation of the formal model to compute these effect predicates [33, 34]. This partition analysis is similar to the DNF analysis. An effect predicate corresponds to a disjunct of the whole formula for each operation. The formula, representing the DNF of the operation, is computed by introducing some optimizations to control the combinatorial explosion underlying to the DNF partitioning. Basically, this partition analysis consists of unfolding predicates along branches, and introducing primed variables to denote the after values. Each computed before-after predicate constitutes one effect predicate.

An effect predicates PS_i has the form:

$$PS_i = J \wedge Before_i \wedge After_i$$

where J, $Before_i$ and $After_i$ denote the invariant properties, the before and after predicates of the effect i. The domain of the effect predicates PS_i is defined by $J \wedge Before_i$.

Let $x_1, x_2, \ldots x_n$ be the state and input variables and I be the set of all the elements $(x_1, x_2, \ldots x_n)$ satisfying the invariant J. If Θ is the Before part of an effect predicate, the set D defined as a subset of I by Θ is:

$$D = \{ (x_1, x_2, \ldots x_n) \in I \mid (x_1, x_2, \ldots x_n) \text{ satisfies } \Theta \}$$

will be called the *domain* of this effect predicate. So the boundary values of a predicate can be defined by its domain. The assumption in this approach is that all of the domains are bounded. For $D \subset Z^n$, it is equivalent to saying that the set D is finite. The criteria and results in the above approach make sense for closed continuous domains as well as for discrete domains.

Chapter 3: Problem Statement

Problem statement describes the gap in the existing work and problem formulation. The Gap in existing work shows, what the limitations are in the existing work and which technique they used. In problem formulation, we give appropriate solution to solve the existing problem and suggest the novel work.

In previous chapter we have seen various approaches for test case generation from various formal model based specifications like Z, VDM and B. We found that Z and VDM specification are not as efficient as B for test case generation. Reasons for using B formal specification in proposed work for test case generation as compared to Z and VDM specification are:

- Z is primarily a specification notation, whereas B is a complete method that covers the whole lifecycle, including specification, refinement, and code generation.
- For specification Z is not as flexible as B notation, since schemas cannot be used in many different ways, whereas the B specification constructs are more tailored to specifying machines and to supporting refinement.
- Z specification is not flexible for specifying states of a system. In Z, the schema calculus is typically used to incrementally build a state space schema, an initialisation schema, and several operation schemas, but there is no language construct which directly defines a state machine. In contrast, B has no schema notation, but provides a abstract machine notation for specifying state machine, with many ways of layering and combining machines.
- Z specifies operations in different ways and has no provision for specifying non-determinism. In Z an operation schema contains a single predicate which relates the input and output, whereas in B, an operation is specified by a precondition and postcondition or generalised substitution predicate which updates the states. The generalised substitution notation in B includes constructs for assignment, choice, parallelism and non- determinism.

- There is a constraint in Z specification, the invariants are to be included in every operation, whereas in B, the operations are specified independently from the invariants and there is a proof obligation to show that each operation preserves the invariant.
- The greater programmatic feel of the B notation is increased by the use of generalized substitutions, as opposed to VDM's relational conditions. It's more convenient to give greater algorithmic detail in the B machine, and is more useful for the development of algorithms.
- The use of invariants in operation definitions differs in both VDM and B. In B, post conditions (in the form of generalized substitutions) have to be written so as to ensure the maintenance of the invariant. In VDM the state invariant is effectively part of the state typing information, and as such is assumed to be maintained in addition to the postcondition.
- The read and write frames are given explicitly in a VDM operation, whereas in B the variable access and modification is implicit in the form of the generalized substitution.

3.1 Comparison of Test Generation from Z, VDM and B Specifications:

	Z	B	VDM
Method	Test Template Framework	B Testing Tools (BTT)	
1. Selecting Test Data	This framework manually defines a tree of possible tests for each operation of a system. The root of the tree is valid input space of the operation. Each tree node specifies a set of	The BTT uses constraint logic programming solver (CLPS-B) to analyze each atomic predicate P in the specification. Then the CLPS calculates a set of boundary values for each variable V. So for each	Extraction of definitions by collecting all parts such as precondition, postconditon, and invariants. By rewriting

	input values, called test template, and is written as a Z schema, and are test cases in form of formal notation.	state variable in the specification, we obtain a set of all its boundary values.	the pre & post conditions of a VDM into DNF.
2. Test Cases	In TTF, children's of a node are generated by applying a test strategy, such as partition analysis, which partitions a predicate into DNF, & tests each disjunct. Leaves of the tree specify the actual tests to be performed on operation.	After the predicate P has put into the CLPS-B solver, it calculates a set of Lim(P, V) for each variable V. So for each variable in the B specification, we obtain all its boundary values, $Lim(V) = \bigcup_p Lim(P,V)$	After Transformation of the VDM specification to DNF, we get the disjoint sub-domains, represents test cases.
3. Sequencing of Tests	Sequencing of Tests was not originally addressed by TTF. No algorithms or heuristics was defined for test sequencing. Later sequencing was done manually by the test template framework for Z specification.	To take the SUT from initial state to boundary state, for each state variable V, & each of its boundary values $L \in Lim(V)$, the B specification is executed symbolically using the CLPS-B solver, to find the sequence of operations that reaches a state where V= L is true.	No sequencing is performed in test case generation from VDM specification.

4. Generating Test Oracles	Oracles are generated by conjoining, each leaf with the original operation schema, which is then projected onto the output variables.	The expected output of each operation, Op is obtained from the CLPS-B solver during the above simulation of Init : P : Op.	By extracting a Finite automata (FA) from the specification; the FA can be used to drive test execution results.

3.2 Gaps in Existing Work

Test case Generation from Z Specification: In generating test cases from Z specification, the boundary states calculated by test template framework are specific to each operation. So it generates less number of test cases required to test requirements behavior. Also in a system with complex state space, there are several operations that need to be tested on the partitions of that state space, which results in duplication within the TTF trees for those operations, because the same state-partitioning subtree must be derived for each operation. TTF builds the entire state machine before testing commences and does not address the instantiation of leaves in subtree.

Test case Generation from VDM Specification: In generating test cases from VDM specification, only the test generation process was defined, but only partition analysis was automated. At the first step of finite state automata generation, there is possible state explosion and the non- discovery problem which makes it difficult to determine all the FSA states and transitions. At the second step, there is also the problem of finding the shortest sequences of operations which cover all the transitions of the FSA.

Test case Generation from B Specification: In test case generation from BZ-Testing Tools, the tool does not provide the complete coverage of the model. Also in computation of boundary goals in BZTT, we need a stronger reachability test before beginning the effect- predicate computation. The BZTT tool does not address the issue of reachability of each boundary goal in test case computation. Also the computation of effect-predicate using disjunctive normal form partitioning is very complex in BZTT.

3.3 Problem Formulation

Testing requirements is major issue in software testing for verification and validation of the system under test. By testing requirements we can reduce the costs to detect and remediate defects in later stages of development, also it reduces the time to deliver the software. The approaches we investigated for test case generation from B machine specification in chapter2, in order to test requirements, do not address the issue of coverage of the requirements, described in a B behavioral formal model. In proposed approach, for complete coverage of the requirements, in test case generation from B specification, we annotate all the requirements for the system under test in the Atelier-B Tool, which provide help in later stages to detect which requirements are yet to be tested and which are covered by generated test cases. Pro-B tool is used for model checking of B specifications to randomly animate our specifications.

Proposed approach addresses the reachability issues by producing the coverage matrix between the informal requirements and the B formal model specifications. In proposed approach, we extract the invariants and postconditions of our specification, parse them in a general form, and apply test selection criteria to obtain test cases for each of our test objective. Parsing of our specifications in a generalized form enables later to change test selection criteria, to cover different parts of the model. Proposed approach uses B specifications in order to remove syntax errors by type checking, automatic refinement, generating proof obligation, and automatic and interactive animation of our specifications.

Chapter 4: Testing Functional Requirements using B Model Specifications

4.1 Methodology

As we investigated in chapter 3, B specification are more appropriate for specifying abstract machines and for supporting refinements, so in this section we present an approach for testing functional requirements of the system based on the specification provided by B formal machine model. In proposed approach we generate test cases, for the purpose of verification of requirements. The approaches used for test case generation from B model specification, does not provide complete coverage of the requirements in the model. In proposed approach, we use All-requirements coverage criteria; this criterion requires that all requirements are covered in the test suite. For this, we record all the requirements inside the B formal behavioural model, as annotations on various parts of the model, so that the test generation process can ensure that all requirements have been tested.

We now describe all of the steps in our approach in test case generation for testing functional requirements, in order to provide complete coverage of the requirements. We also describe a case study, System Process Scheduler, in which we tests all requirements for the scheduler. Figure 4.1 shows all of the steps in our proposed methodology.

Informal requirements: The proposed system takes informally stated requirements as input for test generation. Informal requirements for the system under test are specified in the English textual notation. These requirements describe the detailed test objectives for the system to be tested. These requirements provide the scope of the system, and dictate which aspects of system are important to test and which can be omitted. We considered to test only the functional requirements of the system to be tested.

Requirements Annotation in B Model: In this step informal requirements are transformed into B formal model notations. Since B provides a rigorous structure constructs, which makes it possible to represent complex information and the data structures of the application at an abstract level. Atelier-B tool is used for describing informal English language requirements in a formal set theory notational form which

removes inconsistencies, ambiguousness and informality in requirements. The tool provides requirements to be annotated with requirement identifiers, which provide help in later stages to detect which requirements are tested or not.

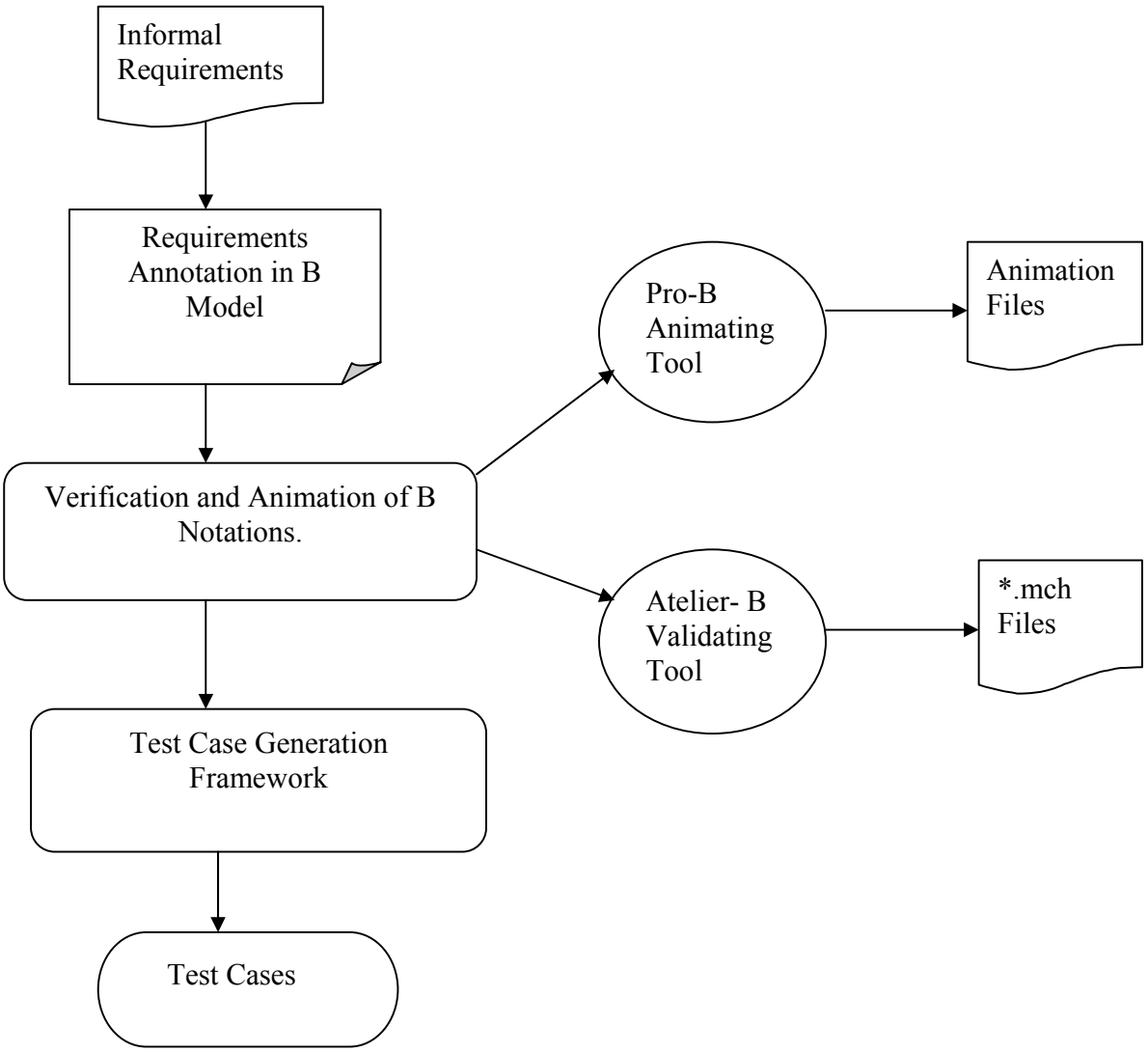


Fig.4.1 Test case generation in proposed methodology.

Verification and Animation of B Specification

In proposed work, Atelier-B tool is used for type checking, automatic refinement and generating proof obligations in order to remove syntax errors. We used ProTest-B tool, which provides automatic test environment for B specifications. ProB is an animator and model checker for the B specifications. It allows fully automatic

animation of B specifications, and can be used to systematically check a specification for errors. ProB is used to animate the B specification, either interactively or automatically. We used it to systematically model check a B specification for errors. ProB covers B notation, such as non-deterministic operations, ANY statements, operations with complex arguments, sets, sequences, relations, functions, abstractions, set comprehensions, constants and properties, for interactively animating and proving correctness of the components used.

Test Case Generation Framework

In proposed approach we extract the pre and post conditions of B specifications from the *.mch file in Atelier-B, then we parse the B constructs in a generalized form. Then according to our test selection criteria, we cover all paths of our model.

Test selection: Test selection criteria and the parsed syntax of B formal specifications are provided to the test generation framework to generate a set of *test cases*, for each desired test objective. We take decision coverage as our test selection criteria, as we want each Boolean decision in our model to be tested with a true and a false outcome.

Test Generation: Each generated test case with proposed approach for testing requirements is a sequence of operation calls. The test cases generated are the concatenation of four subsequences as in ISO [9646] standard: Preamble sets test up, getting the SUT into the correct state to run the test, Body is the test's main part, often a single call to the operation being tested, Observation**,** also called the *verification* or *identification* sequence, checks the test results and tries to ensure that the SUT is in the correct state, and Postamble takes the SUT back to initial state, in order to start the next test.

Coverage Matrix: In order to trace the coverage of all the requirements in our specification, we create coverage matrices between the informal requirements and the generated test cases.

4.2 Experimental Results:

Process Scheduler is used as an example for demonstrating test case generation for testing functional requirements of the Scheduler.

Informal Requirements for Process Scheduler

Informal requirements for the scheduler provide the detailed understanding of the working of the system, i.e. what the system does, how it performs its functions, what output it should produce under normal circumstances and what under exceptional cases. The informal requirements we consider here are the major requirements for the scheduler. These informal requirements specify which requirements are most critical to test, and which are less important in system development.

A process executing on a CPU is defined by its code, or program, and by the contents of its working space such as, data, stack, etc. When several processes can be executed at the same time, they are activated in turn, with each process following the life cycle shown in Figure 4.2. At any one time, the system may have some processes ready to be scheduled; some processes waiting for some external action before they become ready, for example a process waiting for an input-output request to complete; and a single running process.

The system must respect the following requirements. To make it easy to refer to these requirements later, for the purpose of coverage of requirements, we label each requirement with a requirement identifier such as [REQ1], [REQ2] and so on. The Atelier-B tool annotate requirements in form of /*@REQ: Requirement_id @*/.

- Each process is identified by a unique process identifier (ID) ---------- [REQ1].
  This requirement states that, all the processes running in the system must have its unique identity.
- The only time there is no running process is when there are none ready to be scheduled. ---------- [REQ2].
  This requirement state when there is no process in the ready state, then there will be no running process in the system.
- The initial state of the system is idle and empty of processes ---------- [REQ3].

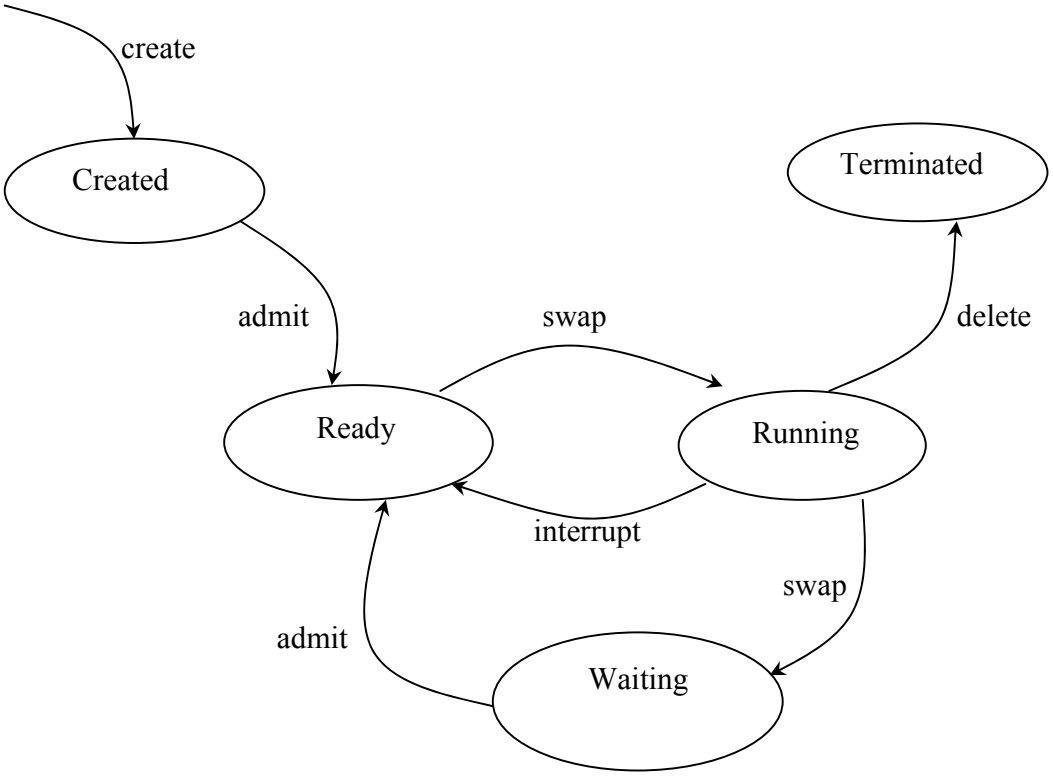


Fig 4.2 Process life cycle within Scheduler.

The scheduler has six operations to perform:

1. create(pro), create operation, which introduces a new process pro into the system, putting that process into a JobQueue and in created state. ---------- [REQ4].

   This operation may throw the following exceptions:

   • *MemoryFull* when all the possible processes have already been created -[REQ5].

   • *AlreadyExists* when the process pro already exists in the system ------ [REQ6].

2. deletepro(pro), delete operation, which delete/erases a running process pro from the system and move that process in terminated state. ---------- [REQ7].

   This operation may throw the following exceptions:

   • *UnknownProcess* if pro is not a known process ---------- [REQ8].

• *BusyProcess* if process pro is running or ready ---------- [REQ9].

3. admit(pro), a admit operation is called when a process has finished waiting and is now ready to execute, for example, when a disk read request has been satisfied or a user input event becomes available. It moves the waiting process into a ready state, when the process has finished its waiting work---------- [REQ10]. or directly into the ready state, when the process is newly created. -------[REQ11].

It may throw the following exceptions:

• *UnknownProcess* if process pro is not known to the scheduler ----------[REQ12].

• *BusyProcess* if process pro is in a running or ready state rather than a waiting state. ---------- [REQ13].

4. swap operation, swaps the currently running process out and replaces it by a ready process. ---------- [REQ14], or leaves the system idle if there are no ready processes ---------- [REQ15]. If swap is called when there is no running process, which means that there are also no ready processes, it does nothing. ---------- [REQ16].

5. interrupt(pro), interrupt operation, which moves a running process pro into a ready state, for example, suppose a process is executing and suddenly someone presses any control button then the currently executing process moves into a ready queue, waiting again to be assigned for execution. ---------- [REQ17].

It may throw the following exceptions:

• *UnknownProcess* if process pro is not known to the scheduler ----------[REQ18].

• *NoActiveProcess* if process pro is not in a running state. ---------- [REQ19].

6. A status(pro) operation can be called to determine the status of a known process pro. The status operation displays all of the created, admit, running, waiting and terminated processes in the system. ---------- [REQ20].

To design the process scheduler, we must decide on the scope of the model and which aspects are most important to test and which can be omitted. To do so, we first decide the test objectives for process scheduler, then the signatures of the operations in the model, the state variables of the model and their data types, and finally the pre and post conditions of each operation.

For the first step, our test objective is to test the functional correctness of the scheduler, leaving the real-time aspects of the system.

For the second step, the signatures of the operations in the model, we want to test all six of the scheduler operations. We also model exceptions by adding an output parameter to each operation that may throw an exception. This output parameter will be OK when the operation does a normal return, i.e. no exception or otherwise the name of the exception thrown by the function called, e.g., UnknownProcess, BusyProcess, AlreadyExits, NoActiveProcess etc.

The third step is to design the data structures of our specification. Each process is modeled by its identifier. We can now design the data parts of our specifications, by defining sets, constants, variables, and their data types. The most important structure is the scheduler variable, which is a partial function from ID to STATE. The scheduler variable is a partial function because only the known processes need to have a status, processes that have not yet been created with create operation will not appear in the scheduler data structure.

The last step is to write the body of each operation. That is, we design the functionality of each operation by transforming the informal requirements for the operation into precise preconditions and postconditions in B.

We now write in detail the specifications of all the operations create(pro), deletepro(pro), admit(pro), swap(pro), interrupt(pro), status(pro) of process scheduler containing sets, variables, invariants, and initialisation. Each operation modeled in Atelier-B creates its own .mch file, which contains the detailed structure for each operation. Fig.4.3 shows Atelier-B tool displaying sets, data and create operation for Process Scheduler.

File Process.mch: This file contains the descriptions of sets, variables, invariants, and initialization and create operation constructs in B specification.

We add an invariant to restrict the number of running processes to at most one; the scheduler |> {Running} operator extracts just the Running processes out of the scheduler function:

card(scheduler |> {Running}) <= 1

and another invariant to make sure that the CPU is kept busy, as REQ2 says if there are no running processes, then there must be no ready processes:

(scheduler |> {Running} = {} => scheduler |> {Ready} = {})

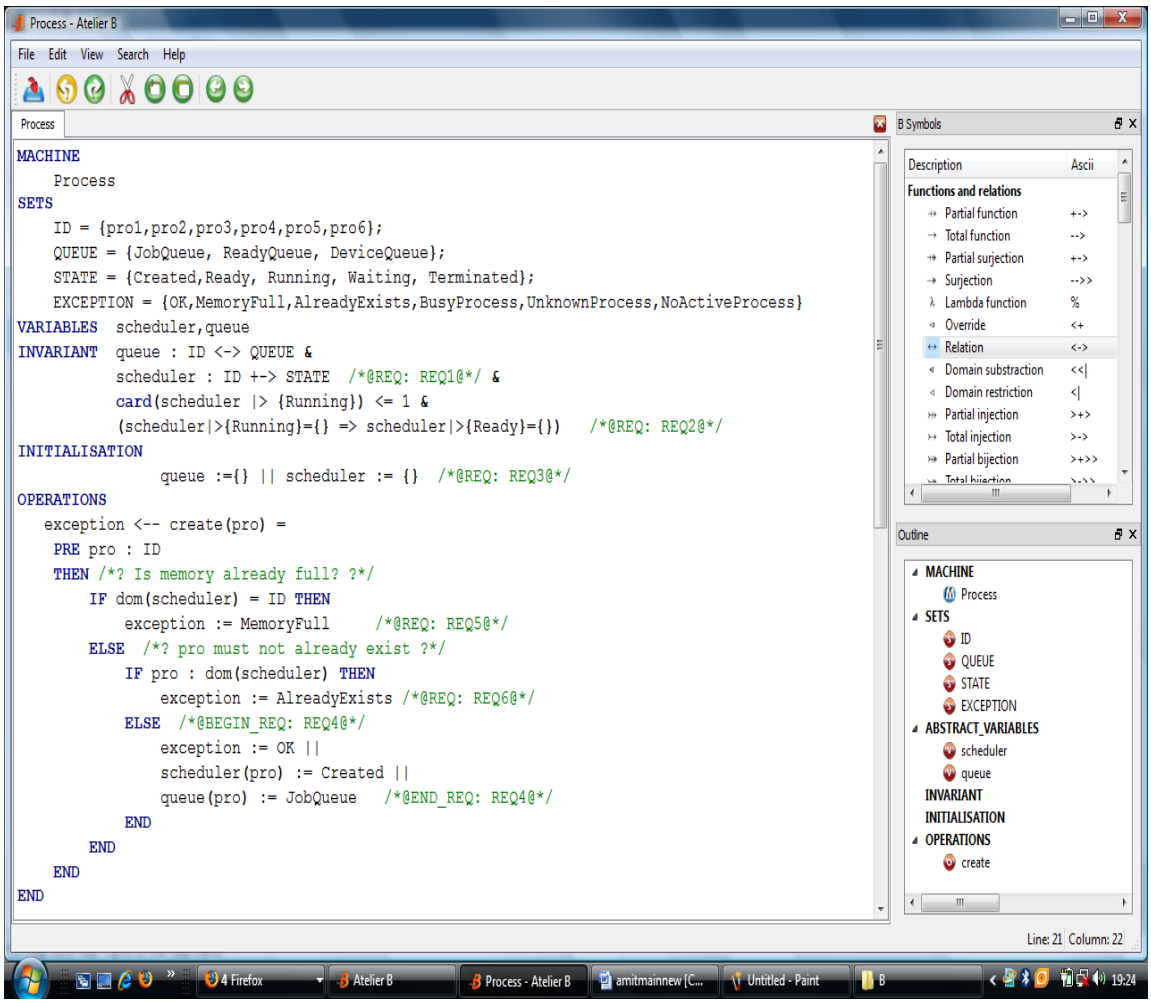


Figure 4.3 Process Machine Structure in Atelier-B.

```
MACHINE
   Process
SETS
   ID = {pro1, pro2, pro3, pro4, pro5, pro6};
   QUEUE = {JobQueue, ReadyQueue, DeviceQueue};
   STATE = {Created, Ready, Running, Waiting, Terminated};
   EXCEPTION = {OK,MemoryFull, AlreadyExists, BusyProcess, UnknownProcess,
                        NoActiveProcess};
VARIABLES
   scheduler, queue
INVARIANT
   queue : ID <-> QUEUE  &
   scheduler : ID +-> STATE &
   card(scheduler |> {Running}) <= 1 &
   (scheduler |> {Running}= {} => scheduler |>{Ready}={})
INITIALISATION
    queue := {} || scheduler := {}
OPERATIONS
     /*? This operation introduces a new process in created state. pro is the new
     process which is created and placed in jobqueue. Exception generated is: OK or
     MemoryFull or AlreadyExists ?*/

   exception <-- create(pro) =
   PRE    pro: ID
      THEN
      /*? If memory already full?  ?*/
      IF dom(scheduler) = ID THEN
         exception := MemoryFull
      ELSE
         /*? pro must not already exist ?*/
         IF pro : dom(scheduler) THEN
```

```
        exception := AlreadyExists
      ELSE
        exception := OK ||
        scheduler(pro) := Created ||
        queue(pro) := JobQueue
      END
    END
  END
END
```

File delete.mch: This file contain the description of the deletepro operation. The deletepro operation deletes a process (pro) from the system, when it has finished executing and changes the process state to terminated. The exception generated in deletepro operation is OK or BusyProcess or UnknownProcess. Figure 4.4 shows deletepro operation in B specification constructs.

```
OPERATIONS
  exception <-- deletepro(pro)=
  PRE   pro : ID
    THEN
     IF  pro /: dom(scheduler) THEN
        exception := UnknownProcess
     ELSE
        IF  scheduler(pro) /= Waiting THEN
            exception := BusyProcess
        ELSE
            exception := OK ||
            scheduler := scheduler - {pro |-> Waiting} ||
            scheduler(pro) := Terminated ||
            queue(pro) := queue(pro) <<| ReadyQueue
         END
     END
```

END

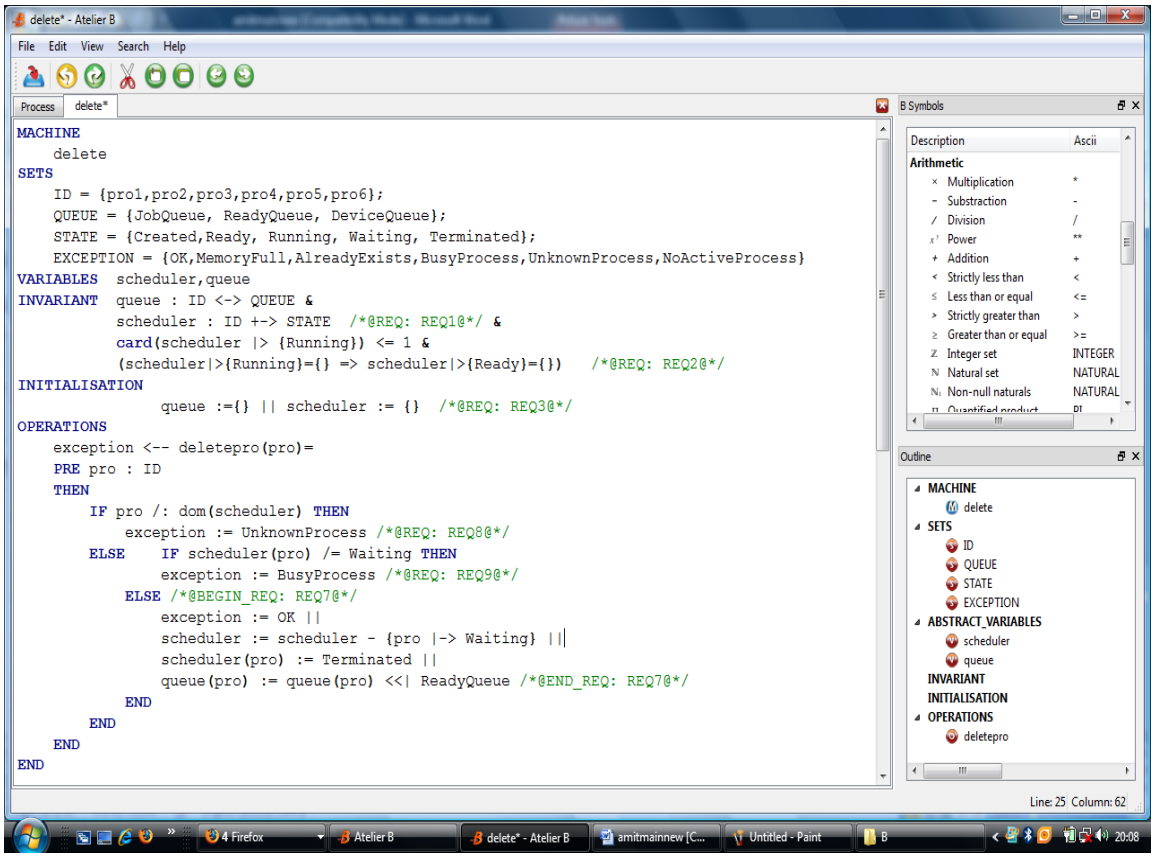


Figure 4.4 delete.mch file in Atelier-B.

File admitoperation.mch: This file contains the description of the admit operation. The admit operation moves a waiting process pro into a ready state, when the process has finished waiting. Also after creation of new process, pro moves to ready state. Figure 4.5 shows admit operation in B specification constructs.

OPERATIONS

exception <-- admit(pro) =

PRE pro : ID

THEN

IF pro /: dom(scheduler) THEN

exception := UnknownProcess

ELSE

```
        IF scheduler(pro) /= Waiting THEN
            exception := BusyProcess
        ELSE
            exception := OK ||
             IF  Waiting : ran(scheduler) THEN
                 scheduler(pro) := Ready ||
                 queue(pro) := queue(pro) <<| DeviceQueue
             ELSE
                 scheduler(pro) := Ready ||
                 queue(pro) := queue(pro) <<| JobQueue
         END
      END
   END
```

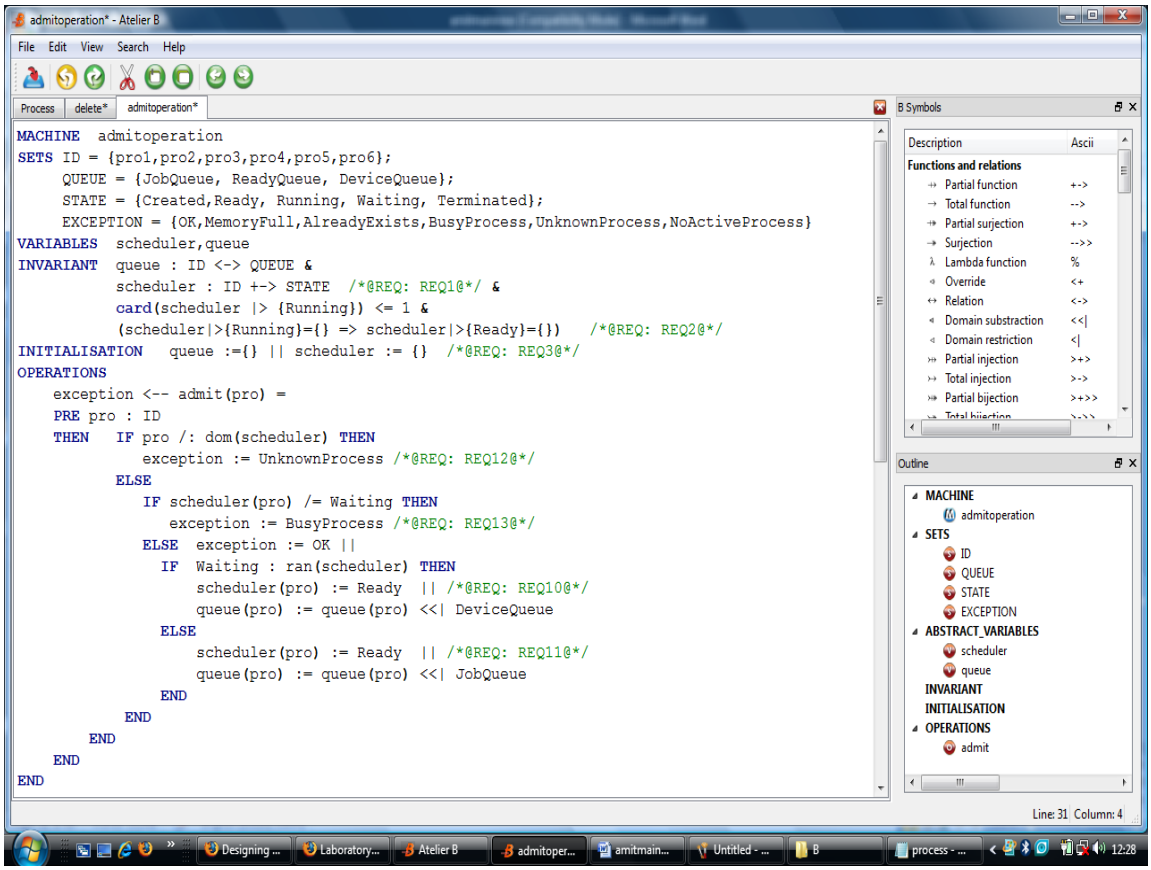


Figure 4.5 admitoperation.mch file in Atelier-B.

File swapoperation.mch: This file contains the description of the swap operation. The swap operation swaps currently Running process for a ready one. It leaves the system idle if there are no ready processes. When there is no running process, which means that there are also no ready processes, it does nothing. Figure 4.6 shows swap operation in B specification constructs.

```
OPERATIONS
  swap =
     IF Running : ran(scheduler)
        THEN  ANY current
            WHERE current: ID & (current|->Running): scheduler
          THEN
              IF Ready: ran(scheduler)
              THEN
                 ANY pro WHERE pro : ID & (pro |-> Ready) : scheduler
                 THEN
                   scheduler := scheduler <+ {pro |-> Running, current |-> Waiting} ||
                   scheduler(pro) := Running ||
                   queue(pro) := queue(pro) <<| ReadyQueue
                END
             ELSE
                   scheduler(current) := Waiting  ||
                   scheduler(pro) := Waiting ||
                   queue(pro) := queue(pro) -> DeviceQueue
           END
         END
      ELSE   Skip
   END
END
```

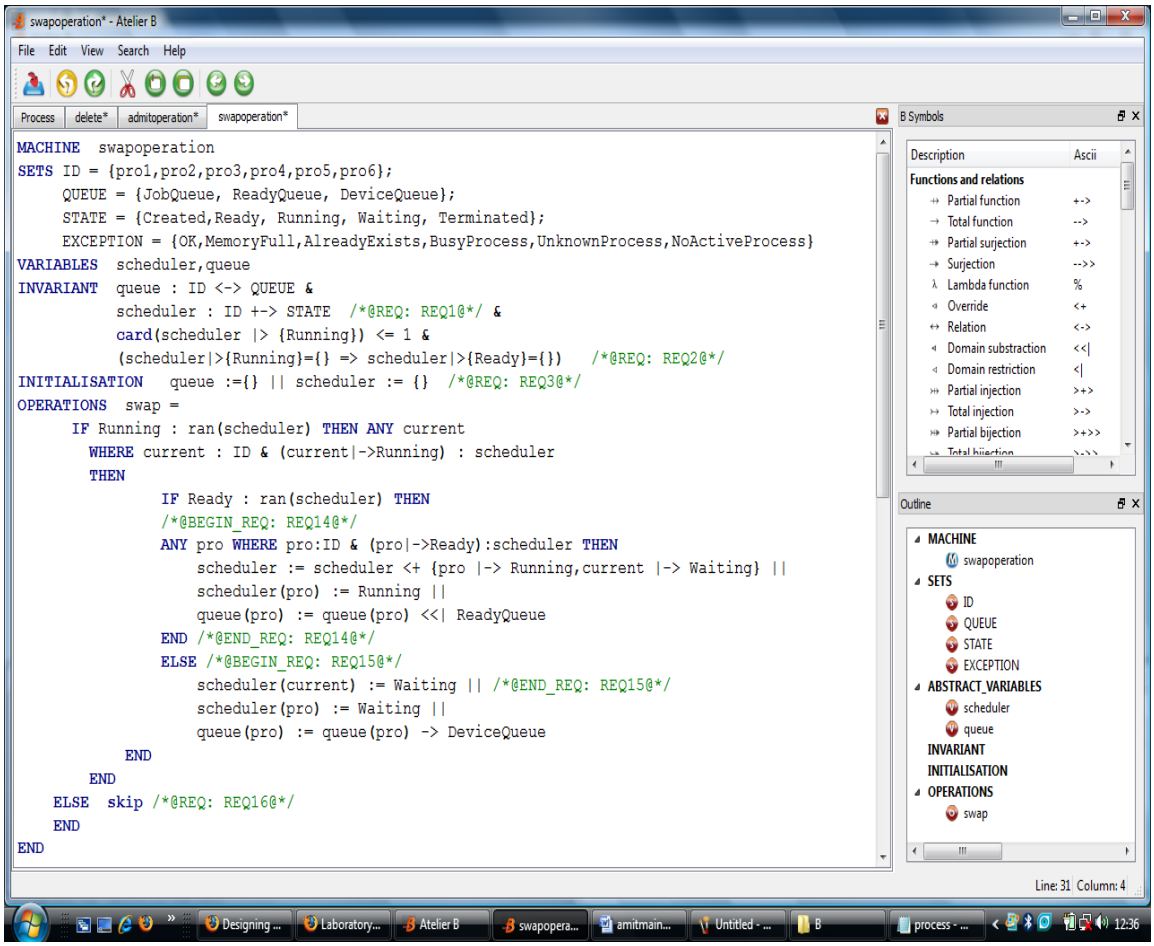


Figure 4.6 swapoperation.mch file in Atelier-B

File interruptoperation.mch: This file contains the description of the interrupt operation. The interrupt operation moves a running process pro into a ready state. Figure 4.7 shows interrupt operation in B specification constructs.

OPERATIONS

exception <-- interrupt(pro) =

PRE pro : ID

THEN

IF pro /: dom(scheduler) THEN

exception := UnknownProcess

ELSE IF scheduler(pro) /= Running THEN

exception := NoActiveProcess

ELSE exception := OK ||

```
            IF  Running : ran(scheduler) THEN
                scheduler(pro) := Ready  ||
                queue(pro) := queue(pro) -> ReadyQueue
        END
    END
END
```

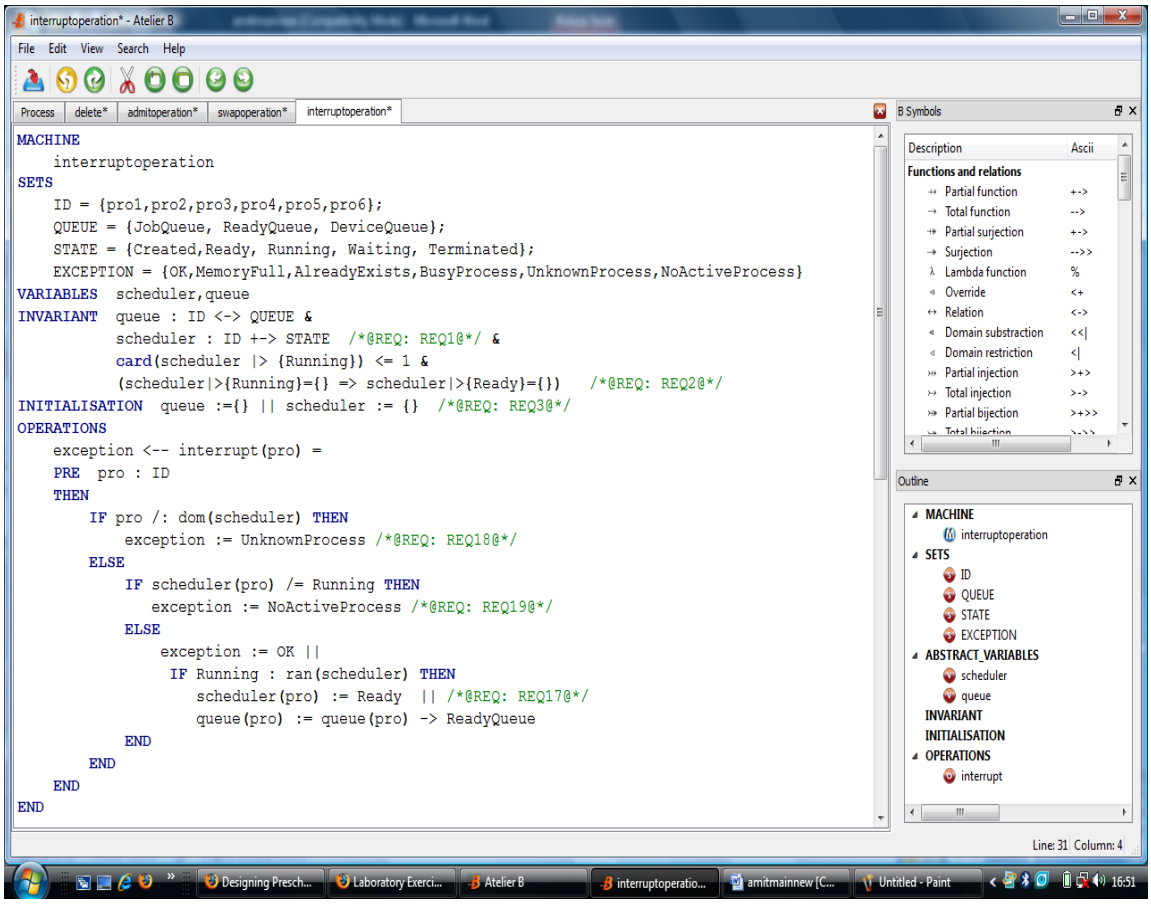


Figure 4.7 interruptoperation.mch file in Atelier-B

File statusoperation.mch: The status operation displays all of created, running, admit, waiting and terminated processes in the system. Figure 4.8 shows the B specification constructs of the status operation.

```
OPERATIONS
    crd, adt, rng, wtg, trd <-- status =
    PRE
```

```
        crd <: PID &
        adt <: PID &
        rng <: PID &
        wtg <: PID &
        trd  <: PID
    THEN
        crd := dom(scheduler |> {Created}) ||
        adt := dom(scheduler |> {Ready}) ||
        rng := dom(scheduler |> {Running}) ||
        wtg := dom(scheduler |> {Waiting}) ||
        trd := dom(scheduler |> {Terminated})
    END
END
```

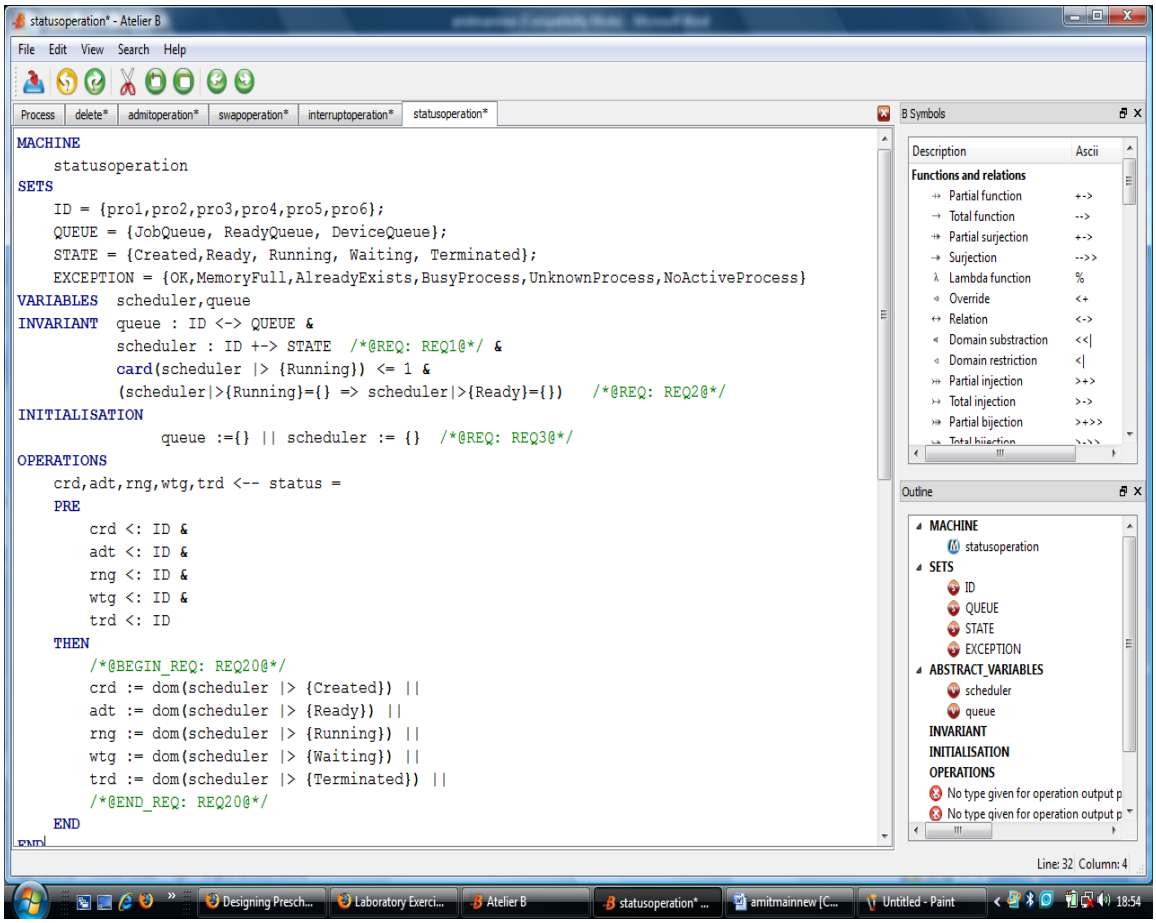


Figure 4.8 statusoperation.mch file in Atelier-B.

Verification and Animation of B Specifications

Atelier-B tool is used for type checking, automatic refinement and generating proof obligations of B specifications to remove syntax errors. We used the following functions of Atelier-B:

- Syntax Analysis and Type Checking: This function provides the syntax analysis and type checking of B components. The syntax checking ensures that the sources for the selected machines comply with the B language syntax.

  Type checking controls:

  - identifier conflicts, typing rules, and missing declarations,
  - language restrictions and visibility rules.

- Generating Proof Obligations: This function produces the proof obligations of a component. The component must be type checked. There is a separate proof obligation for initialization and a proof obligation for each of the operations. Proof Obligations for the Process Component are shown in figure 4.9. Generating proof obligations creates four files in the PDB:
  - The comp_name.po file contains the PO of the comp_name component.
  - The comp_name.opo file contains the obvious PO of the comp_name component.
  - The comp_name.pmi file contains the status of the proof obligations (proved/not proved) as well as the interactive demonstrations.
  - The comp_name.stc file contains a description of the component.

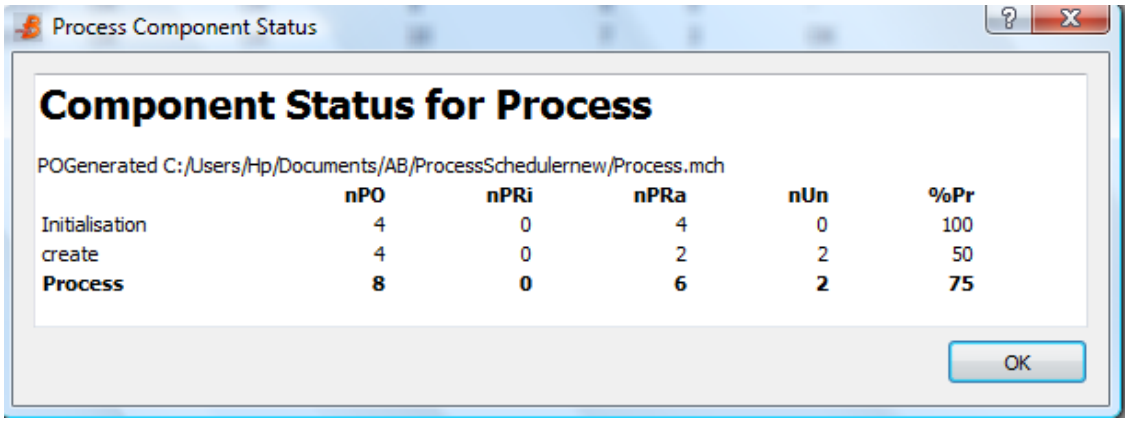


Figure 4.9 Proof Obligations for Process Component.

- Automatic refinement: This functionality of the Atelier B permits to generate refinements automatically, based on the use of Proof Obligations. Hierarchical structure of Atelier-B, showing the various components of the scheduler system are shown in Figure 4.10.

  The Atelier B prover is composed of an automatic prover and an interactive prover. The automatic prover enables to attempt to demonstrate automatically a set of proof obligations, by applying a given set of general mechanisms. The prover is parameterizable according to its force (Fast, force 0 to force 3). The higher the force is, the longer the proof takes. The interactive prover enables the user to assist the automatic prover in its demonstration task, by carrying out the proof. The prover is carried out using interactive commands. These commands are applied for a given proof obligation and are saved for this PO. Figure 4.11 shows the Prover component of Atelier-B for Automatic refinements.

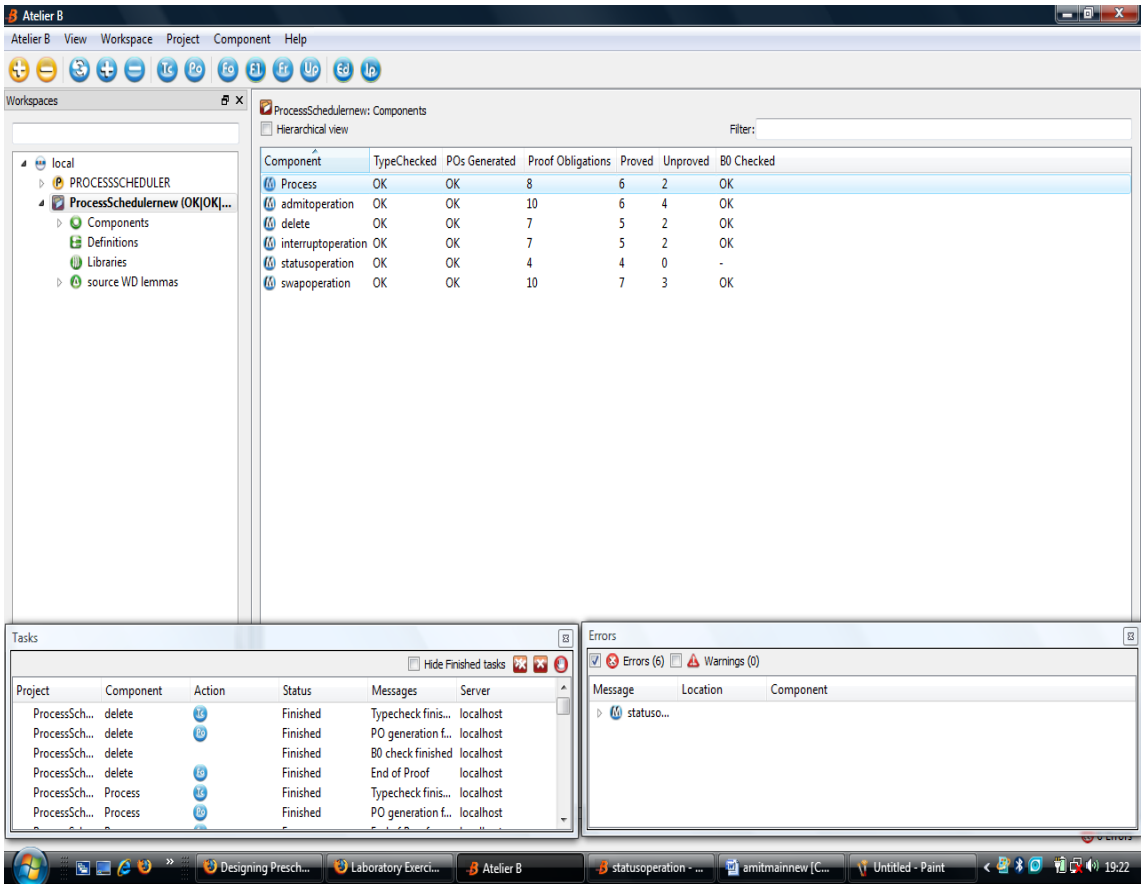


Figure 4.10 Hierarchical structure of Atelier-B

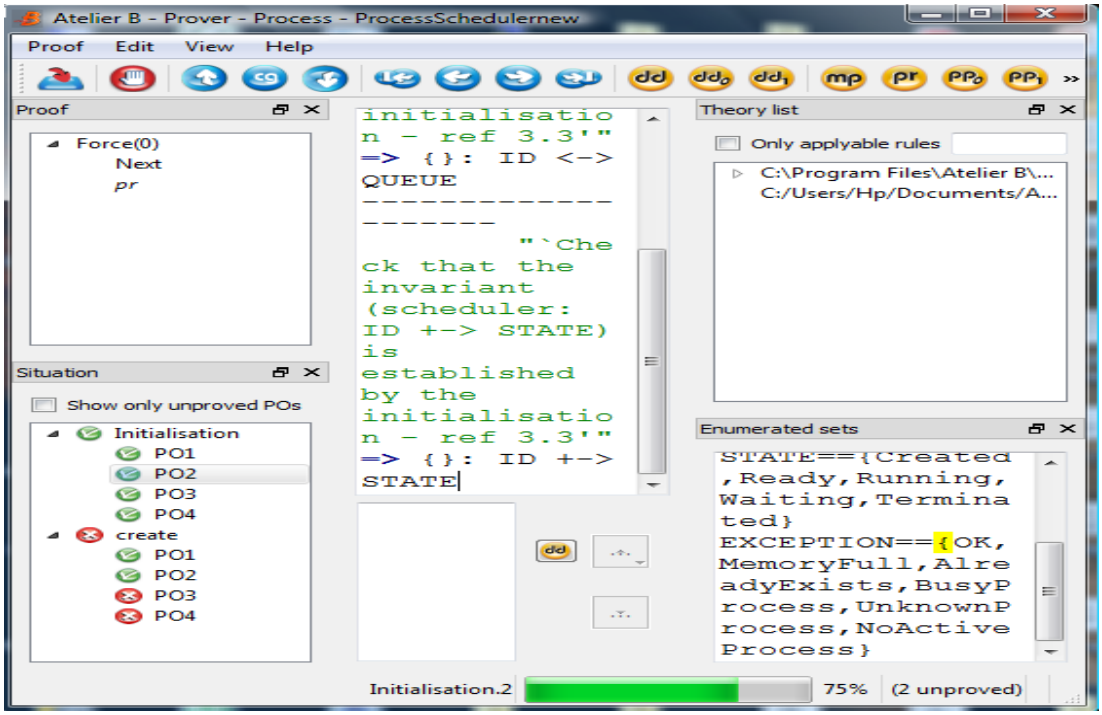


Figure 4.11 Prover for automatic refinement in Atelier-B for Process component.

ProB Tool: We have used Pro-B tool for animating and model checking of our B specification. ProB is used to systematically model check our B specification for errors. We used it to animate our B specification, either interactively or automatically and it also helps to correct and understand our specification.

Animation and Visualization in ProB:

We used animation facilities of ProB to gain confidence in our specifications. When the B specification is opened, the syntax checker analyses it and, if a syntax error is detected, it is then reported, and highlighted in the specification. Also if the B specification contains features that are not supported by ProB or constraints that are not satisfiable, an appropriate message is displayed. When these checks are passed, the B machine is loaded, and will display the operations that can be performed in the Enabled Operations pane. For our specification, ProB first *initialize constants*, and then computes the possible values and displays one *initialize constants* virtual operation for each possible group of constant values. Secondly it *initializes machine* and finally it performs the *backtrack* operation. We also *visualize* our specification

using ProB to analyze and understand the behavior of our B specification. ProB displays the state space as a graph whose nodes correspond to states and arcs correspond to operations. Figure 4.12 shows the random animation of our specifications in ProB.

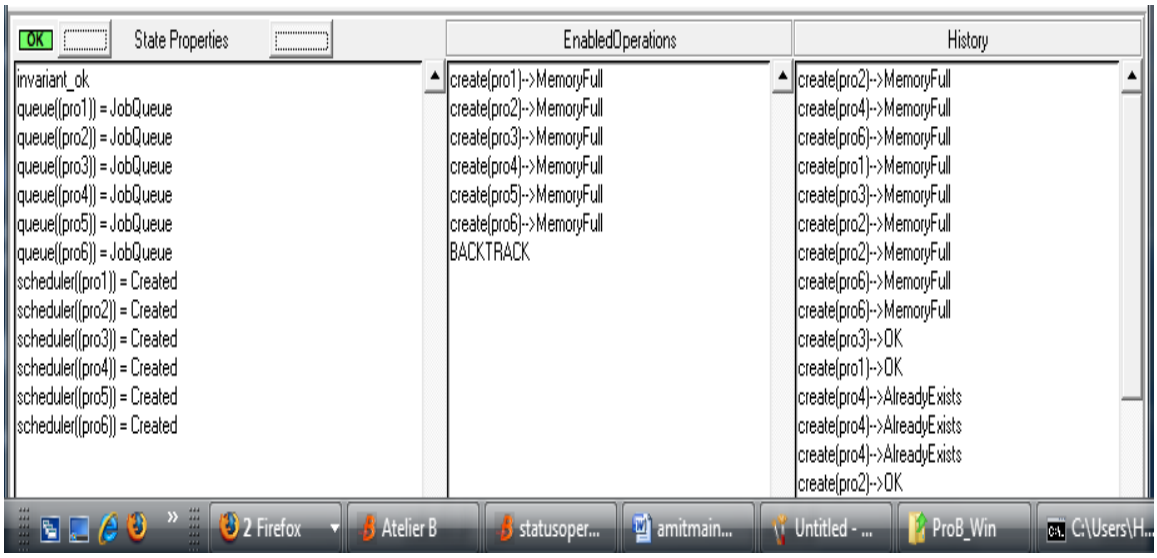


Figure 4.12 Random Animation of Process Component.

We used ProB for model checking of our specification to analyze any errors during the execution of the operations. Figure 4.13 shows debugging of the create operation, and analyzing whether constraints are satisfied or not. Coverage of operations in our model is shown in figure 4.14.

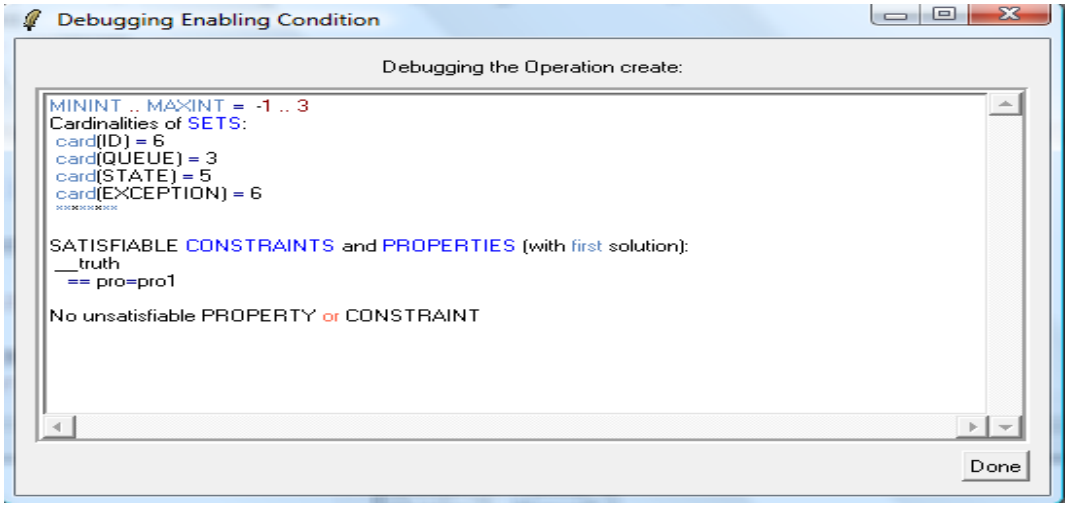


Figure 4.13 Debugging of create operation

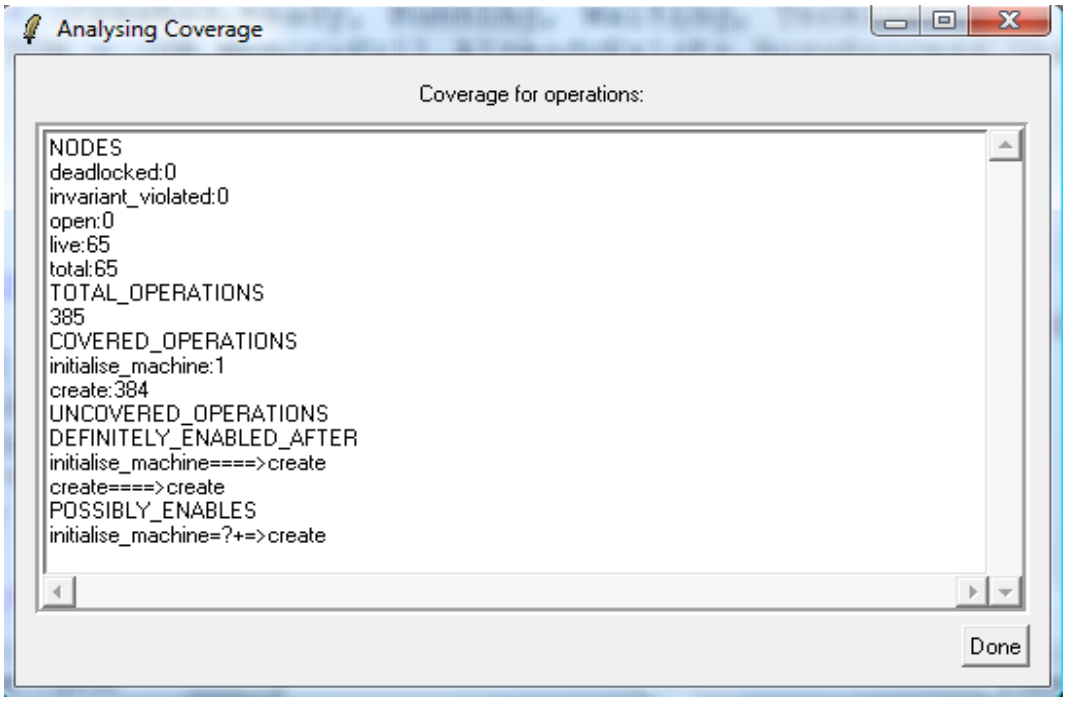


Figure 4.14 Coverage of operations in our model.

Test Cases

The test cases for testing informal requirements of the process scheduler are shown in table 4.1. We obtain 17 test cases for our scheduler system, and each test case is a sequence of operation calls. We use status as an observation operation, and after the invocation of each test case, the test case return the system to its initial state. The generated test cases includes the expected return values of each operation calls.

Table 4.1 Generated Test Cases for Process Scheduler.

Testcase_id	**Test Cases**
T1	create(pro1)=MemoryFull; status(pro1)=({},{},{},{},{}); deletepro(pro1)= ok.
T2	create(pro1)=ok; create(pro1)=AlreadyExists; status(pro1) = ({pro1},{},{},{},{}); deletepro(pro1)=ok.
T3	create(pro1)=ok; status=({pro1},{},{},{},{}); deletepro(pro1)=ok.
T4	deletepro(pro1)=UnknownProcess; status(pro1)=({},{},{},{},{}).
T5	create(pro1)=ok, admit(pro1)=ok; deletepro(pro1)=BusyProcess;

	status(pro1)=({},{pro1},{},{},{}); swap=ok, deletepro(pro1)=ok.
T6	create(pro1)=ok; deletepro(pro1)=OK; status(pro1) = ({},{}, {},{}, {pro1}).
T7	admit(pro1)=UnknownProcess; status(pro1)=({},{},{},{},{}).
T8	create(pro1)=ok, admit(pro1)=ok; admit(pro1)=BusyProcess; status(pro1)=({},{pro1},{},{},{}); deletepro(pro1)=ok.
T9	create(pro1)=ok, admit(pro1)=ok, status(pro1)=({},{},{},{pro1}, {}); swap=ok, deletepro(pro1)=ok.
T10	create(pro1)=ok; admit(pro1)=ok; status(pro1) =({}, {pro1}, {},{}, {}); swap=ok, deletepro(pro1) =ok.
T11	create(pro1)=ok, admit(pro1)=ok, swap=ok; status(pro1)=({},{}, {pro1}, {},{}); deletepro(pro1)=ok.
T12	create(pro1)=ok; admit(pro1)=ok; swap=ok; status(pro1)=({},{}, {},{pro1},{}); deletepro(pro1)=ok
T13	swap=ok; status(pro1)=({},{},{},{},{}).
T14	interrupt(pro1)=UnknownProcess; status(pro1)=({},{},{},{},{}).
T15	create(pro1)=ok, interrupt(pro1)=ok; interrupt(pro1)=NoActive Process; status(pro1)=({},{},{},{},{}); deletepro(pro1)=ok.
T16	create(pro1)=ok, interrupt(pro1)=ok, swap=ok; status(pro1)=({}, {pro1}, {}, {},{}); deletepro(pro1)=ok.
T17	status= ({},{},{},{},{}).

We present here the updating of the state and output variable for the test case T3.

/* initial state */ [scheduler = {}]

create(pro1) = OK; [scheduler = {pro1|-> Created}]

status = ({pro1},{},{},{},{}); [scheduler = {pro1|-> Created}]

deletepro(pro1) = OK [scheduler = {}]

Coverage Matrix between Requirements and Test Cases

As we tagged our B formal model specifications with requirements identifiers, we generate coverage matrices that show the relationships between the informal

requirements and the generated tests in Table 4.2. The matrix of requirements and test cases shows us how well each requirement is tested. The invariant (INVAR) and initialization (INIT) entry mean that every test is exercising those requirements. This matrix is useful for identifying requirements that have been tested adequately. This can help us to determine whether the failure is due to incorrect requirements or a fault in the system under test.

Table 4.2 Coverage Matrix from Requirements to Tests

Requirement_ID	**Generated Test Cases**
REQ1	INVAR
REQ2	INVAR
REQ3	INIT
REQ4	T3
REQ5	T1
REQ 6	T2
REQ7	T6
REQ8	T4
REQ9	T5
REQ10	T9
REQ11	T10
REQ12	T7
REQ13	T8
REQ14	T11
REQ15	T12
REQ16	T13
REQ17	T16
REQ18	T14
REQ19	T15
REQ20	T17

Chapter 5: Conclusions and Future Scope

The course of this work was to propose an approach for testing of requirement artifact, so that errors can't be propagated to the next phases of software development. Formal specification language was used for requirement specifications, which applies rigorous mathematical and set theory constructs in providing precise specification of the system. B formal specification language is used in proposed work to specify requirements for the purpose of testing of requirements. Comparison among the formal languages like Z, VDM and B enables the user to select a specification language, based on merits and demerits of above approaches.

5.1 Conclusions

In this thesis work, we proposed an approach that generates test cases for testing of functional requirements of the system. Following are the major conclusions drawn from the work undertaken are:

- Specifications annotated with requirement identifiers, help in later stages to detect which requirements are covered and which are yet to be tested.
- Coverage matrices between requirements and generated test cases help to resolve the reachability issues in system under test.
- B specifications are more appropriate for testing requirements of the system as compared to Z and VDM because B covers whole lifecycle including specification, refinement and generating proof obligations.
- Coverage of requirements in proposed approach is better than other approaches that we explored in chapter 2, in test case generation for testing requirements of the system.
- Atelier-B and ProB tool used for type checking, automatic refinement, generating proof obligations and animation of our specifications, removes syntax errors.

5.2 Future Scope of the work

- The current work can be extended further for large scale applications.
- This approach can be extended to cover the performance requirements of the system by adding more details in requirement annotations.
- Further work can be explored to test other software artifacts by using formal B specifications.

References:

[1]. B. Beizer. *"Software Testing Techniques",* Van Nostrand Reinhold, 2nd edition, 1990.

[2]. R.S.Pressman. *"Software Engineering: A Practitioner's Approach"*, 3rdEdition, McGraw Hill, New York (1992), p. 559.

[3]. Standard for Software Test Documentation (IEEE829) http://ww.ieee.org.

[4]. B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software Systems.* John Wiley & Sons, New York, USA, 1995.

[5]. G. Myers. *The Art of Software Testing*. Wiley- InterScience, 1979.

[6]. Perry, D. and Kaiser, G. *Adequate testing and object-oriented programming.* JOOP (Jan./Feb. 1990), 13-19.

[7]. The Open Group, http://tetworks.opengroup.org/.

[8]. Testinghttp://atlas.kennesaw.edu/dbraun/csis4650/A&D/testing_tutorial.html.

[9]. Black Box Testing http://www.cse.fau.edu/~maria/courses/cen/Black1.html

[10]. White Box Testing http://testtinggeek.com

[11]. Bender RBT Inc. *Requirements Based Testing Process Overview,* www.BenderRBT.com.

[12]. BOWEN, J. P., BOGDANOV, K., CLARK, J., HARMAN, M., HIERONS, R. M., AND KRAUSE, P. 2002. *FORTEST: Formal methods and testing*. In 26th IEEE Computer Software and Applications (COMPSAC 2002). 91.101.

[13]. HOARE, C. A. R. 1996. *How did software get so reliable without proof?* Lecture Notes in Computer Science 1051, 1-17.

[14]. KEMMERER, R.A. 1985. *Testing formal specifications to detect design errors.* IEEE Transactions on Software Engineering 11, 1, 32.43.

[15]. G. Bernot, M.-C. Gaudel and B. Marre. *Software Testing Based on Formal Specifications: A Theory and a Tool*, Software Engineering Journal, November 1991.

[16]. M.-C Gaudel. *Testing can be formal too*, TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, LNCS 915, Springer- Verlag, Aarhus, Denmark, May 1995.

[17]. BOWEN, J. P., BOGDANOV, K., CLARK, J., HARMAN, M., HIERONS, R. AND KRAUSE. Using Formal Specifications to Support Testing. In ACM Computing Surveys, Vol. 41, No. 2, Article 9.

[18]. LEE, D. AND YANNAKAKIS, M. 1994. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers 43,* 3, 306.320.

[19]. LEE, D. AND YANNAKAKIS, M. 1996. Principles and methods of testing finite-state machines. *Proceedings of the IEEE 84,* 8, 1089.1123.

[20]. HOARE, C. A. R. 1985. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science.

[21]. MOSSES, P. D. 2004. *CASL Reference Manual: The Complete Documentation the Common Algebraic Specification Language*. Lecture Notes in Computer Science, vol. 2960. Springer-Verlag.

[22]. ALUR, R., GROSU, R., HUR, Y., KUMAR, V., AND LEE, I. 2000. Modular specification of hybrid systems in CHARON. In *Hybrid Systems: Computation and Control, Third International Workshop, HSCC 2000*. 6.19.

[23]. GOGUEN, J. A. AND TARDO, J. J. 1979. An introduction to OBJ: a language for writing and testing formal algebraic specifications. In *The IEEE Conference on specifications of Reliable Software*. IEEE Computer Society Press, 170.189.

[24]. SPIVEY, J. M. 1988. Understanding Z: A Specification Language and its Formal Semantics. Cambridge University Press.

[25]. SPIVEY, J. M. 1992. The Z Notation: A Reference Manual, 2nd ed. Prentice Hall International Science in Computer Science.

[26]. JONES, C. B. 1991. Systematic Software Development using VDM, 2nd ed. Prentice Hall International Series in Computer Science.

[27]. ABRIAL, J.-R. 1996. The B-Book: Assigning Programs to Meanings. Cambridge University Press.

[28]. AMLA, N. AND AMMANN, P. 1992. Using Z specifications in category partition testing. In *COMPASS '92, 7th Annual Conference on Computer Assurance*.Gaithersburg, MD, USA, 15.18.

[29]. R M Hierons. Testing from a Z specification. *Software Testing, Verification and*

Reliability, 7:19–33, 1997.

[30]. S. Burton, "Automated Testing from Z Specifications," Departtment of Computer Science, University of York, 2000.

[31]. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. *FME'93:Industrial Strength Formal Methods. LCNS 670*, pages 268–284, April 1993.

[32]. B. Legeard and F. Peureux. Generation of functional test sequences from B formal specifications - Presentation and industrial case-study. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE'01)*, pages 377–381, San Diego, USA, November 2001. IEEE Computer Society Press.

[33]. F. Bouquet, B. Legeard, and F. Peureux. CLPS-B – A constraint solver for B. In *Proceedings of the ETAPS'02 International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 188–204, Grenoble, France, April 2002. Springer Verlag.

[34]. B. Legeard, F. Peureux, and M. Utting. Controlling Test Case Explosion in Test Generation from B Formal Models. *The Journal of Software Testing, Verification and Reliability*, 14(2):81 – 103, 2004.

[35]. Lionel Van Aertryck, Marc Benveniste, and Daniel Le M´etayer. CASTING: a Formally Based Software Test Generation Method. Published in IEEE International Conference on Formal Engineering Methods (ICFEM'97), first edition. 12-14 november 1997, Hiroshima, Japan.

[36]. Manoranjan Satpathy and S. Ramesh. Test Case Generation from Formal Models through Abstraction refinement and Model Checking. Published in AMOST'07, July 9-12, 2007, London, UK, ACM 978-1-59593-850-3/07/0007.

[37]. Clarke E., Grumberg O., Jha S., Lu Y., Veith H. (2003). Counterexample-Guided Abstraction Refinement for Symbolic Model Checking, Journal of ACM, Vol 50(5), pp.752-794.

[38]. He, J., Hoare, C.A.R., Sanders J.W. (1986). Data refinement refined, ESOP'86, LNCS 213, pp. 187-196.

[39]. Nikolai Kosmatov, Bruno Legeard, Fabien Peureux and Mark Utting. Boundary Coverage Criteria for Test Generation from Formal Models. Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04).

[40]. F. Ambert, F Bouquet, S. Chemin, S. Geunaud, B. Legeard, F. Peureux. and M. Utting. BZ-Testing Tools: A Tool set for test generation from B using constraint Logic programming. In Proc. Of Formal approaches to testing of software, Workshop of Concur'02, technical report, INRIA, pp 105-120.

List of Papers

1. Amit Gupta, Rajesh Bhatia, "Testing Functional Requirements using B Model Specifications", Communicated in International Conference on Software Engineering, CONSEG 09, Chennai, India (December 17-19, 2009).