A Novel Approach for Generating Test Suites for Component-Based Safety Critical Systems

Sudhan Kanade

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science at
Concordia University,
Montreal, Quebec, Canada

July 2004

Your file Votre référence
ISBN: 0-612-94702-5
Our file Notre référence
ISBN: 0-612-94702-5

ABSTRACT

A Novel Approach for Generating Test Suites for Component-Based Safety Critical Systems

Sudhan Kanade

Safety-critical system is a class of systems whose failure may cause severe consequences as such systems have absolute demands regarding correctness of functional as well as timing behavior of the system. There are existing informal and formal ways of testing safety critical systems, though formal-methods-based approaches provide rigorous and provable mathematical model for testing these systems. Nowadays component-based systems are preferred over monolithic systems because they help us to express complex information in more clear and unambiguous manner. But with the increase in size of the system, the need for proving the correctness of the system tends to be quite complex, thus requiring some mechanism to show the conformance of the system towards its requirements. Thus working towards these requirements, we have proposed a methodology for generating test suites for component-based safety critical system. Our proposed CAGILY framework introduces a formal framework for identifying a set of test cases from a well-specified system to validate critical functionalities of the system. This framework incorporates the concept of component identification and specification, and defines contracts/morphisms by adapting the theories of constraint cross-product in category theory to generate sets of comprehensive test suites. The formalization of the system is provided by specifying the system composition using the specification and verification tool called Specware. Further, we have developed and implemented Sampurna tool, which generates a set of test cases depending on the constraints imposed on the system. We illustrate the effectiveness of our proposed approach by applying it to a case study of mine pump problem. The important feature of our approach is that, we have been able to decompose the system depending upon their functional requirements and then testing the system for its critical functionality.

ACKNOWLEDGEMENTS

It has been a great privilege for me to work with Dr. Purnendu Sinha, an exceptional researcher and teacher, who introduced me to real-time systems, fault tolerant systems, test case generation and component-based design and allowed me to pursue research in Software Engineering domain, a subject close to my heart. He has been extraordinarily patient and supportive, have been always available for discussion and responding speedily to research reports. I would like to take this oppurtunity to thank him for his contiuned encouragement and guidance throughout the course of my research.

My sincere thanks goes to Professor Rachida Dssouli, with her invaluable insights on the topic and timely inputs. She was always there to help out at every stage and provide constructive comments on the research work.

I would also like to thank my fellow graduate student, Mr. Nikhil Varma, for his discussions concerning my research, that helped focus my efforts towards the completion of this thesis.

I extend my whole-hearted gratitude to my family for their encouragement and support, without which it would have been impossible to finish this work. Finally, I would like to thank my colleagues in the research group and my friends for their valuable discussions, support and advice.

Sudhan Kanade, July 2004

Contents

1 Introduction **1**

1.1 Component Based System Design 1

1.2 Software Testing 2

1.3 Need for Functional Testing 3

1.4 Importance of Test Cases 3

1.5 Role of Formalism in Testing 5

1.6 Different Approaches in Test Case Generation 6

1.6.1 State Machine Approach 7

1.6.2 Model Based Approach 9

1.6.3 Category-Partition Testing 9

1.6.4 Shortcomings of Above Discussed Approaches 11

1.6.5 Generation of Test Cases Using Logical Specification 12

1.7 Background and Motivation 14

1.8 Our Contributions 16

1.8.1 Sampurna Tool 17

1.9 Outline of the Thesis 18

2 Background - Category Theoretic Approach for Composition **19**

2.1 Category Theory 19

2.2 Categories in Category theory 20

2.3 Modularization in Category Theory 21

2.3.1 Signature 22

2.3.2 Specification 22

2.3.3 Specification Morphism 22

2.3.4 Pushout Operation 23

2.3.5 Module Interface 24

2.3.6 Parameter Interface (PAR) 25

2.3.7 Import Interface (IMP) 25

2.3.8 Export Interface (EXP) 25

2.3.9 Body Interface (BOD) 26

2.4 Specware 26

3 CAGILY - An Approach for Generating Test Suites 28

3.1 CAGILY Framework 28

3.1.1 Component Identification 29

3.1.2 Component Specification 30

3.1.2.1 Composition of Module Specifications 30

3.1.2.2 Composition with Constraints 31

3.1.3 Contract Definition 32

3.1.4 Mapping Contracts to Morphism 32

3.1.5 Test Case Generation Procedure 32

3.1.5.1 Cross Product of Categories 32

3.1.5.2 Constrained Based Cross Product 33

3.1.5.3 Sampurna: Test Suite Generation Tool 33

3.2 Mine Pump Problem - An Overview 34

3.3 CAGILY - A Detailed Illustration 36

3.3.1 Component Identification 36

3.3.2 Component Specification 37

3.3.3 Contract Definition 41

3.3.4 Mapping Contracts to Morphism 44

3.3.5 Test Case Generation Procedure 44

3.3.6 Illustration of Sampurna Tool 46

3.3.7 Effectiveness of CAGILY Framework 47

3.4 Specification and Verification using Specware 47

4 Implementation of Sampurna Tool 59

4.1 Database Tool for Generating Test Suites 61

4.2 Tables for Component Modules 62

4.3 Queries for Generating Test Suites 64

4.3.1 GetAllVariables 65

4.3.2 ConstrainedVariables 66

4.3.3 UnconstrainedVariables 66

4.3.4 Sensors And Controller Module 66

4.3.4.1 AirFlowSensorReqType 67

4.3.4.2 CarbondioxideReqType 67

4.3.4.3 ManualReqType 67

4.3.4.4 MethaneSensorReqType 68

4.3.4.5 PumpControllerReqType 68

4.3.4.6 WaterSensorReqType 68

4.4 Action Queries 69

4.4.1 Sampurna_Step1 69

4.4.2 Sampurna_Step2 70

4.4.3 Sampurna_Step3_1 70

4.4.4 Sampurna_Step3_2 71

4.4.5 Sampurna_Step3_3 72

4.5 Applying Resulting Test Suites - An Example 74

4.6 Effectiveness of Implementated Sampurna Tool 75

5 Conclusion **77**

5.1 Experience 78

5.2 Contributions 79

A Appendix **80**

A.1 Illustration of Sampurna Tool 80

B Appendix **89**

C Appendix **95**

List of Figures

2.1 Specification 23

2.2 Module 24

3.1 Hierarchical System Decomposition 29

3.2 PushOut Operation 30

3.3 Composition of Two Modules 31

3.4 Composition of two modules with constraints 31

3.5 Mine Pump Controller 34

3.6 Hierarchical Decomposition of Mine-Pump System 36

4.1 Sampurna Tool - Tables 64

4.2 Sampurna Tool - Queries 65

List of Tables

4.2 WaterSensorModule 62

4.4 ManualModule 63

4.6 CarbondioxideModule 63

4.8 AirFlowSensorModule 63

4.10 PumpControllerModule 63

4.11 TESTCASEGENERATION 63

4.12 RESULTANT_TESTCASE 63

4.13 TESTCASE 73

4.14 RESULTANT_TESTSUITE 74

B.1 TestCaseGeneration - 1 89

B.2 TestCaseGeneration - 2 90

B.3 TestCaseGeneration - 3 91

B.4 TestCaseGeneration - 4 92

B.5 TestCaseGeneration - 5 93

B.6 TestCaseGeneration - 6 94

C.1 ResultantTestCase - 1 96

C.2 ResultantTestCase - 2 97

C.3 ResultantTestCase - 3 98

Chapter 1

Introduction

In this thesis, we illustrate the framework to generate functional test suites for component-based system, which utilizes category theoretic approach for system composition. We begin with addressing the importance of testing in component based system. Further we discuss on a need for functional testing and role of formalism in testing. We then explore different approaches like the state machine approach and model based approach used for generation of test cases. Finally, we provide short summary of our CAGILY framework for functional test case generation, followed by the contribution of the thesis.

1.1 Component Based System Design

Component based software engineering is a promising and effective approach, managing the ever growing complexity of the software system providing a reliable and timely services. The overall idea is to partition a system into parts (components) that are language and machine independent, and can be connected via a network. Offering flexible migration to legacy systems and enabling reuse of code, component technologies are viewed in industry as a way of speeding up development and organizing systems with increasing size. But with increase in size of the system, the need for proving the correctness of the system tends to be more complex. However, the increasing complexity and size of such systems also increases the need for some specification and validation process to prove the correctness of the system and to show the conformance of the system towards its requirements.

The system decomposition helps us to express complex information in ways that are easily understood which in turn provides clear and unambiguous definitions of the behavior of the system. It affects the test case generation process by reducing time required to validate a batch of test cases, provides scalability and better test coverage.

1.2 Software Testing

Testing is the process of executing a program or system with the intent of finding errors. It is an indispensable step in the process of software development. Although testing can only show the presence of errors, not the absence of them, it is still a very effective way of gaining software reliability and supporting software quality assurance. However, doing testing is time consuming and expensive. It is widely believed that more than fifty percent of the time and cost of software development is spent in testing. In the software engineering area, many research projects are aimed at finding out more cost effective ways of doing testing.

Testing can be categorized in many different ways. If categorized by testing levels, there are unit testing, integration testing and system testing and if categorized by code accessibility, there are white box/structural testing, and black box/functional testing. Various aspects of software need to be checked with other types of testing, such as stress testing, usability testing, and performance testing and so on. For example, regression testing is selective retesting of a software system that has been modified to ensure that any bugs have been fixed and that no other previously working functions have failed as a result of the reparations and that newly added features have not created problems with previous versions of the software.

Besides these general categories, special types of software may need to be tested in more special ways. For instance, there are different approaches for object oriented software and real time reactive system testing. Among all categories of tests, one of the most important categories is black box testing or functional testing.

1.3 Need for Functional Testing

Functional testing is based on some form of specifications of the intended behavior of the software to be tested. It is implementation independent, as we do not need to know the internal design, structure or code of the software. Functional testing is important because after all, its the software's functions that are delivered to and used by end users, and the software is expected to act exactly as how it is specified in the original requirements. The main objective of functional testing is to ensure that every requirement is actually fulfilled. Other types of test such as white box or structural test, performance test, or stress test are necessary and effective too, but without black-box testing based on specifications, it is still possible that the software product is missing functions originally specified.

1.4 Importance of Test Cases

The requirements of the software to be developed may be expressed informally in a natural language, or rigorously written in a formal language. The requirement documentation then becomes the basis of the software design as well as the functional black box testing of the system under test.

There are three main steps in testing, namely defining test cases, doing the tests and evaluating test results [Edwa96]. Since defining test cases is the first step, it is in fact the basis of a successful and good test process. With the intent of test in mind, we believe that the more errors found by executing a set of test cases, the better those test cases are.

One can define test case as:

- A set of test input, execution condition, and expected results description for a particular objective.
- The smallest entity that is always executed as a unit, from beginning to end.

For different purposes of test, test cases can be organized as one or more test suites where a test suite is a group of tests with a common purpose. Traditionally, for informal specification based functional testing, the requirements are analyzed and a test suite is defined manually by testers. There are many methods used to select test cases. Commonly used approaches include equivalence partitioning, boundary value analysis and error guessing. Lesser-used methods include cause effect graphing, syntax testing, state transition testing and graph matrix [Edwa96]. Obviously, the manual inconsistency and incompleteness in informal specifications can cause problems in the test case generation and selection procedure. For instance, as the construction of test cases is subjective, each test case designer may choose different sets of test cases, and it is hard for one to judge what kind of test case is more effective and complete. Moreover, there is a lack of criteria for the completeness of a test suite. We know that it is impossible to do exhaustive testing, because from the technical point of view, we never know whether all errors in system under test are located, and from the managerial point of view, there is always a time limit for testing. However we do need some sort of coverage measurement criteria that can help us gain more confidence in the quality of system under test. Although many code and structural coverage measurement tools are in use today, for black-box testing, it is more meaningful to have some functional coverage measurement criteria to ensure that no function is missing in an implementation. Now the question is how? Formal specification based black-box testing might be the answer.

1.5 Role of Formalism in Testing

Today software is penetrating into more and more areas of our life, and many systems are safety-critical. They command higher quality and reliability requirements to prevent human-life and property loss. For the quality assurance of safety-critical systems, functional testing plays a more crucial role in gaining confidence in the systems functional behavior correctness. Researchers have been working on the application of formal methods in the software development life cycle to help reduce errors left in software products.

Ideally, one would like safety-critical systems to be proved correct by formal methods. But as current techniques of proving the correctness of entire systems using formal methods are immature, costly, and rarely used in practice, we still need to depend on testing for the validation of software systems. Fortunately, besides the proof of program correctness, formal methods benefit many other activities in the whole system development life cycle. A specification written in a formal language eliminates ambiguity and inconsistency. At the early stage of software development, analysis and design of systems based on formal specifications reduce the likelihood of errors in the products of those periods. And at later stages, formal specifications can be used to improve the efficiency of the testing process. Formal specifications can also be used in tasks such as deriving test oracles as well as deriving test cases in more logical and rigorous ways. They also allow measuring functional test coverage and quality of testing. More importantly, they make automation of test oracle derivation and test suite generation possible, thus can improve the efficiency and effectiveness of the whole testing process.

Research work has been done for methods and tools of deriving test oracles and test suites from various forms of formal specifications. There are model-based, logic-based or other forms of formal languages that all have mathematical semantics and syntax. Some formal languages are executable, making the validation of specifications and simulation of the system under test (SUT) easier. It takes effort to write formal specifications with most of the existing formal languages. In some cases, particular forms of formal specifications are required just for the purpose of performing tasks such as automatic test oracle derivation, and thus a lot of extra effort of writing such complex specifications is needed.

1.6 Different Approaches in Test Case Generation

Testing is often made more difficult by decisions made during requirements specification and software design. A method of assessing the testability of requirements specifications and software architectures would therefore help to further reduce the cost of testing. The high cost of Verification & Validation (V&V) is also compounded by the fact that it is often performed on many different iterations of the software. Iterations can be the result of faults in the software or faults in the requirements specification being detected late in the development lifecycle. Clearly, a better understanding of the requirements earlier in the project and a "right first time" approach to coding would dramatically decrease the recurring costs of V&V. This is obviously an ideal goal and would be difficult to foresee in practice. The effective test case generation would reduce V&V costs allowing more effort to be targeted at the requirements specification phase and at designing efficacious testing criteria.

There are obvious practical difficulties associated with testing software or hardware systems such as having poorly expressed requirements, informal design techniques and nothing executable available until the coding stage. On top of these are the various psychological and managerial problems.

Historically, there have emerged different classifications of testing techniques. Some approaches make a distinction between static and dynamic testing techniques. Static techniques are those that examine software without executing it and encompass activities such as inspection, symbolic execution and verification. Dynamic techniques are those that examine the system with a view to generating test data for execution. There is also one more distinction between these testing techniques. The test cases that are derived without reference to the implementation of the system (i.e., they are created with reference to the specification, or some other description of what the system should do) are termed black-box techniques. That is, the systems are treated as a black box and its functionality is determined by supplying it with different combinations of inputs. In contrast to this, test cases that are derived by examining the implementation of the system are termed "white-box" (programming style, control method, source language, database design, etc.). Other terms have been introduced over the years and now black-box techniques are sometimes called "functional" or "specification-based" and white-box techniques may be referred to as "structural" or "code-based" or even "glass-box".

Model-based specifications facilitate proof-based analysis and can also provide an explicit definition of both the control and data parts of the specification for testing purposes. Model-based specifications are based on predicate logic and set theory. The functionality of the system is expressed in terms of operations and functions that transform the state-space of the system. These operations and functions are expressed in terms of a standard collection of data types and type constructors. The state-space is restricted to valid states using state invariant predicates. Operations are expressed in terms of a pre-condition over the state and inputs that must hold before the operation can occur and a post-condition that restricts the values of the state and outputs following the operation.

Basically two types of testing techniques are present in specification based testing, one is with using State Machine approach, i.e., techniques for generating tests from FSMs and EFSMs, and secondly the techniques for generating tests from model-based specifications.

1.6.1 State Machine Approach

Finite State Machines (FSM) is a formal specification method that is widely studied and used to model software systems, especially sequential circuits and communication protocols. FSM is broadly used perhaps partly due to the fact that at a low level of abstraction, a system is often most easily understood as a finite state machine [Dsso97]. Because conformance testing for protocols is of great importance, researchers have been studying the ways to automate conformance testing tasks such as to generate test sequences (test cases) based on FSM specifications. The techniques developed in the past for generating test sequence for FSM all require that an FSM model has some properties or can be transformed to an FSM that has such properties. Generally, to test a system specified in FSM, two things need to be checked. One is to verify that the transition produces the expected output, and the other is to identify that the transition leads to the correct target state.

A state transition diagram or a state table is used to represent an FSM. A state transition diagram is a directed graph whose vertices correspond to the states of the machine and whose edges correspond to the state transitions; each edge is labeled with input and output associated with the transition [Lee96], while a state table is a table used to represent the same states, input/output and state transitions in text format.

In [Sidh89], author describes the four most used protocol conformance test sequence generation techniques: T-method, U-method, D-method and W-method, and the prerequisites for the implementation of each method. The T-method, also called Transition Tour method, was originally proposed by Naito and Tsunoyama [Nait81]. This method is equivalent to a solution to a graph theory problem of searching for the shortest covering path of a directed graph. Here, checking of next state might be omitted. This method generates the shortest test sequence among the four aforesaid methods. Although the T-method cannot guarantee the fault coverage and sometimes additional tests are required. It has been deemed as the best and most practical test sequence generation method, and many researchers showed interest in it and have developed variants based on it.

Although there are many FSM-based automatic test case generation methods and tools implemented, they all have some limitations. First, these methods all make certain assumptions of an FSM model, e.g., requiring minimal, strongly connected, completely specified FSM, and secondly the FSMs structure is flat and non-hierarchical. It can only be used to model sequential systems without concurrency. Moreover, FSM specifications only deal with control flow, not the data flow, for example, it cannot have variables in states nor operations determined by variable values.

Extended Finite State Machine extends original FSM model in two ways. First, a transition has a function (with variable) rather than a single pair of input/output values and second, each state represents a set of values (with variable) for the internal state rather than a single value. In other words EFSM extends FSMs by adding three elements: hierarchy, concurrency and communication, making them suitable for specifying the behaviors of complex reactive systems [Dsso97].

1.6.2 Model Based Approach

Testing based on Z specifications was first proposed by Hayes [Hayes86] in the context of testing abstract data types. Hayes described the testing problem as checking that the state invariants and pre-conditions are maintained in the implementation as well as the input-output relation.

Hall [Hall88] first described the possibilities for automatically generating test domains for Z specifications based upon the partitioning of input sets, output sets and states. He suggested automatically parsing the specification while applying attributed grammar style operations to develop the partitions.

Dick and Faivre [Dick93] suggested partitioning the specification into disjunctive normal form as a means of automatically extracting test cases from VDM-SL specifications. However, this leads to unstructured test sets with limited fault detection properties. The paper also described a method by which test cases could be sequenced.

1.6.3 Category-Partition Testing

The category partition method [Ostr88] is based on partition testing and equivalence classes. The principle of partitioning the input domain with respect to properties of the specification was first introduced in 1975 in a paper by Goodenough and Gerhart [Good75]. Equivalence classes are formed by partitioning the input domain into sets of data that exhibit similar behavior in the specification. It is then assumed that only one test point needs to be selected from each equivalence class to verify the implementation against the specification. The first step of the category partition method consists of decomposing the specification into functional units that can be independently tested. The functional units are then analyzed to identify the parameters and environment conditions that affect the function's behavior. Parameters are explicit inputs to operations and environment conditions are properties of the system state which hold when the function is executed. The next step is to choose categories of information that characterize major properties of the parameter or environment condition. Each category is then partitioned further into choices which represent sets of similar values (equivalence classes). The individual test cases can then be created by selecting values from choices for each category, taking into account the constraints on how the different choices interact.

Amla and Amman [Amla92] took Ostrand and Balcers [Ostr88] description of the category partition method and used it as the basis for a method of deriving tests from Z specifications. They suggested that category partition method is applicable to natural-language functional specifications, which may be incomplete and unstructured. The testers will need undue effort to define testing requirements, thus hampering the effectiveness of the method. On the other hand, they argue that testing requirements are, to a large extent, already captured in formal specifications. They analyze the feasibility of applying category partition method to Z specifications and verify that testing requirements can be derived from the formal specifications more easily.

Using the notion of test templates, Stocks and Carrington [Stock96] developed a unified, flexible, and formal framework for specification-based testing. Their framework provides not only a formal model of tests and test suites, but also a method for applying the model in testing. In this way, test suites can be constructed in a concise and formal manner. They also investigate several application areas of the framework, including test oracles, refinement and regression testing (defined earlier in section 1.2). However, the heuristics used to partition the specification were left informally specified and the scope for automating the partitioning process itself was left unexplored.

The Classification Tree Method [Groch93] is a similar technique for repeatedly applying test heuristics to a specification to derive a hierarchy of abstract test case specifications. Leaves in the tree are combined and instantiated to form the test data. Tool support has been developed for this method and allows a tester to structure and select test cases using an intuitive user interface. The testing heuristics used by the method were restricted to type analysis and reduction to disjunctive normal form.

The approach presented in [Chen03] is an extension of Ostrand and Balcers [Ostr88] approach which captures different constraints among various ranges of values of the parameters and environment conditions (choices) and then combines these to form test frames. The algorithm checks for consistency and performs automatic deductions of relations between the choices. In [Ostr88], where the number of generated test frames can only be reduced by means of incorporating additional constraints among choices, as a result, the tester does not have a direct control on the exact number of test frames generated. Also, after all the constraints have been taken into consideration, further reduction will not be possible. But in [Chen03], the authors claims that there is a systematic approach in reducing the number of test frames from the choice relationship which in turn are controlled by constraints. This is done by prioritizing the individual choices depending upon the software testers expertise and experience in the application domain. In this way, the choices with higher priorities can first be used to generate test frames, thus respecting both the resource constraints and the relative importance of the choices.

1.6.4 Shortcomings of Above Discussed Approaches

The techniques described above have addressed the problem of constructing test cases but have not included the issues of selecting test data or checking the outputs of the implementation against the input/output relations given in the specification. The ability to select representative test data and to check the results of the tests against some correct implementation of the requirements is essential to be able to implement a complete testing strategy. Therefore it is important to ensure that test case generation techniques are compatible with appropriate test oracle and test data generation techniques.

Apart from these methods used to generate test cases, we also discuss one more approach which has received widespread recognition in the researching community and in the industry.

1.6.5 Generation of Test Cases Using Logical Specification

The method presented in [Mand95] is a novel approach for semi-automatically generating the test case by proposing a tool for functional test case generation for real time systems. This is done by using formal specification language TRIO, which is an extension of temporal logic defined to deal with strict timing requirements. The proposed method can be applied to verification of any kind of system, not explicitly to software/hardware verification. As the TRIO formulas are executable, they can be easily used to check validity or satisfiability.

If A is a formula and t is a term of the temporal type, $Futr(A,t)$ and $Past(A,t)$ are formulas: their intuitive meaning is that the formula A holds at an instant t time units away in the future (resp., in the past) with respect to the current time value. This can be shown by:

$$in \longleftrightarrow Futr(out, 5) \qquad \text{(F)}$$

Here the time dependent predicate in means that a message has arrived at one end at the current time and the predicate out means that a message is emitted from the other end in 5 time units.

A leaf tableau is generated by decomposing the TRIO formula which we get from the logical specification of the system. These tableaus are infact properties that hold at a certain point in time, where each property is called an event, and further each set of events is called a history.

A history is complete if it contains a unique truth value for each predicate of the formula at each instant of the time domain. A complete history satisfies a formula F at time i if its evaluation in which predicate values are defined by the events of the history yields a *true* value. Thus, each leaf tableau that does not contain any contradiction contains a history.

The algorithm that checks the satisfiability of a TRIO formula is called a history generator. The history generator produces the elementary test cases from the specification provided by the user. The subset of test case is called partial test case. Also the complexity of the history generation algorithm is exponential with respect to both the number of quantifications in the formula and the cardinality of the interpretation domains. This interpretation algorithm is called a history checker. Thus the history generator and the history checker are used as the core for this tool for systematically generating functional test cases from TRIO specification.

During the interpretation of a formula F specifying a given property of a system, behaviors (i.e., simulations) of the specified system compatible with F are generated, they are called histories. The main idea underlying this approach is to use histories as test cases for the system implementation. A test case for a given formula F is a complete history that satisfies F at one or more instants of the time domain and this test case doesnt have classification of input events and output events separately.

In general, there is hardly any need of testing the complete expected system behavior. Here they try to identify a limited number of relevant events and check the system behavior against them and try to extrapolate the behavior under all remaining circumstances.

The paper supports the proposed approach by providing two case studies. The first case study (Therac-25) describes about the accident that occurred at the East Texas Cancer Center in Tyler, Texas. This study illustrates, how test cases could have been generated that would have uncovered the flaws that led to the accidents reported in the literature of the

Therac-25 mission. They showed, how the related functional requirements can be easily specified in TRIO and suitable test cases can be produced in a simple and natural way from such specifications.

The second case study illustrates about ELSA (Experimentation of a logical approach to the specification and verification of automation control systems), an industrial project recently conducted in the framework of the European Union program ESSI. This project concerned specification, design and verification of the system controlling load balance in the energy production station of pondage power plants operated by the Italian Energy Board.

TRIO has proved to be an adequate language for real-time specification. However, its use becomes difficult when considering large and complex systems, because TRIO specifications are very finely structured and the language does not provide powerful abstraction mechanisms and lacks an intuitive and expressive graphical notation.

1.7 Background and Motivation

There are continuing efforts in component based system for composition of systems using different approaches [Berg96]. The paper presents generic object-oriented models which are useful in analyzing and relating synchronization and real-time constraint-inheritance anomalies in a uniform way. Based on these generic models, a number of important synchronization and real-time inheritance anomalies are identified and discussed. They present a few possible solutions to both synchronization and real-time constraint-inheritance anomalies and propose modular and composable synchronization and real-time specification extensions to the object-oriented model using the concept of composition-filters. The applicability of the proposed mechanisms has been illustrated through various examples. Continuing along the same theme, we started exploring the use of category theory in composing modules having real time and synchronization constraints [Varm03]. The paper introduces the formal framework to facilitate the composition process using category theoretic approach, and show the correctness of composition with constraints using these concepts. Even though the concept of contracts has been introduced and widely used in the computing world [Raus02], the [Varm03] approach utilizes the concept quite effectively. The contracts, which are basically a sets of constraints defined from the system specification, which play important role in composition of different modules of the system.

We have taken *Mine pump problem* [Jose97] as the classical example to formulate and use it as the case study for the proposed approach. Firstly, all the components of the mine pump system are identified and each of these components are specified formally with sorts, operations and equations for their parameter, import and export interfaces. A set of contracts or constraints for each of these components are defined along with their specification. Then the internal and external contracts are defined based on the requirement of the mine pump system. These contracts are set of constraints which a component or a system has to satisfy. Furthermore, internal contracts are the constraints imposed on the stand-alone component; this generally deals with the initial values and constraints on the operations that can be performed by the component. External contracts are introduced as a result of inter-component interaction. The resulting constraints being imposed effect on the operation of the interacting components.

These contracts are then mapped with morphism function, such that two categories or components combine to form a composed category or subsequently reusable component. In other words, morphism that combines two components is the functional implementation of the internal and external contracts that exist in each of the components. Thus, it can be summarized that morphisms are derived from the contracts that exist in each of the components.

The morphism function which gets defined above, play important role in composition of system. The composition of each module is done using *union* operation and morphism governs the exposition of contracts of different modules. The final resultant composition should provide the correctness of the composed module. The benefit of this framework is that it facilitates tracing of impacts of influences of a specific constraint imposed on a module could have on other modules over an interaction. The proposed approach takes into account the real time and synchronization constraints imposed on the system. Once the system is composed using the category theoretic approach, the need of verification and validation of the composed system is very essential to prove the correctness of the composition. So the need for testing in component based system arised in our ongoing research work.

1.8 Our Contributions

As compared to other approaches which were discussed above, ours approach attempts to integrate component-based design of complex systems and constraint-based test generation into a unified framework. The proposed framework CAGILY, that incorporates the concept of component identification and specification, and defines contracts/morphisms and constraint-based cross-product in category theory. The component identification is done by decomposing the system hierarchically into their primitive subcomponents. The identified components are then specified using four interfaces Parameter, Import, Export and Body. Contracts are introduced as a result of inter-component interaction and are derived from the specification. The morphisms define a rule in which two categories or components interact to subsequently form a system or a reusable component. This morphism that combines two components is the functional implementation of the contracts that exist in each of the components.

The first step in test case generation procedure is the constraint cross products between the components, which is done till we get composite constraint cross products of all the components. The purpose of introducing cross product in the cagily framework is to capture all the necessary and possible set of variables covering different scenarios. Constraints are used to restrict the irrelevant and unnecessary sets of variables still satisfying the test coverage. Finally, Sampurna Tool is used to generate the final set of test suites from the composite component.

1.8.1 Sampurna Tool

Once we arrive at the sets of variables/export functions, which form the final constrained-cross product for the entire composite system. The first step in Sampurna tool is to bring (import) the relationship chain (sets of variables associated to respective export functions) to the final sets of variable considering all the possible combination of values/range for which the variables may attain. This ensures that we arrive at the end values which are the actual inputs to the system.

After constraining the cross product between the categories, we arrive at a smaller list of relationships which are still having some sets of test cases which are meaningless and redundant. There is a need to reduce this set of test cases so as to optimize set of test cases which will satisfy the functional aspects of the system. This is achieved by using a priori knowledge of the system such that the variable-value pairs that are not attainable/possible with respect to the system specification are not included in the resulting test cases.

The final step in Sampurna tool is to parse the set of test cases with respect to contracts so as to obtain the expected output of the system. This associated values/range of the variables will steer the system through different state and will help in finding any discrepancies as compared to the correct behavior of the system. The final resultant test suite which we get after this iteration is a complete set of test cases which targets the critical functional testing of the system. This framework is illustrated by using the classical mine pump problem.

The proposed framework is specifically used to test the functional aspects of the system. And this functional aspect can only be tested if the system is specified correctly. Also to make the test case generation process more robust, the process should be made automated, which will reduce the chances of human errors during the testing process.

We have specified and verified the composition of the system using the Kestral Institute's *Specware tool.* And finally implemented the Sampurna tool using database tool.

1.9 Outline of the Thesis

In Chapter 2, we provide definition and concepts of category theory, which we used for our proposed approach. This is followed by discussion on *Specware Tool*, which is used for proving the correctness of the formal specification we have derived. Chapter 3, describes the CAGILY approach for generating functional test cases using category theoretic approach. This will be followed by a case study of classical mine pump problem involving decomposition, component specification and generating set of constraints from the given specification. Further, we have shown the correctness of the composed specification using Specware. The implementation of the Sampurna Tool is illustrated in Chapter 4. We have illustrated how Sampurna Tool can be used for generation of functional test cases. Finally Chapter 5 concludes with discussions and future research directions.

Chapter 2

Background - Category Theoretic Approach for Composition

In this chapter, we explain a formal framework utilizing concepts of category theory. At first, we provide an overview of the modularization in category theory by providing different building blocks of the modules and further providing details regarding different terms and definations. Next we introduce the algebraic specification of modules using category theory along with their interconnections. Lastly we provide details about the *Specware tool* that we have used in this thesis for the purpose of specifying and proving the correctness of the composed modules.

2.1 Category Theory

In this section, we will explain the formal role of category theory concept which forms the basis of the approach presented in this thesis.

Category theory is a relatively young branch of mathematics stemming from algebraic topology, and designed to describe various structural concepts from different mathematical fields in a uniform way. Category theory provides a bag of concepts and theorems about those concepts that form an abstraction of many concrete concepts in diverse branches of mathematics including computing science. Hence it will come as no surprise that the concepts of category theory form an abstraction of many concepts that play a role in algorithmics.

Quoting Hoare [Hoare89]“Category theory is quite the most general and abstract branch of pure mathematics. The corollary of a high degree of generality and abstraction is that the theory gives almost no assistance in solving the more specific problems within any of the sub disciplines to which it applies. It is a tool for the generalist, of little benefit to the practitioner {.......}”.

The language of category theory facilitates an elegant style of expression and proof (equation reasoning) for the use in algorithmics, this happens to be reasoning at the function level, without the need (and the possibility) to introduce arguments explicitly. Also, the formulas often suggest and ease a far-reaching generalization, much more so than the usual set-theoretic formulations.

Category theory has itself grown to a branch in mathematics, like algebra and analysis that is studied like any other one. One should not confuse the potential benefits that category theory may have (for the theory underlying algorithmics) with the difficulty and complexity, and fun, of doing category theory as a specialization in itself.

2.2 Categories in Category theory

A category is a collection of data that satisfy some particular properties. So, saying that such-and-so forms a category is merely short for asserting that such-and-so satisfy all the axioms of a category. Since a large body of concepts and theorems has been developed that are based on the categorical axioms only, those concepts and theorems are immediately available for such-and-so if that forms a category. For an intuitive understanding in the following definition, one may interpret objects as sets, and morphisms as typed total functions, we shall later provide some more and quite different examples of a category, in which the objects arent sets and the morphisms arent functions.

It has been lately established that the mathematical axioms present in category theory can be used in the field of computer science. The concepts proposed by category theory have been identified by various people [Guo02, Willi99]. The concepts have been widely used in many researches which aim to propose a formal methodology to the development process of software systems. We try to exploit the mathematical axioms in our proposed CAGILY framework and the reason for choosing Category theory is the ease in translating functional aspects of the system to mathematical notations.

2.3 Modularization in Category Theory

Modularization is an important aspect in component based system. Modularization helps us in identifying different building blocks of the systems, thus achieving decomposition of the system. But this modules when combined together to form a composite system, yields to provide better understanding of the system. Modularity is maintained by structuring the description of the system in terms of modules that allows very precise description. A module comprises of three components, Interface - which collects all the resources including internal and external relationships, Construction - this defines individual functioning and denotation of resources, Behavior it represent the particular semantic view of the module. These components form the conceptual units which are syntactically and semantically provided. The basic components of the modules are:

- *Module Modules* - form the building blocks of a modular system. They represent particular system components that should be seen as a unit.
- *Module Interconnections* - It forms the architectural structure of the module. They represent the way of interaction between the modules.
- *Operations on Modules* - Operations define the modules and their interconnections

and they are given syntactically and semantically.

A Category [Pier91] consists of a collection of C objects and C arrows (morphism) between objects. And these two collections have to respect the following properties:

- C -arrows are composable. In other words, each morphism f is associated with an object A that is its domain and an object B that is its codomain. i.e., for all morphisms $f : A \rightarrow B$ and $g : B \rightarrow C$, there exists a composed morphism $gof : A \rightarrow C$.
- Arrow composition is associative, for all $h : C \rightarrow D$, $h \; o \; (gof) = (hog) \; o \; f$.
- There is a C -arrow from each object to itself. This is called the identity morphism.

2.3.1 Signature

A signature $SIG = (S, OP)$ consists of a set S, the set of sort, and a set OP, the set of constant and operation symbols. The structure SIG is called a signature.

2.3.2 Specification

A specification $SPEC = (SIG, AX)$ consists of two parts: the signature SIG and a set of axioms AX which describes the behavior of the system with respect to the environment.

Thus a signature defines only the syntax of a solution and extending the signature with axioms gives the signature operations. Further, this extension with their respective axioms gives specification.

2.3.3 Specification Morphism

A specification morphism is a map from the sorts and operations of one specification to the sorts and operations of another similar specification such that,

1. axioms are translated to theorems, and

2. source operations are translated compatibly to target operations.

The operation by which the objects can be linked together or manipulated is shown in the following subsection.

2.3.4 Pushout Operation

Module specifications are defined by utilizing the notion of push-out operation from category theory. Given specifications A and B, and a specification R describing syntactic and semantic requirements along with two morphisms f and g, the push-out operation gives specification P which contains A and B.

$$\begin{tikzcd}
R \arrow[r, "f"] \arrow[d, "g"'] & A \arrow[d, "h"] \arrow[ddr, dashed, "h1"] & \\
B \arrow[r, "k"] \arrow[drr, dashed, "k1"'] & P \arrow[dr, "i"] & \\
 & & P1
\end{tikzcd}$$

Figure 2.1: Specification

Formally, given specification morphism $f : R \rightarrow A$ and $g : R \rightarrow B$, a specification P together with specification morphisms $h : A \rightarrow P$ and $k : B \rightarrow P$ is called the push-out of f and g, provided that the module commutes, i.e., $hof = kog$, where denotes composition. Furthermore the following universal condition holds:

for *all objects P1 and all morphism H1: A→P1 and K1: B→ P1 such that H1 o F = K1 o G, there exists an unique morphism I : P → P1 such that I o K = K1 and I o H = H1.*

The later part of the pushout definition ensures that the P chosen to construct the pushout is the minimal. Module specifications are defined by utilizing the notion of push-out operation.

2.3.5 Module Interface

The term module interface itself defines what services a module provides and what it requires in order to be used in a system. These module interfaces helps in interacting with other modules using export and import interfaces that are essentially specifications and the sorts and operations linked together through specification morphisms.

$$\begin{tikzcd}
\text{Parameter (R)} \arrow[r, "f"] \arrow[d, "g"'] & \overset{\text{(Resource provided by this module)}}{\text{Export (A)}} \arrow[d, "h"] \\
\underset{\text{(Resource provided by other modules)}}{\text{Import (B)}} \arrow[r, "k"'] & \text{Body (P)}
\end{tikzcd}$$

Figure 2.2: Module

An algebraic module specification consists of components, called import, export, parameter and body. A module specification $MOD = (PAR, EXP, IMP, BOD, f, h, g, k)$ consists of four specifications:

a) PAR, parameter specification

b) EXP, export interface

c) IMP, import interface specification

d) BOD, body specification

And their four mapping morphisms f, h, g and k such that the following diagram commutes, i.e., $f \circ h = g \circ k$.

The brief description of all the modules is given below:

2.3.6 Parameter Interface (PAR)

The parameter module is the combination of import and export modules and sometimes intersection of both the modules. With this parameter part, divide and conquer abstraction is enhanced, as well as reusability of a module is obtained. It can be seen as the parameter of the whole module as it appears to the outside by its interface.

2.3.7 Import Interface (IMP)

The import interface is used to specify those resources which are to be provided by other modules which are further used in the modules body for construction of the resources to be exported. It is an algebraic specification consisting of a signature, which names and types the resources to be imported and eventually lists properties of these resources, which form restrictions for the import of actual resources and provide information for the use of these resources in the body of the module. The explicit formulation of an import interface is especially useful in the stepwise development of a modular system. It allows a top down way of construction where resources are named and used, but only later to be realized by other modules.

2.3.8 Export Interface (EXP)

The export interface contains those resources which are realized by the module and are to be used by other modules or an application environment. In a module specification these resources are declared in the same way as the resources of the import interface. It restricts sorts and operations treated in a module to those which are visible for the user of the module. This realizes hiding of resources, which serves the purpose of protection of resources, abstraction from internal details, and independence from particular forms of construction in the body of the module.

2.3.9 Body Interface (BOD)

The body part of a module contains the construction of the resources declared in the specification of the export interface. For this purpose, the body may contain auxiliary sorts and operations which do not belong to any other part of the module but depend on the particular choice of construction. The realization of sorts and operations declared in the specification of the export interface is encapsulated in the module, not accessible to the user of the module

All the above concepts are expressed and implemented in Slang language using Specware tool.

2.4 Specware

In this section we discuss importance of Specware Tool and its usage in specifying and verifying the concepts of our approach.

Specware is an automated software development tool that allows users to preceisely specify the desired functionality of their applications and to generate provably correct code based on these requirements[Srini96]. The foundations of Specware are category theory, sheaf theory, algebraic specification and general logics. Using Specware, one can construct formal specifications modularly and refine such specifications into executable code through progressive refinement. Further, Specware allows you to express requirements as formal specifications without regard to the ultimate implementation or target language. Specifications can describe the desired funcitonality of a program independently of such implementation concerns as architecture, algorithms, data structures, and efficiency. This makes it possible to focus on the correctness, which is crucial to the reliability of large software systems. Using Specware, the analysis of the problem can be kept separate from the implementation process, and implementation choices can be introduced piecemeal, making it easier to backtrack or explore alternatives.

Specware can be further catagorized into

- **Description :** Descriptions in Specware are written in one of several logics. They basically represent a collection of properties, which we intend to build. Descriptions are then progressively refined by adding more properties, till a model is reached which satisfes these properties.

- **Composition:** Refinement, complexity and scale are handled well by utilizing the composition operators which allows complex descriptions to transform into smaller ones. The colimit operation from category theory is pervasively used for composing structures of various kinds in Specware.

Besides composition operators, one needs bookkeeping facilities and information presentation at various abstraction levels. Specware uses category theory for bookkeeping and abstraction.

Specware allows to articulate software requirements, make implementation choices, and generate provably correct code in a formally verifiable manner. The progression of specifications forms a record of the system design and development that is invaluable for system maintenance.

Thus, in this chapter we have tried to discuss the concept of category theory which is used for our proposed approach. We have given a brief description on the formal verification tool Specware. Further, in next chapter we will discuss our approach (CAGILY) and illustrate the framework using a case study of Mine Pump Problem.

Chapter 3

CAGILY - An Approach for Generating Test Suites

In this chapter, we propose the framework for generating test suites for component based system using category theory. We then describe various steps involved in generation of test suites. Next we illustrate our proposed approach by providing a case study of mine pump problem. And finally we provide formal specification and verification of the illustrated case study using Metaslang language of Specware.

3.1 CAGILY Framework

Testing is an important phase in software development process to identify any discrepancies between the actual behavior of the implemented system's functions and the desired behavior as described in the system's functional specification.

The objective of the CAGILY (**CA**te**G**ory framework for **I**dentifying test cases for-mal**LY**) framework is to present a systematic and methodological approach for functional test case generation from a system specification. The CAGILY framework involves the following five steps in test case generation of a specified system as discussed in the subsequent subsections,3.1.1 through3.1.5.

3.1.1 Component Identification

The first step in the CAGILY methodology is component identification by decomposing the system. System decomposition requires domain knowledge and tends to be somewhat complex as the present day systems span across various domains. We follow a formal methodology to decompose the system. Essentially, we view the system as being structured in a hierarchical manner. We first sub-divide the system into components and then further continue to sub-divide these components until the primitive components that can be implemented as single artifacts are found.

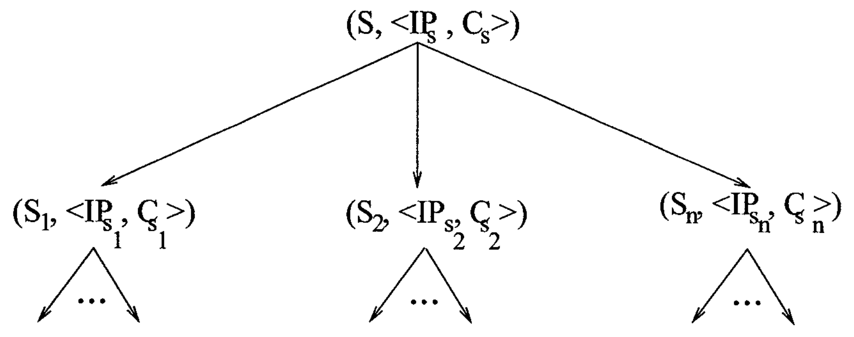


Figure 3.1: Hierarchical System Decomposition

In order to represent the hierarchical design process a system is represented not by a single structural design (IP, C) (IP is the set of interaction points, and C is the set of components), but rather by a set of structural designs $\{(IP_1,C_1),..., (IP_n,C_n)\}$ where each element of the set represents the design at a particular level of the design hierarchy. For example, assume that the structure of a system S is to be specified. The first step is to derive the top level structural specification (IP_s,C_s) which defines the structrual decomposition of S. Given that $C_s= (C_{s1},..., C_{sn})$ structural specifications can be created for C_{s1}, ... , C_{sn} which gives the decomposition of the components of S. The decomposition process then proceeds to the required level of granularity. The Graphical representation of the decomposition process is as shown in Figure 3.1.

3.1.2 Component Specification

In our framework, a component (or module) consists of four interfaces, namely *Parameter, Import, Export* and *Body* (See Figure3.2 and refer to[Ehrig90] for further details). Formally, given specifications A and B, and a specification R describing module's requirements along with two morphisms f and g, the *push–out* operation gives the component specification P which contains A and B.

$$\begin{tikzcd}
R \arrow[r, "f"] \arrow[d, "g"'] & A \arrow[d, "h"] \arrow[ddr, dashed, "h1"] & \\
B \arrow[r, "k"] \arrow[drr, dashed, "k1"'] & P \arrow[dr, "i"] & \\
 & & P1
\end{tikzcd}$$

Figure 3.2: PushOut Operation

As shown in Figure3.2, given specification morphism $f : R \rightarrow A$ and $g : R \rightarrow B$, a specification P together with specification morphisms $h : A \rightarrow P$ and $k : B \rightarrow P$ is called the *push–out* (*of f and g*), provided that the module commutes, i.e., $h\ o\ f = k\ o\ g$, where o denotes composition.

3.1.2.1 Composition of Module Specifications

As illustrated in [Ehrig90], the composition scheme allows two modules to be interconnected via export and import interfaces. The push-out of the two modules is the resulting specification of the composed module. Figure3.3 depicts the composition operation, where Module 1, $M_1 = (R_1, A_1, B_1, P_1)$ and Module 2, $M_2 = (R_2, A_2, B_2, P_2)$. Module1 imports via specification B_1 whatever Module2 exports via specification A_2. The resulting composed module $M_{12}$is (R_1, A_1, B_2, P_{12}), where $P_{12}$is the push-out of $P_1$and P_2over B_1.

$$\begin{tikzcd}
 & & R_1 \arrow[r, "f_1"] \arrow[d, "g_1"'] \arrow[dll, "t"'] & A_1 \arrow[d, "h_1"] \\
 & & B_1 \arrow[r, "k_1"'] \arrow[dl, "s"'] \arrow[ur, phantom, "\text{Module 1}\ M_1"] & P_1 \arrow[dd, "m_2"] \\
R_2 \arrow[r, "f_2"] \arrow[d, "g_2"'] & A_2 \arrow[d, "h_2"] & & \\
B_2 \arrow[r, "k_2"'] \arrow[ur, phantom, "\text{Module 2}\ M_2"] & P_2 \arrow[rr, "m_1"'] & & P_{12}
\end{tikzcd}$$

Fig (a)

$$\begin{tikzcd}
R_1 \arrow[r, "f_1"] \arrow[d, "g_2 o\, t"'] & A_1 \arrow[d, "m_2 o\, h_1"] \\
B_2 \arrow[r, "m_1 o\, k_2"'] \arrow[ur, phantom, "M_{12}"] & P_{12}
\end{tikzcd}$$

Fig (b)

Figure 3.3: Composition of Two Modules

3.1.2.2 Composition with Constraints

Given two module specifications with constraints $MC_1$and MC_2, the resultant module specification with constraints can be given as MC_{12}= $\{(R_1, C_{R1}),(A_1, C_{A1}),(B_1, C_{B2}),P_{12}\}$. The graphical representation of this composition with constraints is as shown in figure3.4. The detail description of the concept is given in [Ehrig90].

$$\begin{tikzcd}
(R_1, C_{R1}) \arrow[r, "id"] \arrow[dd, "t"'] & (R_1, C_{R1}) \arrow[r, "f_1"] \arrow[d, "g_1"'] & (A_1, C_{A1}) \arrow[d, "h_1"] \\
 & (B_1, C_{B1}) \arrow[r, "k_1"'] \arrow[d, "s"] \arrow[ur, phantom, "MC_1"] & P_1 \arrow[dd, "m_2"] \\
(R_2, C_{R2}) \arrow[r, "f_2"] \arrow[d, "g_2"'] & (A_2, C_{A2}) \arrow[d, "h_2"] & \\
(B_2, C_{B2}) \arrow[r, "k_2"'] \arrow[ur, phantom, "MC_2"] & P_2 \arrow[r, "m_1"'] & P_{12}
\end{tikzcd}$$

Figure 3.4: Composition of two modules with constraints

3.1.3 Contract Definition

A software component can be defined as an independently deployable unit of composition with contractually specified interfaces[Szyp97]. In component-based engineering, *contracts* are the services that suppliers *offer* to potential clients[Raus02].

We define contracts as a set of constraints which a component or a system has to satisfy. Contracts are introduced as a result of inter-component interaction and are derived from the specification. The resulting contracts are imposed on the operation of the interacting components.

3.1.4 Mapping Contracts to Morphism

Morphisms define a rule in which two categories or components interact to subsequently form a system or a reusable component. Contracts play an important role in the morphism function definition. The morphism that combines two components is the functional implementation of the contracts that exist in each of the components.

3.1.5 Test Case Generation Procedure

The concept of cross product is introduced in this framework to capture all the possible combination of variables so as to generate set of test case scenarios. The constraints are applied over this cross product to restrict the irrelevant test cases thus achieving comprehensiveness and still satisfying test coverage. After obtaining the final constraint-cross product, based on *a priori* knowledge of the working principle of the system, the redundant and irrelevant test cases are being removed.

3.1.5.1 Cross Product of Categories

As illustrated in[Pier91], the **Product Category** of any pair of categories C and D has an object pair (A, B) such that A is an object of category of C and B is an object of category of D, and the morphisms are also pairs, consisting of one morphism in C and one in D. Such pairs can be composed component-wise.

3.1.5.2 Constrained Based Cross Product

The objective of having constraint-based cross product is to ensure that operations are only applied to expressions that are relevant[Doye97]. Large sets of test cases will be generated by taking cross products between the modules, and will be impractical to apply all those test cases. By doing a constraint-based cross product between the modules, we thereby remove some combination of variables which are not relevant.

3.1.5.3 Sampurna: Test Suite Generation Tool

Sampurna generates a set of test cases from the final category obtained as shown in the previous Section3.1.5.2. Further, it eliminates the variable-value pairs that are not attainable/possible with respect to the system specification by using *a priori* knowledge of the system.

Finally, it parses and generates a suite of test cases with respect to the specified set of constraints. The expected output of the tool is a test case containing variables and their associated values/range of values that would steer the system through different states so as to detect any discrepancies with respect to the expected correct behavior of the system. We have discussed test case generation procedure and Sampurna Tool in subsequent sections using a case study of classical mine pump problem.

The fundamental rule (rule extraction) used in this Sampurna tool can be written as follows. Consider a set of condition : $A{+}B = C$, where A, B and C denotes the variables used in the set of constraints. So when we take value C' then behaviour does not have values of A' and B' into its equation. Likewise we identify all such sets of variable value pairs and remove them from the set of test cases.

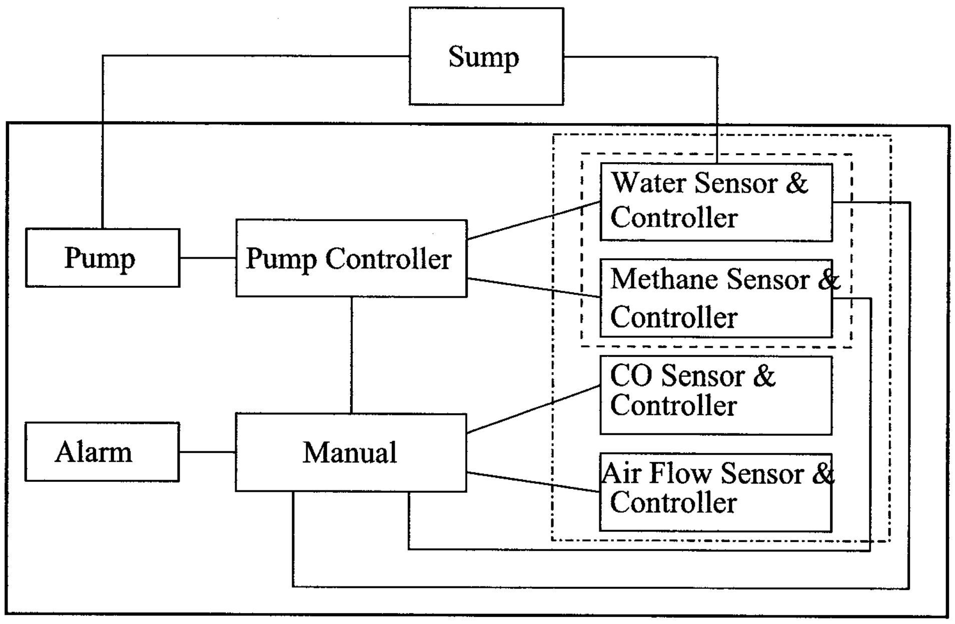

Sump
Pump
Pump Controller
Alarm
Manual
Water Sensor & Controller
Methane Sensor & Controller
CO Sensor & Controller
Air Flow Sensor & Controller

Figure 3.5: Mine Pump Controller

3.2 Mine Pump Problem - An Overview

We illustrate our proposed approach through the case study of mine-pump system. Water percolating into a mine is collected in a sump to be pumped out of the mine (See Figure3.5). There are certain safety requirements associated with the operation of the pump[Jose97].

The system has four sensors: a water-level sensor, a methane sensor, a carbon-monoxide sensor, and an airflow sensor. The different sensors are considered as independent modules. All sensors read data periodically. Like the water level sensor detect when water is above a high and a low level. A pump controller switches the pump *on* when the water reaches the high water level and off when it goes below the low water level. If, due to a failure of the pump, the water cannot be pumped out, the mine must be evacuated within certain time T. Similarly when any of the other three sensors go high, then an alarm must be raised and the operator informed within one second of any of these levels becoming critical so that the mine can be evacuated within one hour. To avoid the risk of explosion, the pump must be operated only when the methane level is below a critical level.

Human operator can also control the operation of the pump, but within limits. An operator can switch the pump *on* or *off* if the water is between the low and high water levels. A special operator, the supervisor, can switch the pump *on* or *off* without this restriction. In all cases, the methane level must be below its critical level if the pump is to be operated. Reading from all the sensors, and a record of the operation of the pump, must be logged for later analysis.

There are certain other information which the specification does'nt provide accurately. For example, the pump may fail when the water is at any level, does the time of one hour for the evacuation of the mine apply to all possible water levels? More crucially, how is pump failure detected? Is pump failure always complete or can a pump fail partially and be able to displace only part of its normal output? And what is the gurantee that the sensors will always work correctly. Similarly, the controller system obtains information about he level of water from Highwater adn Lowwater sensors and of methane from Methane sensor. Detailed data is needed about the rate at which water can enter the mine, and the frequency and duration of methane leaks; the correctness of the controller is predicated on the accuracy of this information. But finally, to satisfy the safety requirements of the mine pump, we have identified certain operating condition of mine pump, which needs to be fulfilled for proper functioning of the mine pump system.

The operating conditions of the mine-pump are:

1. The water should be pumped out whenever the water level rises above the high-water level.

2. The pump must not be operated if the methane level or any air level is critical.

3. The pump can also be operated manually. An operator can switch the pump on when water level is between low-water and high-water levels. However, a supervisor can switch the pump on without any restrictions.

4. Both the users should check the methane level and other air-levels before switching

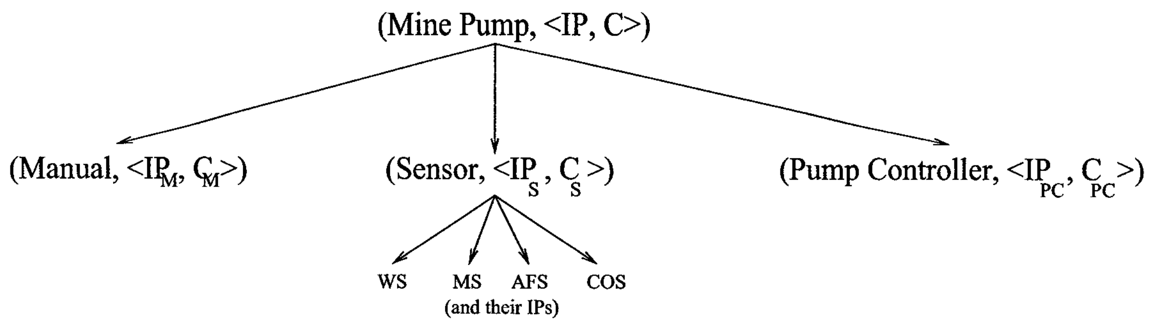

(Mine Pump, <IP, C>)
(Manual, <IP$_M$, C$_M$>)
(Sensor, <IP$_S$, C$_S$>)
(Pump Controller, <IP$_{PC}$, C$_{PC}$>)
WS
MS
AFS
COS
(and their IPs)

Figure 3.6: Hierarchical Decomposition of Mine-Pump System

pump on.

These conditions are the system requirements and the set of test cases that are to be generated should cover these scenarios while testing the implemented system Next, we follow the steps outlined in Section3.1 to generate test cases for the mine-pump system.

3.3 CAGILY - A Detailed Illustration

3.3.1 Component Identification

System functionalities, as identified by interaction (or interface) points, can be used by external entities (as in the case of the supervisor) or by other methods within the system. Based on the requirements, we identify following interface points for the mine pump system.

- **Interface Points (IP)** = *{GetAirLevel(ReadAirSensor), GetMethaneLevel (ReadMethaneSensor), GetCOLevel(ReadCOSensor), GetWaterLevel(ReadWaterSensor), GetUserInput(), StartPump(), StopPump(), StartAlarm(), StopAlarm()}*

In the next step, we identify objects or components which will possess these IPs. Note that there can be many other internal functions that might exist in the components besides these external functionalities. The Figure3.6 illustrates the component identification of mine pump system.

- **Components (C)** = *Sensors* with IPs {GetAirLevel(ReadAirSensor), GetMethaneLevel (ReadMethaneSensor), GetCOLevel(ReadCOSensor), GetWaterLevel(ReadWaterSensor), *ManualOperator* with IPs {StartAlarm(), StopAlarm()}, *Controller* with IPs {StartPump(),StopPump()}

Here, for IPs and Cs, the notation $X(Y)$, Y denotes the name of the function that the interface function X invokes. After having identified IPs and Cs for the mine pump system, we further sub-divide the *Sensor* component into different sensor modules. The following are the constituent components for the mine-pump system.

1. Water Sensor (WS) module
2. Methane Sensor (MS) module
3. Carbon Monoxide Sensor (COS) Module
4. AirFlow Sensor Module (AFS) Module
5. Manual Module (M) Module
6. Pump Controller (PC) Module

3.3.2 Component Specification

Each of these six components are specified formally with *sorts*, *operations* and *equations* for their *parameter, import* and *export* interfaces. Then the contracts are defined based on the requirement of the mine-pump system. In our framework, these contracts get specified in temporal logic. Note that the contracts which are mentioned are only from the purview of intercomponent interaction and not for the stand alone components.

Water Sensor (WS) Module: The water level should be read every δ_{w}time units, and information be sent out every δ_{wi} time units. The specification uses the variable wl, Lwl and Hwl to represent the water level, low water level and high water level.[1]

[1]Note that δ_{w}, δ_{m} , δ_{co}, δ_{af} are the period of 100ms, and δ_{wi} and $\delta_{\mathrm{mi}}$are the deadlines of 60ms.

WS-Parameter =

Sorts Real : wl,Lwl,Hwl

enum : resultw {low, high, ok}

Opns : Val-resultw : wl, Lwl, Hwl $\rightarrow$ resultw

Eqns : Val-resultw(wl, Lwl, Hwl) = resultw

WS-Import = WS-Parameter

WS-Export = WS-Parameter +

Sorts : enum : signalw {true, false}

/*The function **InformPCW** informs the pump controller about a critical water level*/

/*The function **InformLwl** informs the pump controller about a low water level*/

/*The function **AllowOp** sets a variable which determines the permission for the operator to operate the pump*/

/*The function **AllowSup** sets a variable which determines the permission for the Supervisor to operate the pump*/

/*The function **Readwl** reads the present water level*/

Opns : Readwl : $\rightarrow$ real

InformPCW, InformLwl : resultw $\rightarrow$ signalw

AllowOp, AllowSup : resultw $\rightarrow$ signalw

Eqns : InformPC(resultw) = signalw

InformLwl(resultw) = signalw

AllowOp(resultw) = signalw

AllowSup(resultw) = signalw

Readwl() = wl

Methane Sensor (MS) Module: The methane level should be read every $\delta_{\rm m}$ time units, and information be sent out every $\delta_{\rm mi}$ time units.

MS-Parameter =

Sorts : Real : ml,Cml

enum : resultm {low, high}

Opns : Val-resultm : ml, Cml $\rightarrow$ resultm

Eqns : Val-resultm(ml, Cml) = resultm

MS-Import = MS-Parameter

MS-Export = MS-Parameter +

Sorts : enum : signalm {true, false}

/*The function **Readml** reads the present methane level*/

/*The function **InformPCM** informs the pump controller about a critical methane level*/

/*The function **InformSupOpM** sets a variable which determines the permission for the operator or supervisor to operate the pump*/

Opns : Readml : $\rightarrow$ real

InformPCM : resultm $\rightarrow$ signalm

InformSupOpM : resultm $\rightarrow$ signalm

Eqns : Readml() = ml

InformPCM(resultm) = signalm

InformSupOpM(resultm) = signalm

Carbon-monoxide Sensor (COS) Module: Initially, the CO level is below the critical level. No information should be sent out by the controller. The CO level should be read every δ_{co} time units, and information be sent out every δ_{coi} time units.

COS-Parameter =

Sorts: Real : col,Ccol

enum : resultco {low, high}

Opns : Val-resultco : col, Ccol $\rightarrow$ resultco

Eqns : Val-resultco(col, Ccol) = resultco

COS-Import = COS-Parameter

COS-Export = COS-Parameter +

Sorts : enum : signalco {true, false}

/*The function **InformSupOpCO** informs the supervisor or operator about a critical CO level*/

Opns : Readcol : $\rightarrow$ real

InformSupOpCO : resultco $\rightarrow$ signalco

Eqns : Readcol() = col

InformSupOpCO(resultco) = signalco

Airflow Sensor (AFS) Module: Initially, the airflow level is below the critical level. No information should be sent out by the controller. The airflow level should be read every δ_{af} time units, and information be sent out in δ_{afi} time units.

AFS-Parameter =

Sorts : Real : afl,Cafl

enum : resultaf {low, high}

Opns : Val-resultaf : afl, Cafl $\to$ resultaf

Eqns : Val-resultaf(afl, Cafl) = resultaf

AFS-Import = AFS-Parameter

AFS-Export = AFS-Parameter +

Sorts : enum : signalaf {true, false}

/*The function **InformSupOpAF** informs the supervisor or operator about a critical Air-flow level*/

Opns : Readafl : $\to$ real

InformSupOpAF : resultaf $\to$ signalaf

Eqns : Readafl() = afl

InformSupOpAF(resultaf) = signalaf

Manual (M) Module: Initially it is assumed that neither the operator or the supervisor are giving a request for the pump or the alarm to be switched on or off. Whenever a condition for switching the pump or alarm, on or off is satisfied the request should be sent out in δ_{sai} time units.

M-Parameter =

Sorts : Int : IDUser

enum : mode {operator, supervisor, none}

Opns : Findmode : $\to$ mode

Eqns : Findmode() = IDUser

M-Import = M-Parameter

M-Export = M-Parameter

Sorts : enum : signalMM {true, false}

enum : signalA,signalPC {reqOn,reqOff,noReq}

/*The function **ReqPumpC** takes the values from signalPC(reqOn,reqOff,noReq)to signify the request sent to the pump controller */

/*The function **ReqAlarm** takes the values from signalA(reqOn,reqOff,noReq)to signify the request sent to the Alarm */

Opns : ReqPumpC : $\to$ signalPC

ReqAlarm : $\rightarrow$ signalA

Pump Controller (PC) Module: Whenever a condition for switching the pump on or off is satisfied the request should be sent out in δ_{pi} time units.

PC-Parameter =

Sorts : Int : State

enum : PState {on, off, fail}

Opns : PumpState: $\rightarrow$ PState

Eqns : PumpState() = State

PC-Import = PC-Parameter

PC-Export = PC-Parameter +

Sorts : enum : PumpReq {onReq, offReq, noReq}

/*The function **InformPump** takes one of the values from PumpReq(onReq,offReq,noReq) to signify the present request given to the pump

Opns : InformPump :$\rightarrow$ PumpReq

3.3.3 Contract Definition

The following are the contract relationships that exist between the various modules in the mine-pump system. Contracts 1 to 5 belongs to *Manual Module* and from 6 to 9 belongs to *Pump Module*.

1. The operator cannot send any request to the pump, if their exists a critical state at time T.

$$\forall t[findmode = operator, AlowOp = false \wedge t = T]$$
$$\rightarrow \Diamond(ReqPumpC = noreq \wedge T < t \leq T + \delta_{sai})$$

2. The operator can send a request only if a critical state does not exist.

$$\forall t[findmode = operator, AlowOp = true \wedge t = T]$$
$$\rightarrow \Diamond(ReqPumpC = reqOn \wedge T < t \leq T + \delta_{sai})$$

3. The supervisor can send a request only if a critical state does not exist.

$$\forall t[findmode = supervisor, InformSupOpM = false,$$
$$InformSupOpCO = false, InformSupOpAF = false \wedge t = T]$$
$$\rightarrow \Diamond(ReqPumpC = reqOn \wedge T < t \leq T + \delta_{sai})$$

4. The operator cannot send a request if a critical state exists.

$$\forall t[findmode = operator, InformSupOp = true \wedge t = T]$$
$$\rightarrow \Diamond(ReqPumpC = noReq \wedge T < t \leq T + \delta_{sai})$$

$InformSupOp = (InformSupOpM \wedge InformSupOpAF \wedge InformSupOpCO)$

5. If the methane level is high, the alarm should be switched on manually within the time δ_{sai}.

$$\forall t[signalm = True \wedge t = T]$$
$$\rightarrow \Diamond(ReqAlarm = reqOn \wedge T < t \leq T + \delta_{sai})$$

6. If the water level is critical ,the methane level is not critical and the pump is not on, then the pump should be switched on in duration δ_{pi}.

$$\forall t[InformPCW = true, InformPCM = false, State = off \wedge t = T]$$
$$\rightarrow \Diamond(InformPump = pOnReq \wedge T < t \leq T + \delta_{pi})$$

7. The pump should not operate when the methane level is critical.

   (a)

$$\forall t[InformPCW = false, InformPCM = true, State = off \wedge t = T]$$
$$\rightarrow \Diamond(InformPump = noReq \wedge T < t \leq T + \delta_{pi})$$

(b)

$$\forall t[InformPCW = false, InformPCM = true, State = on \wedge t = T]$$
$$\rightarrow \Diamond(InformPump = pOffReq \wedge T < t \leq T + \delta_{pi})$$

8. The pump should be switched off when the water level goes below the low level.

   (a)

$$\forall t[InformLwl = true, InformPCM = false, State = on \wedge t = T]$$
$$\rightarrow \Diamond(InformPump = pOffReq \wedge T < t \leq T + \delta_{pi})$$

   (b)

$$\forall t[InformLwl = false, InformPCM = false, State = on \wedge t = T]$$
$$\rightarrow \Diamond(InformPump = noReq \wedge T < t \leq T + \delta_{pi})$$

   (c)

$$\forall t[InformLwl = false, InformPCM = true, State = on \wedge t = T]$$
$$\rightarrow \Diamond(InformPump = pOffReq \wedge T < t \leq T + \delta_{pi})$$

9. The pump should respond to manual requests in time δ_{pi}.

   (a)

$$\forall t[ReqPumpC = reqOn \wedge t = T]$$
$$\rightarrow \Diamond(InformPump = pOnReq \wedge T < t \leq T + \delta_{pi})$$

   (b)

$$\forall t[ReqPumpC = reqOff \wedge t = T]$$
$$\rightarrow \Diamond(InformPump = pOffReq \wedge T < t \leq T + \delta_{pi})$$

3.3.4 Mapping Contracts to Morphism

The contracts identified will be used for defining the morphism functions which ensure the constraint imposition on the category cross product. Specifically, the morphism function does not select the value of the variables which are not identified as potential values that could have an impact on the system behavior. It should however be noted that the categories which have no contracts binding between them would follow a normal cross product because the constraining function has no parameters. We give below an example to illustrate a mapping between the contracts and the morphism function.

$$\begin{gathered}\forall t[signalm = True \wedge t = T] \\ \rightarrow \Diamond(ReqAlarm = reqOn \wedge T < t \leq T + \delta_{sai})\end{gathered}$$

This is an interaction between the methane sensor module and the manual module. The morphism function would constrain the other value of *signalm* (i.e., false) in the cross product between the two modules.

3.3.5 Test Case Generation Procedure

In this section we constructively take the constrained cross product of individual component (category), to build up a final category. The following illustrates the steps that are followed.

1. As there are no contract relationships that exists between different sensor modules, the cross product of sensor modules without constraints will produce combination of all the variables present in the composition of different sensor module called **Composite-Sensor** module. To show all the possible combination of variable sets present in the **Composite-sensor** category is not feasible (looking at the size of generated variable sets). Hence we illustrate only one of the cross product relationship of **Water Sensor** module and **Methane Sensor** module, which would give the following set of relationship.

   (a) *(InformPCW=true/false,InformPCM=true/false)*

(b) *(InformLwl=true/false,InformPCM=true/false)*

(c) *(AlowOp=true/false,InformPCM=true/false)*

2. We then take the constrained cross product of the **Composite-sensor** module with the **Manual** module. This relationship is constrained with contracts relations 1 through 5 as mentioned in Section3.3.3. We call the module obtained from the cross product as **Sen-Manual**. As explained earlier, the constraints block the unwanted combinations of test cases. For example in relation (a) mentioned below, the combination of cross product of the variables has two sets $(AllowOp = false, X)$ which is the set generated from the earlier **Composite-sensor module** and the $(findmode = operator)$, which is variable from **Manual** module. In which the value of X can be $InformPCM = true$ which says that critical methane level is high, and so inform the pump controller. Similarly, we have other sets of variables as shown below.

   (a) *((AlowOp=false,X),findmode=operator)*

   (b) *((AlowOp=true,X),findmode=operator)*

   (c) *((InformSupOpM=false,InformSupOpCO=false,InformSupOpAF=false), findmode=operator)*

   (d) *(InformSupOp=True,findmode=operator) Where InformSupOp = (InformSupOpM* $\land$ *InformSupOpAF* $\land$ *InformSupOpCO)*

   (e) *((Signalm=true,L),Y)*

   X,L,Y is used to denote all other combinations of variable values of other categories along with the variable value. For example, X can take variables such as InformSupOpM, InformSupOpCO and so on. Similarly L and Y takes different variable with respective variable values.

3. Finally we take the constrained cross product of the modules **Sen-Manual** and **Pump Controller,** constrained with contract relationships 6 through 9 as mention in Sec.3.3.3.

As explained above the variable set obtained in relation (a) as shown below has $(InformPCW = true, InformPCM = false, X)$ which is from **Sen-Manual** module and *State=off* is a variable from **Pump Controller** module. Again, here the value of X can be $AllowOp = false$, which is the variable from the water sensor module of **Composite-Sensor** module.

Similarly, we have other sets of variables which form the final constrained-cross product for the entire composite system.

(a) *((InformPCW=true,InformPCM=false,X),State=Off)*

(b) *((InformPCW=false,InformPCM=true,X),State=Off)*

(c) *((InformPCW=false,InformPCM=true,X),State=On)*

(d) *((InformLwl=true,InformPCM=false,X),State=On)*

(e) *((InformLwl=false,InformPCM=false,X),State=On)*

(f) *((InformLwl=false,InformPCM=true,X),State=On)*

(g) *((ReqPumpC=reqOn, Y),Z)*

(h) *((ReqPumpC=reqOff, Y),Z)*

X,Y is used to denote all other combinations of variable values of other categories along with the variable value.

3.3.6 Illustration of Sampurna Tool

The objective of Sampurna tool is to generate sets of test suites which are comprehensive and complete. To aid readability to the reader, the detailed illustration of all three steps of Sampurna tool is given in Appendix A.

3.3.7 Effectiveness of CAGILY Framework

The objective of the **CAGILY** framework is to obtain a complete and comprehensive set of test cases using a formal approach. One way of testing is to come up with an erroneous scenario which must be detected by set of test suite of this framework. For example if we take the external contract condition (6) from Section3.3.3, in which pump is operated only when there is a high level of water and the methane level is false. But if the critical carbon monoxide *(Ccol)* level is *true (high)* then the overall scenario changes. The erroneous value of variable *Ccol* will become true which will lead to a state in which pump will remain in *off* state despite of high water level and non critical methane level, i.e., $(Hwl = true$ and $Cml = false)$. This state has been detected by our test suite generated in Step 3(a)(iii) of Section3.3.6 and the pump will not switch to *on* state. The above mentioned example is illustrated in next chapter.

3.4 Specification and Verification using Specware

In this section, we provide the specification and verification of the properties for mine pump system. We have used Kestrel's Specware tool [Srini96] for the formal specification and verification of the case study. Specware is a refinement-based approach to software development that supports rigorous and explicit modularity in the specification and development of software components. There is also a provision for refinement of an abstract specification, primarily done by refining algorithms and data structures. We emphasize that each step in the refinement process is constrained to preserve correctness.

We start with specification of individual modules using sorts, operations and axioms. Further, we will try to compose different modules till we get a resultant composite module of all the modules. This module provides the information for water level in the mine pump.

```
WSM = spec % Water Sensor Module
sort wl = Nat sort Valresultw = Boolean
% The operation Valresult takes values of wl, Lwl and Hwl to signify
```

```
%present water level
op WLevel :  wl -> Nat
% say wl = x and value of wl > x is Hwl and wl < x is Lwl
sort signalw = Boolean
op InformPCW : signalw -> Boolean % Informs the pumpcontroller about
%High Water Level
op InformLwl :  signalw -> Boolean % Informs the Pump Controller about
%the Low Water Level
op AllowOp :  signalw -> Boolean % Assigns the permission for Operator
%to operate the Pump
op AllowSup :  signalw -> Boolean % Assigns the permission for Supervisor
%to operate the Pump
```

Having specified the parameters and operations needed for formalizing the Water Sensor Module, we now specify various attributes (properties) of the module in terms of its axioms. The axioms used in this modules specify the low water level and high water level in the tank. The axioms HWaterLevel and NoHighLow specifies effect of high water level will effect into permission denied for operator to operate the pump and when there is normal water level in the tank, there should not be any signal sent to pump controller to operate the pump respectively.

```
op Lwl :  Valresultw -> Boolean
axiom Lwl is fa (v:Valresultw) ~(Hwl(v)) & Lwl(v)
op Hwl :  Valresultw -> Boolean
axiom Hwl is fa (v:Valresultw) ~(Lwl(v)) & Hwl(v)
%When there exists High Water Level equals true then Operator should
%not operate the pump.
op HWaterLevel :  Valresultw * signalw -> Boolean
axiom HWaterLevel is fa (wh:Valresultw,ws:signalw)
Hwl(wh) & ~(AllowSup(ws))
%if there exists normal level of water in the tank, then no signal should
%be sent to pump controller to operate the pump.
op NoHighLow :  Valresultw * signalw -> Boolean
axiom NoHighLow is fa (h1:Valresultw,h2:signalw)
```

```
Hwl(h1 = false) & Lwl(h1 = false)
=> InformPCW(h2 = false) & InformLwl(h2 = false)
endspec
```

The translation of water sensor module is done to ensure that the properties i.e., the axioms can be used in other modules to prove the module properties and their theorems.

```
WSM_to_ALL_TRANSLATION = translate(WSM) by {wl +-> wl, Valresultw +->
Valresultw, signalw +-> signalw, WLevel +-> WLevel, Lwl +-> Lwl,
Hwl +-> Hwl, InformPCW +-> InformPCW, InformLwl +-> InformLwl, AllowOp
+-> AllowOp,AllowSup +-> AllowSup, HWaterLevel +-> HWaterLevel,
NoHighLow +-> NoHighLow}
```

The next module specified using the formalised sorts and operations is the Methane Sensor Module. Here the translation for water sensor module is imported and included in the this specification for future use.

```
MSM = spec % Methane Sensor Module
import WSM_to_ALL_TRANSLATION
sort Valresultm = Boolean
sort signalm = Boolean
op Cml :  Valresultm -> Boolean
op InformPCM : signalm -> Boolean
op InformSupOpM : signalm -> Boolean
```

we now specify various attributes (properties) of the module in terms of its axioms. The axiom HMLevel signifies the high methane level and a signal is send to pump controller to act accordingly.

```
%If the critical Methane level exists the signal should be sent to pump
%controller for not operating the pump
op HMLevel :  Valresultm * signalm * Valresultw -> Boolean
axiom HMLevel is fa (v1:Valresultm,s1:signalm,vv:Valresultw)
```

```
   Cml(v1 = true) & Hwl(vv)
   => InformPCM(s1 = true) & InformSupOpM(s1 = true)
   endspec
```

We need to compose the water sensor module with methane sensor module, which can be done by specifying the morphisms that links them. We formalize the morphisms between these two specifications in the following manner:

```
   WSM_to_MSM = morphism WSM -> MSM
   { HWaterLevel +-> HWaterLevel,NoHighLow +-> NoHighLow }
```

We then define the diagram with WSM and MSM specifications as the nodes, and the morphisms as the link between them.

```
   MSMm = diagram { a +-> WSM, b +-> MSM, i:  a->b +-> morphism WSM->
   MSM { HWaterLevel +-> HWaterLevel, NoHighLow +-> NoHighLow }}
```

We finally construct the composite specification of the WSM and MSM modules by taking the colimit of the diagram as shown below:

```
   MSMmm = colimit MSMm
```

The translation is done the same way as before, but this time it would have what WSM module had translated, and also the sorts, operations and properties of the MSM module.

```
   MSMmm_to_ALL_TRANSLATION = translate(MSM) by {Valresultm +-> Valresultm,
signalm +-> signalm, Cml +-> Cml, InformPCM +-> InformPCM, InformSupOpM +->
InformSupOpM, wl +-> wl, Valresultw +-> Valresultw, signalw +-> signalw,
   WLevel +-> WLevel, Lwl +-> Lwl, Hwl +-> Hwl, InformPCW +-> InformPCW,
   InformLwl +-> InformLwl, AllowOp +-> AllowOp, AllowSup +-> AllowSup,
   HWaterLevel +-> HWaterLevel, NoHighLow +-> NoHighLow }
```

Here we specify all the sorts and operations for Carbon Monoxide Sensor Module as specified in earlier two modules.

```
CMSM = spec % Carbon Monoxide Sensor Module
import MSMmm_to_ALL_TRANSLATION
sort Valresultco = Boolean
op Ccol :  Valresultco -> Boolean
sort signalco = Boolean
op InformSupOpCo :  signalco -> Boolean
```

The property for this module is specified by HCDLevel, which informs the operator / supervisor and the controller about the critical level of carbon monoxide level.

```
op HCDLevel :  Valresultco * signalco * Valresultw -> Boolean
axiom HCDLevel is fa (va1:Valresultco,si1:signalco,vv1:Valresultw)
Ccol(va1 = true) & Hwl(vv1) => InformSupOpCo(si1 = true)
endspec
```

Here the composition of methane sensor module and carbon monoxide module is done similar to the way explained before by specifying the morphism that links between them. And finally constructing the composite specification of MSM and CMSM, where MSM already includes specification for WSM, by taking the colimit of the diagram. Further, we need to translate this composed specification with sorts and operations of all the previous modules.

```
MSM_to_CMSM = morphism MSM ->
CMSM {HMLevel +-> HMLevel}
CMSMc = diagram { a +-> MSM, b +-> CMSM, i:  a->b +-> morphism MSM->
CMSM {HMLevel +-> HMLevel}}
CMSMcc= colimit CMSMc
CMSMcc_to_ALL_TRANSLATION = translate(CMSM) by {Valresultm +->
Valresultm, signalm +-> signalm, Cml +-> Cml, InformPCM +-> InformPCM,
InformSupOpM +-> InformSupOpM, wl +-> wl, Valresultw +-> Valresultw,
signalw +-> signalw, WLevel +-> WLevel, Lwl +-> Lwl, Hwl +-> Hwl,
InformPCW +-> InformPCW, InformLwl +-> InformLwl, AllowOp +-> AllowOp,
AllowSup +-> AllowSup, HWaterLevel +-> HWaterLevel, NoHighLow +->
NoHighLow, Valresultco +-> Valresultco, Ccol +-> Ccol, InformSupOpCo +->
InformSupOpCo, HCDLevel +-> HCDLevel}
```

The specification for Air Flow Sensor Module is given below with their respective sorts and operations defined. The translation of carbon monoxide module is imported in this module to include all the modules specified earlier.

```
ASM = spec % Air Flow Sensor Module
import CMSMcc_to_ALL_TRANSLATION
sort Valresultaf = Boolean
op Cafl :  Valresultaf -> Boolean
sort signalaf = Boolean
op InformSupOpAf :  signalaf -> Boolean
```

The axiom for air flow module HCAfl signfies critical air flow level in the mine pump and transmits signal to operator/Supervisor and controller to act accordingly.

```
op HCAfl :  Valresultaf * signalaf * Valresultw -> Boolean
axiom HCAfl is fa (a1:Valresultaf,a2:signalaf,vv2:Valresultw)
Cafl(a1 = true) & Hwl(vv2) => InformSupOpAf(a2 = true)
endspec
```

Then we will compose the air flow sensor module with the carbon monoxide moduel which is already composed with other modules. This composition is done by specifying the morphisms linked between the modules. Further, we define the diagram CMSM and ASM specifications as the node and morphisms as the link between them. Then the final composite module is generated using the colimit of the diagram defined earlier. Finally, the translation of this composite module is done with sorts and operations of this module aswell as all the previous modules.

```
   CMSM_to_ASM = morphism CMSM -> ASM {HCDLevel +-> HCDLevel}
   ASMs = diagram { a +-> CMSM, b +-> ASM, i:  a->b +-> morphism
   CMSM->ASM{HCDLevel +-> HCDLevel}}
   ASMss = colimit ASMs
ASMss_to_ALL_TRANSLATION = translate(ASM) by {Valresultm +-> Valresultm,
   signalm +-> signalm, Cml +-> Cml, InformPCM +-> InformPCM, InformSupOpM
   +-> InformSupOpM, wl +-> wl, Valresultw +-> Valresultw, signalw +->
   signalw, WLevel +-> WLevel, Lwl +-> Lwl, Hwl +-> Hwl, InformPCW +->
   InformPCW, InformLwl +-> InformLwl, AllowOp +->
   AllowOp, AllowSup +-> AllowSup, HWaterLevel +->
   HWaterLevel, NoHighLow +-> NoHighLow, Valresultco +->
   Valresultco, Ccol +-> Ccol, InformSupOpCo +-> InformSupOpCo, HCDLevel
   +-> HCDLevel, Valresultaf +-> Valresultaf, Cafl +-> Cafl, signalaf +->
   signalaf, InformSupOpAf +-> InformSupOpAf, HCAfl +-> HCAfl}
```

The manual module is specified with different sorts and operations along with the importing the air flow sensor composite module as shown below:

```
   MM = spec % Manual Module
   import ASMss_to_ALL_TRANSLATION
   sort Findmode = Boolean
   op Operator :  Findmode -> Boolean
   op Supervisor :  Findmode -> Boolean
   sort signalMM = Boolean
   sort signalA = Nat % 1 = reqOn, 2 = reqOff, 3 = noReq
   sort signalPC = Nat % 1 = reqOn, 2 = reqOff, 3 = noReq
   op ReqPumpC : signalPC * signalMM * Nat -> Boolean
   op ReqAlarm :  signalA * signalMM * Nat -> Boolean
```

The axioms for manual module are SendRequest and NoRequest. For Send Request, for any operator or supervisor, if any sensor value is not critical then the operator can put the pump in *ON* state . Similarly, when the critical level is high, the operator or supervisor cannot send any request to put the pump in *ON* state.

```
%Operator can send a request only if a critical state does not exists
op SendRequest :  Findmode * signalaf * signalco * signalm * signalMM *
signalPC * Valresultw -> Boolean
axiom SendRequest is fa(f:Findmode,af:signalaf,co:signalco,m:signalm,
mm:signalMM,pc:signalPC,vvm:Valresultw)
Operator(f) or Supervisor(f) & InformSupOpAf(af=false) &
InformSupOpCo(co=false) & InformSupOpM(m=false) & Hwl(vvm)
=> ReqPumpC(pc,mm,1)
%Operator cannot send a request if a critical state exists
op NoRequest :  Findmode * signalaf * signalco * signalm * signalMM *
signalPC * Valresultw -> Boolean
axiom NoRequest is fa (f1:Findmode,af1:signalaf,co1:signalco,m1:signalm,
mm1:signalMM,pc1:signalPC,vvm1:Valresultw)
Operator(f1=true) & InformSupOpAf(af1=true) or InformSupOpCo(co1=true)
or InformSupOpM(m1=true) & Hwl(vvm1) => ReqPumpC(pc1,mm1,3)
endspec
```

We will compose the manual module with air flow sensor module which itself is composed with other sensor modules. After composing the specification using the morphisms between the two modules, we will define the diagram between these modules to construct the composite manual module by taking the colimit of the diagram.

```
ASM_to_MM = morphism ASM -> MM {HCAf1 +-> HCAf1}
MMm = diagram { a +-> ASM, b +-> MM, i:  a->b +-> morphism ASM-> MM
{HCAf1 +-> HCAf1}}
MMms = colimit MMm
```

The translation of the composite manual module is done as shown below:

```
MMms_to_ALL_TRANSLATION = translate(MM) by {Findmode +-> Findmode,
Operator +-> Operator, Supervisor +-> Supervisor, signalMM +-> signalMM,
signalA +-> signalA, signalPC +-> signalPC, ReqPumpC +-> ReqPumpC,
ReqAlarm +-> ReqAlarm, SendRequest +-> SendRequest, NoRequest +->
NoRequest,Valresultm +-> Valresultm, signalm +-> signalm, Cml +-> Cml,
InformPCM +-> InformPCM, InformSupOpM +-> InformSupOpM, wl +-> wl,
Valresultw +-> Valresultw, signalw +-> signalw, WLevel +-> WLevel,
Lwl +-> Lwl, Hwl +-> Hwl, InformPCW +-> InformPCW, InformLwl +-> InformLwl,
AllowOp +-> AllowOp, AllowSup +-> AllowSup, HWaterLevel +-> HWaterLevel,
NoHighLow +-> NoHighLow, Valresultco +-> Valresultco, Ccol +-> Ccol,
InformSupOpCo +-> InformSupOpCo, HCDLevel +-> HCDLevel, Valresultaf +->
Valresultaf, Cafl +-> Cafl, signalaf +-> signalaf, InformSupOpAf +->
InformSupOpAf, HCAfl +-> HCAfl}
```

The final specification of pump controller module is done as shown below. This is done by specifying the different sorts and operations along with importing the composite manual module.

```
PCM = spec % Pump Controller Module
import MMms_to_ALL_TRANSLATION
sort PState = Nat % 1 = on, 2 = off, 3 = fail
op State :  PState * Nat -> Boolean
sort PumpReq = Nat % 1 = pOnReq, 2 = pOffReq, 3 = noReq
```

This operation takes place when the controller needs to send OnRequest/OffRequest/noRequest to the pump.

```
op InformPump :  PumpReq * Nat -> Boolean
```

This action of operation PumpOnState takes place the controller receives high water level with all other critical value of sensors as normal, it sends request to put the pump into *ON* state.

```
% If the water level is critical, the methane level is not critical and
%the pump is not on the pump should be switched on
op PumpOnState :  signalw * signalm * PState * PumpReq *
Valresultw -> Boolean
axiom PumpOnState is fa (sw:signalw,sm:signalm,ps:PState,pr:PumpReq,
vvp1:Valresultw)
InformPCW(sw=true) & InformPCM(sm=false)& State(ps,2) & Hwl(vvp1) =>
InformPump(pr,1)
```

The operation PumpOff results when the pump controller receives critical value of sensor values then the controller sends a signal to put the pump into *Off* state.

```
% The Pump should not operate when the methane level is critical
op PumpOff :  Valresultm * Valresultco * Valresultaf * PumpReq -> Boolean
axiom PumpOff is fa(vam:Valresultm,vac:Valresultco,vaa:Valresultaf,
pur:PumpReq)
Cml(vam) or Ccol(vac) or Cafl(vaa) => InformPump(pur,2)
endspec
```

The morphisms between the composite manual module and pump controller module is formally specified by composition as shown below:

```
MM_to_PCM = morphism MM ->
PCM {SendRequest +-> SendRequest, NoRequest +-> NoRequest}
```

We define the diagram with MM and PCM specifications as the nodes and the corresponding morphisms as the links between them. This is formally represented as:

```
PCMm = diagram { a +-> MM, b +-> PCM, i:  a->b +-> morphism MM->
PCM{SendRequest +-> SendRequest, NoRequest +-> NoRequest}}
```

We finally construct the composite specification of the MM and PCM modules by taking the colimit of the diagram, which is specified as:

```
PCMmm = colimit PCMm
```

The translation provided by the composed module which will be used by other modules for their operation is shown below:

```
PCMmm_to_ALL_TRANSLATION = translate(PCM) by {Findmode +-> Findmode,
signalA +-> signalA,signalMM +-> signalMM,signalPC +-> signalPC,
Operator +-> Operator,Supervisor +-> Supervisor,ReqPumpC +-> ReqPumpC,
ReqAlarm +-> ReqAlarm,SendRequest +-> SendRequest,NoRequest +-> NoRequest}


PPCM = spec
import PCMmm
% The Pump should be switched off when the water level goes below
%the low level or the pump should operate accordingly when the critical
%state exists
```

The final theorem that is to be proved from the composed specification is given below:

```
theorem OnOffPump_for_CriticalState is fa
(vw2:Valresultw,vm2:Valresultm,vco2:Valresultco,vaf2:Valresultaf,
sw2:signalw,sm2:signalm,ps2:PState,pr2:PumpReq,co2:signalco,af2:signalaf,
mm2:signalMM,pc2:signalPC,fm2:Findmode)
HMLevel(vm2,sm2,vw2) or HCDLevel(vco2,co2,vw2) or HCAfl(vaf2,af2,vw2) or
Hwl(vw2) => ~(Lwl(vw2)) => NoRequest(fm2,af2,co2,sm2,mm2,pc2,vw2)
=> PumpOff(vm2,vco2,vaf2,pr2)
endspec
```

The theorem, OnOffPump_for_CriticalState shows the final condition of the pump controller to make the pump to On or Off state, which states that when their exists high level of methane level or high level of carbon monoxide level or high level of air flow level and either the water level is high or low then this leads to low water level low or high respectively, depending of upon the Hwl(high water level). This further implies to operation NoRequest of the manual module in which no request can be sent to the pump controller under this critical condition. Finally this leads to the condition in pump controller modules, were the operation PumpOff will send a signal to put the Pump in the *Off* state. We finally verify the above mentioned property by processing the above specification along with the theorem in Specware with a built-in interface to Snark theorem prover. The statement for the proof of this property is given below:

```
p1 = prove OnOffPump_for_CriticalState in PPCM using Hwl, Lwl, PumpOff
```

Till now in this chapter, we discussed the CAGILY framework and illustrated the approach using the classical mine pump problem. We have shown the formal specification and verification of the composed system and showed the correctness of the composed system using Specware tool. In next chapter, we discuss the implementation of Sampurna testing tool and will illustrate how faulty condition can be tested using it.

Chapter 4

Implementation of Sampurna Tool

In this chapter, we provide a brief overview of the CAGILY framework discussed earlier. We further discuss the need for implementing the proposed sampurna tool and why the MS-Access database was selected to generate the test suites. Next we, explain the generation of the database and its design and implementation efforts required for generation of test suites, and finally discuss, the effectiveness of the sampurna tool by providing suitable example.

The objective of functional test case generation has been studied in great deal in the past. The basic requirement of all testing processes involves a complete set of test suite which explores all requirements which the system has to satisfy. However, the completeness of test suite cannot be determined easily.

The approach proposed for generating test suite in the thesis, involves a careful observation of all the possible combinations that can exist in the system and based on that a test suite is generated. The CAGILY framework is presently generating test cases for critical functional properties only , however since the generation of the test cases is driven by the set of contracts for critical behavior this can be extrapolated to cover all functional properties.

The test generation is based on characterization of the contracts, which are generated directly from the system requirements. These contracts are relationships between various variables in the components. These components are identified formally from the system decomposition. The components are comprised of the system, and their variables are actually the system variables which control the behavior of the system.

The constrained cross product amongst the component variables limits the size of the actual set of relationship which is obtained. These relationships would finally comprise the test suite after a few rounds of iterations to filter out redundant test cases. This is done to have a small size of the test suite and also maintaining the completeness of the test suite.

The test case generation procedure processes the large set of test cases obtained in the constrained cross product to the SAMPURNA test generation tool, which through a few rounds of iterations filters out the unwanted/redundant test cases so as to achieve the final test suite.

The following are the steps in the Test Case Generation Procedure:

1. The Constraint based cross product is taken between the two modules or categories. If the modules dont have any external contracts then normal cross product is followed. The purpose of constraining the cross product is to come up with the set of test cases which are relevant to the functional specification of the system. These constraint cross product is done till we get single resultant category which is a combination of all the categories (modules).

2. The next step is the introduction of Sampurna test tool. This tool can further be subdivided into three different sections depending upon their functional classification.

   (a) Once we arrive at the sets of variables, which form the final constrained-cross product for the entire composite system. The first step in Sampurna tool is to bring (import) the relationship chain (sets of variables associated) to the final sets of variable considering all the possible combination of values/range for which the variables may attain. This ensures that we arrive at the end values which are the actual inputs to the system.

(b) After constraining the cross product between the categories, we arrive at a smaller list of relationships which are still having some sets of test cases which are meaningless and redundant. There is a need to reduce this set of test cases so as to optimize set of test cases which will satisfy the functional aspects of the system. This is achieved by using a priori knowledge of the system such that the variable-value pairs that are not attainable/possible with respect to the system specification are not included in the resulting test cases.

(c) The final step in Sampurna tool is to parse the set of test cases with respect to contracts so as to obtain the expected output of the system. This associated values/range of the variables will steer the system through different state and will help in finding any discrepancies as compared to the correct behavior of the system. The final resultant test suite which we get after this iteration is a complete set of test cases which targets the critical functional testing of the system.

The steps (a),(b) and (c) needs some form of iteration process. And looking to the framework proposed in this thesis, the want for generating a program for Sampurna Tool demanded a efficient tool which will generate the required set of test cases. We explored different approaches, finally deciding on using MS-Access database application for generating test suites.

4.1 Database Tool for Generating Test Suites

The important reason for using MS-Access for generating the test suites for our proposed approach was to come up with exact set of test cases which will test the functional behavior of the system under test. And the database tool provides flexibility and ease for generating these test suites efficiently. Here we assume that we have a set of variables which are already defined a priori depending upon the system specification. This variables are stored in different tables depending upon there functionalities. And finally queries are posed to retrieve the desired set of test cases.

The first step towards generating the Sampurna Tool is to have a detailed study of the requirements and depending upon these requirements, we need to design the layout of the tool. We started with designing the layout for different tables depending upon the components of the system. The next step is to design the SQL queries, which takes the tables defined in previous step as its input and using some logical functions generate set of test cases iteratively. There are more than one query used to generate the desired set of test cases. The final output of this queries can be stored in the table and reports can be generated, which can be used by a tester or user of the system.

4.2 Tables for Component Modules

The tables are generated or assigned statically depending upon the modules or components identified earlier in our proposed approach. Here we start with generating six static tables in which particularly the variables defined earlier will be stored. The following tables are shown with their fields and variables.

This table stores the values of variables which fall into the module and which are already defined in the system specification. The graphical presentation of the table is shown below :-

WaterSensor
InformLWL=False
InformLWL=True
InformPCW=False
InformPCW=True

Table 4.2: WaterSensorModule

Manual
ReqPumpC=NoReq
ReqPumpC=ReqOff
ReqPumpC=ReqOn

Table 4.4: ManualModule

Carbondioxide
InfoSupCo=False
InfoSupCo=True

Table 4.6: CarbondioxideModule

AirFlowSensor
InfoSupAf=False
InfoSupAf=True

Table 4.8: AirFlowSensorModule

PumpController
State=Off
State=On

Table 4.10: PumpControllerModule

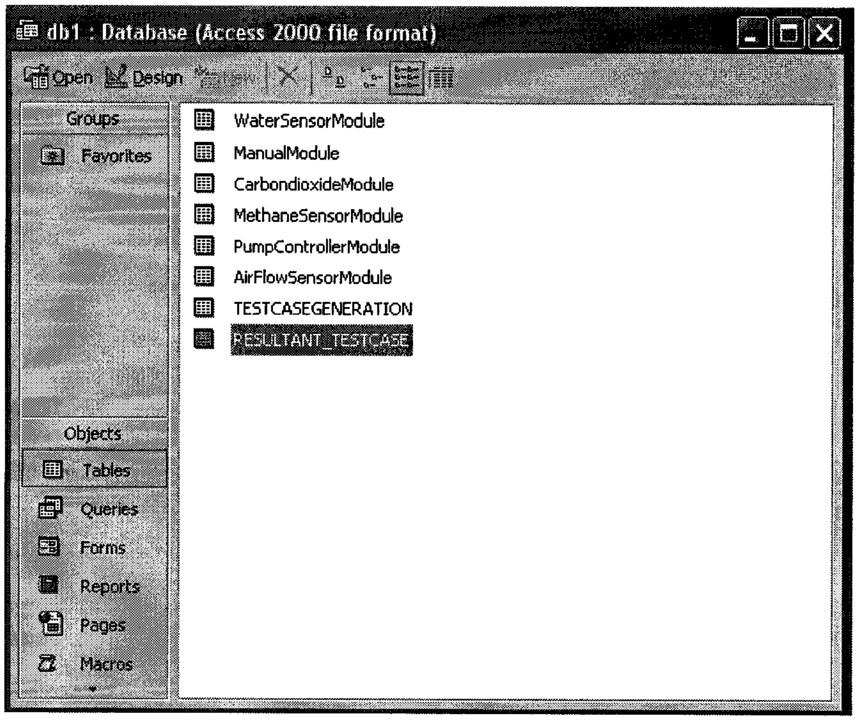
This constitutes the static tables which form the basic platform to construct the desired test cases. The following are the tables which are generated when the above tables are queried using certain constraints depending upon the system specification.

AirFlowSensor	Carbondioxide	Manual	MethaneSensor	PumpController	WaterSensor

Table 4.11: TESTCASEGENERATION

AirFlowSensor	Carbondioxide	Manual	MethaneSensor	PumpController	WaterSensor	OutPut

Table 4.12: RESULTANT_TESTCASE



WaterSensorModule
ManualModule
CarbondioxideModule
MethaneSensorModule
PumpControllerModule
AirFlowSensorModule
TESTCASEGENERATION
RESULTANT_TESTCASE

Figure 4.1: Sampurna Tool - Tables

The TESTCASEGENERATION Table4.11 acts as a static table and stores all the possible set of test cases which are generated using the constraint based cross product between different modules. The RESULTANT_TESTCASE Table4.12 comes into picture when the unattainable and redundant test cases are removed from the resultant test cases. And finally the expected output is generated and stored in the field *OutPut* column of this table. So this forms the final and resultant set of test cases which we get at the end of our implementation. The overall graphical representation of the tables in Sampurna tool is shown in Figure4.1.

4.3 Queries for Generating Test Suites

Queries form the important part of test case generation process. Depending upon how the queries are posed, the test cases are generated. The primary aim of these queries is to take all the possible combination of test cases which the system could generate and then narrow down the test cases to attain the resultant test suites by eliminating the redundant and un-attainable test cases from the set. Hence we will discuss the different queries used for the generation of test suites. Graphical representation of the Queries in Sampurna Tool is shown in Figure4.2.

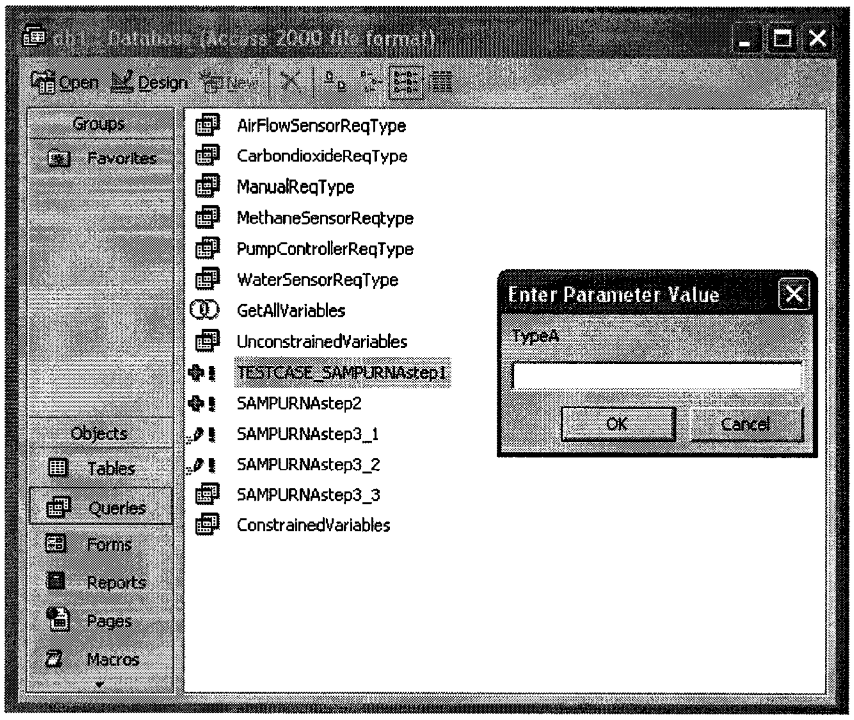


Figure 4.2: Sampurna Tool - Queries

4.3.1 GetAllVariables

The query performs a union operation on all the tables for each component. This process makes sure that all the variable values are covered during the test case generation process.

SELECT AirFlowSensor AS DifferentTypes

FROM AirFlowSensorModule

UNION (SELECT WaterSensor AS DifferentTypes FROM WaterSensorModule)

UNION (SELECT MethaneSensor AS DifferentTypes FROM MethaneSensorModule)

```
UNION (SELECT Carbondioxide AS DifferentTypes FROM CarbondioxideModule)
UNION (SELECT Manual AS DifferentTypes FROM ManualModule)
UNION (SELECT PumpController AS DifferentTypes FROMpumpControllerModule);
```

4.3.2 ConstrainedVariables

Here all the constrained variables are gathered from the user and accordingly, the variables are then processed to get the constraint-cross product of the variables along with unconstrained one. The following is the SQL Query used to generate the constrained variables.

```
SELECT DifferentTypes
FROM GetAllVariables
WHERE DifferentTypes=TypeA Or DifferentTypes=TypeB Or DifferentTypes=TypeC;
```

4.3.3 UnconstrainedVariables

The output of this query is generated using the previous GetAllVariables and ConstrainedVariables queries. It consists of the variables which are unconstrained. The following is the SQL Query used for generating the unconstrained variables.

```
SELECT GetAllVariables.DifferentTypes
FROM GetAllVariables
WHERE GetAllVariables.DifferentTypes<>[TypeA]
And GetAllVariables.DifferentTypes<>[TypeB]
And GetAllVariables.DifferentTypes<>[TypeC];
```

4.3.4 Sensors And Controller Module

The query will find out if there are any constrained variables for the respective module. Depending upon the query, it will take only the constrained variables for generating the test suites, else it will use all the unconstrained variables for the generation of test suite.

4.3.4.1 AirFlowSensorReqType

```
SELECT *
FROM AirFlowSensorModule
WHERE AirFlowSensor Not In
    (SELECT AirFlowSensor
    FROM AirFlowSensorModule, UnconstrainedVariables
    WHERE AirFlowSensor=DifferentTypes
    And Exists (select * from AirFlowSensorModule, ConstrainedVariables
    where AirFlowSensor = DifferentTypes));
```

4.3.4.2 CarbondioxideReqType

```
SELECT *
FROM CarbondioxideModule
WHERE Carbondioxide Not In
    (SELECT Carbondioxide
    FROM CarbondioxideModule,UnconstrainedVariables
    WHERE Carbondioxide=DifferentTypes
    And Exists (select * from CarbondioxideModule, ConstrainedVariables
    where Carbondioxide = DifferentTypes));
```

4.3.4.3 ManualReqType

```
SELECT *
FROM ManualModule
WHERE Manual Not in
    (SELECT Manual
    FROM ManualModule, UnconstrainedVariables
    WHERE Manual=DifferentTypes And Exists
    (select * from ManualModule, ConstrainedVariables where Manual = DifferentTypes));
```

4.3.4.4 MethaneSensorReqType

```
SELECT *
FROM MethaneSensorModule
WHERE MethaneSensor Not in
    (SELECT MethaneSensor
    FROM MethaneSensorModule, UnconstrainedVariables
    WHERE MethaneSensor=DifferentType
    And Exists (select * from MethaneSensorModule, ConstrainedVariables
    where Methanesensor = DifferentTypes));
```

4.3.4.5 PumpControllerReqType

```
SELECT *
FROM PumpControllerModule
WHERE PumpController Not in
    (SELECT PumpController FROM PumpControllerModule, UnconstrainedVariables
    WHERE PumpController=DifferentTypes
    And Exists (select * from PumpControllerModule, ConstrainedVariables
    where PumpController = DifferentTypes));
```

4.3.4.6 WaterSensorReqType

```
SELECT *
FROM WaterSensorModule
WHERE WaterSensor Not In
    (SELECT WaterSensor
    FROM WaterSensorModule, UnconstrainedVariables
```

WHERE WaterSensor=DifferentTypes

*And Exists (select **

from UnconstrainedVariables:- WaterSensorModule, ConstrainedVariables

where Watersensor = DifferentTypes));

4.4 Action Queries

These are the queries which play important role in test case generation. They try to update existing tables or insert some of the fields in existing fields. These queries take all the queries discussed in Section 4.3 and Tables 4.2, 4.4, 4.6, 4.8 and 4.10 as their inputs and insert or update the tables according to the conditions imposed in them.

4.4.1 Sampurna_Step1

It forms a base query for taking the inputs from the tester. It can take three different constraint variable form the tester and then using different combination of queries embeded in it, generates a set of test cases which are stored in a Table 4.11. Mathematically the query does the cross product of constrained and unconstrained variables to generate a resultant test case, the variables of this resultant test case are stored in the existing Table 4.11. This includes all the possible combination which the test case can attain. As there are no variables functions involved in this process, during implementation we have merged the first step of Sampurna Tool with the test case generation process.

INSERT INTO TESTCASEGENERATION

*SELECT DISTINCT **

FROM AirFlowSensorReqType, CarbondioxideReqType, ManualReqType, MethaneSensorReqType, PumpControllerReqType, WaterSensorReqType;

4.4.2 Sampurna_Step2

The query will try to remove the unattainable combination of variables. Like for example certain variable combinations of variables cannot co-exist at a same time in a test case, such test cases are removed in this step.

```
INSERT INTO RESULTANT_TESTCASE
SELECT *
FROM TESTCASEGENERATION
WHERE (((TESTCASEGENERATIONUnconstrainedVariables:-
.Manual)="ReqPumpC=ReqOff")
AND ((TESTCASEGENERATION.PumpController)<>"State=Off"))
OR (((TESTCASEGENERATION.Manual)<>"ReqPumpC=ReqOff")
AND ((TESTCASEGENERATION.PumpController)="State=Off"))
OR (((TESTCASEGENERATION.Manual)="ReqPumpC=ReqOn")
AND ((TESTCASEGENERATION.PumpController)<>"State=On"))
OR (((TESTCASEGENERATION.Manual)<>"ReqPumpC=ReqOn")
AND ((TESTCASEGENERATION.PumpController)="State=On"))
OR (((TESTCASEGENERATION.Manual)<>"ReqPumpC=ReqOn"
And (TESTCASEGENERATION.Manual)<>"ReqPumpC=ReqOff")
AND ((TESTCASEGENERATION.UnconstrainedVariables:-
PumpController)<>"State=On"
And (TESTCASEGENERATION.PumpController)<>"State=Off"));
```

4.4.3 Sampurna_Step3_1

This query is based on Sampurna tool step 3. It forms the first iteration of the step3. The posed query will try to check the constrained variables and then will set the output of the pump to $PState = On$. This is done, depending upon the conditions mentioned in the system specification for making the pump in *On* state.

```
UPDATE RESULTANT_TESTCASE
SET [OutPut] = "PState=Off"
WHERE ((RESULTANT_TESTCASE.AirFlowSensor="InfoSupAf=True"
OR RESULTANT_TESTCASE.Carbondioxide="InfoSupCo=true"
OR RESULTANT_TESTCASE.MethaneSensor="InformPCM=true"
OR RESULTANT_TESTCASE.WaterSensor="InformLWL=True")
OR (RESULTANT_TESTCASE.WaterSensor<>"InformPCW=True"
AND RESULTANT_TESTCASE.Manual="ReqPumpC=ReqOff"))
OR OutPut IN (SELECT OutPut
FROM RESULTANT_TESTCASE WHERE OutPut<>"PState=On");
```

4.4.4 Sampurna_Step3_2

This forms step 3 of Sampurna tool but it falls in the second iteration of the step 3 of the tool. The query checks the variable values and depending upon the constraints gathered from the system specification, generates the desired output for $PState = Off$.

```
UPDATE RESULTANT_TESTCASE
SET [OutPut] = "PState=On"
WHERE ((RESULTANT_TESTCASE.AirFlowSensor<>"InfoSupAf=True"
AND RESULTANT_TESTCASE.Carbondioxide<>"InfoSupCo=true"
AND RESULTANT_TESTCASE.MethaneSensor<>"InformPCM=true")
AND RESULTANT_TESTCASE.WaterSensor<>"InformLWL=True"
AND ((RESULTANT_TESTCASE.Manual<>"ReqPumpC=ReqOff"
OR (RESULTANT_TESTCASE.WaterSensor="InformLWL=False"
AND RESULTANT_TESTCASE.WaterSensor="InformPCW=False"))
OR RESULTANT_TESTCASE.WaterSensor="InformPCW=True"))
OR OutPut IN (SELECT OutPut
FROM RESULTANT_TESTCASE WHERE OutPut<>"PState=Off");
```

4.4.5 Sampurna_Step3_3

In this iteration the test cases which are still redundant and missed out in the earlier step are removed by parsing the test cases with the constraints. The resultant test cases generated forms the test suites for testing the critical functionality of the system. This forms the final iteration of step3 of Sampurna Tool.

```
SELECT RESULTANT_TESTCASE.AirFlowSensor,
RESULTANT_TESTCASE.Carbondioxide,
RESULTANT_TESTCASE.Manual,
RESULTANT_TESTCASE.MethaneSensor,
RESULTANT_TESTCASE.PumpController,
RESULTANT_TESTCASE.WaterSensor,
RESULTANT_TESTCASE.OutPut
FROM RESULTANT_TESTCASE
GROUP BY RESULTANT_TESTCASE.AirFlowSensor,
RESULTANT_TESTCASE.Carbondioxide,
RESULTANT_TESTCASE.Manual,
RESULTANT_TESTCASE.MethaneSensor,
RESULTANT_TESTCASE.PumpController,
RESULTANT_TESTCASE.WaterSensor,
RESULTANT_TESTCASE.OutPut;
```

The following Table 4.13 illustrates the number of test cases generated at the end of Step1 of Sampurna tool. Here we have shown sample of test cases generated, the full set of test cases are attached in Appendix B

AirFlowSensor	Carbondioxide	Manual	MethaneSensor	PumpController	WaterSensor
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True
InforSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=False
InfoSupAF=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False
InfoSupAF=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=False

Table 4.13: TESTCASE

At the end of Step 2 and 3, we eliminate the redundant test cases and generate the output value for the test cases by parsing the constraints imposed by system specification. The values obtained at the end of Step3 are the expected values which the system should satisfy. The following is the Table (See Table 4.14) illustrating the final set of test cases after removing redundant data. This is the sample taken from the full set of test suites generated at the end of final round of iteration of Sampurna tool which are shown in Appendix C.

AirFlowSensor	Carbondioxide	Manual	MethaneSensor	PController	WaterSensor	OutPut
InfoSupAf=False	**InfoSupCo=False**	**ReqPumpC=NoReq**	**InformPCM=False**	**State=Off**	**InformPCW=True**	**PState=On**
InfoSupAf=False	**InfoSupCo=False**	**ReqPumpC=ReqOn**	**InformPCM=False**	**State=Off**	**InformPCW=True**	**PState=On**
InfoSupAf=False	***InfoSupCo=True***	***ReqPumpC=NoReq***	***InformPCM=False***	***State=Off***	***InformPCW=True***	***PState=Off***
InfoSupAf=False	***InfoSupCo=True***	***ReqPumpC=ReqOn***	***InformPCM=False***	***State=Off***	***InformPCW=True***	***PState=Off***
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=True	PState=Off

Table 4.14: RESULTANT_TESTSUITE

4.5 Applying Resulting Test Suites - An Example

The user or the tester can use this test cases as illustrated below:

- The only parameters the tester has to see initially are the parameters in *Manual* and *PumpController (PController)* column(See Table 4.14). Where *PController* variable is showing the present state of the pump, while *Manual* variable is showing the values of the variables by which the the Operator/Supervisor can change the state of the pump manually.

- Now in this step, depending upon the values of all the sensor values, like the *AirFlowSensor, Carbondioxide Sensor, Methane Sensor and Water Sensor*, the tester has to check the expected behaviour by observing values given in the *OutPut* variable column(See Table 4.14).

- So if the expected behaviour of the pump does not reflect the pump condition properly, then the tester can infere that their are some failures or malfunctioning of some sensors or other equipments.

4.6 Effectiveness of Implementated Sampurna Tool

As discussed in Section 3.3.7, depending upon the contract condition (6) when the there is high level of water and methane level is low, the pump should go in *On* state. But as stated earlier, if the carbon monoxide level goes critical then this condition is detected by our Sampurna tool shown in Table 4.14 (with highlighted rows - 3rd and 4th row specifically), and the Pump instead of going into *On* state goes in *Off* state. If the Pump does not turn *Off* during say time T then we can infer that there is some failure in the system. And immediate action is needed to be taken to avoid any incidents. Also when the the carbon dioxide level retains its normal value, the pump might not go into the *On* state which in our case can be detected by the the Sampurna tool. This is shown by first two rows of Table 4.14, which are marked in bold letters(See rows 1^{st} and 2^{nd} in Table4.14). Similarly, we can detect different faulty scenarios using the testing tool.

The only drawback of the proposed tool is that, the system under test, requires component identification by decomposing the system hierarchically. This decomposition requires domain knowledge and tends to be somewhat complex as the present day systems span across various domains. In other words if the system can be decomposed and identified into their primitive components then we can use the Sampurna tool for generation of test suites. It is important to mention that the Sampurna tool process involved in building the test suites are generic for specific class of system. The implementation of Sampurna tool suggested in this chapter using MS-Access does involve some sort of hard coding, but these can changed to other options by providing inputs from the user. This can be achieved by parsing the specification file generated from the Specware tool and generating a file format readable by our MS-Access or any other database.

Chapter 5

Conclusion

This thesis aims at the generation of test suites for component-based system using category theoretic approach. The generation of test suits is based on intra-component interaction between the components. Further, the test case generation process is based on characterization of the contracts, which are generated directly from the system requirements. The CAGILY framework is used to generate test suites for critical functional properties of the system. However since the generation of the test cases are driven by the set of contracts, the framework can be extrapolated to cover all the functional properties.

The reason for targeting the functional area of testing is mainly because, we all know that it is impossible to do exhaustive testing. This is restricted by different factors such as location, technical point of view, managerial point of view and amount of time invested in testing the system. So, it is meaningful to have some functional coverage measurement criteria to ensure that no function is missing in an implementation. Based on this requirements, we proposed a framework which generates a set of test cases to test the critical functional aspect of the system.

In this chapter, we first discuss some of the insights obtained from the generation of test cases for component based system using category theoretic approach in terms of experience gained through the proposed approach and proposing the Sampurna testing tool followed by the contributions of this thesis. Finally, we put forth new directions for further research.

5.1 Experience

Our aim in this thesis has been to integrate component-based design of complex systems and constraint-based test generation in to a unified framework. Concepts of category theory provide a unique approach for defining a module structure and operations for composing modules together. Specifically, we proposed the CAGILY (CAteGory framework for Identifying test cases formalLY) framework for generation of test suites. In this thesis, we have shown how concept of constraint-based cross product in category theory and system contracts play important role in generation of proposed approach.

The major obstacle which we overcame was related to the process of identifying the various primitive components of the system and also specifying this components formally with sorts, operations and equations for their parameter, import and export interfaces. We have outlined five steps for generating the set of test cases using this framework. This steps are generic in nature as far as the system under test can be decomposed into there primative components. We have shown that how set of constraints (contracts) can be mapped to morphism function, which further can be used for generation of comprehensive set of test cases using constraint cross-product. In the test case generation procedure, we arrive at comprehensive set of test cases. Here we have proposed a test case generation tool - Sampurna tool. Which removes the unattainable and redundant set of test cases which gets generated in first round of iteration. We have used classical mine pump problem for our case study. Using this case study, we show the feasibility and effectiveness of our approach and have shown, how some errenous condition can be detected by our tool. To generate a component specification of the system, firstly we need to specify the system using some specification language. We have shown the formal specification for the mine pump, which has been written in an algebraic language, namely, MetaSlang. We have made full use of the Specware development system, which provides for a rigorness in establishing the overall correctness of the composition, and moreover, facilitates further transformation into executable code.

5.2 Contributions

In this thesis, we have shown the generation of test cases for critical functional areas of the system. In Chapter 2, we have shown the concepts of category theory used for composition of modules (components). We also discuss some aspects of Specware tool used to formally specify the system composition. In Chapter 3, we proposed our CAGILY framework for generation of test suites using category theory. We first identify the primitive component of the system and then specify them formally. We then try to define the set of constraints from the system specification, which are further mapped to morphism functions. We then show how the constraint-based cross product can be utilized for generation of comprehensive set of test cases. We, further provide the effectiveness of the proposed framework, illustrating the faulty scenario. In this chapter, we also specify the composition of different modules using Specware tool. This helps us in proving the correctness of the composition. In Chapter 4, we implement the Sampurna tool using MS-Access application.

Our future research directions include finding a generic way of using the Sampurna tool for other systems. Investigating the possibility of automatically generating contracts from the requirements as well as identifying equivalent classes of test cases to eliminate redundant test cases by exploiting deductive processing capabilities of formal techniques as illustrated in [Sinha99].

Appendix A

Appendix

A.1 Illustration of Sampurna Tool

1. **Step 1:** The tool imports the relationship chain in final group of variable values considering all possible combinations.

   Importing the relationship chain refers to gathering of all the sets of variables and their values from all the modules (like Sensor, Manual and Pump module) which are in fact the part of condition set and still satisfy the condition set.

   (a) Doing an import chain on the relation set: $((InformPCW = true, InformPCM = false, X), State = Off)$ We get:

      i. $((Hwl = true, Cml = false, Ccol = false, Cafl = false, ReqPumpC = noReq/reqon/reqoff), State = Off)$

      ii. $((Hwl = true, Cml = false, Ccol = false, Cafl = true, ReqPumpC = noReq/reqon/reqoff), State = Off)$

      iii. $((Hwl = true, Cml = false, Ccol = true, Cafl = false, ReqPumpC = noReq/reqon/reqoff), State = Off)$

      iv. $((Hwl = true, Cml = false, Ccol = true, Cafl = true, ReqPumpC = noReq/reqon/reqoff), State = Off)$

(b) Doing an import chain on the relation set: $((InformPCW = false, InformPCM = true, X), State = Off)$ We get:

   i. $((Hwl = false, Cml = true, Ccol = false, Cafl = false, ReqPumpC = noReq/reqon/reqoff), State = Off)$

   ii. $((Hwl = false, Cml = true, Ccol = false, Cafl = true, ReqPumpC = noReq/reqon/reqoff), State = Off)$

   iii. $((Hwl = false, Cml = true, Ccol = true, Cafl = false, ReqPumpC = noReq/reqon/reqoff), State = Off)$

   iv. $((Hwl = false, Cml = true, Ccol = true, Cafl = true, ReqPumpC = noReq/reqon/reqoff), State = Off)$

(c) Doing an import chain on the relation set: $((InformPCW = false, InformPCM = true, X), State = On)$We get:

   i. $((Hwl = false, Cml = true, Ccol = false, Cafl = false, ReqPumpC = noReq/reqon/reqoff), State = On)$

   ii. $((Hwl = false, Cml = true, Ccol = false, Cafl = true, ReqPumpC = noReq/reqon/reqoff), State = On)$

   iii. $((Hwl = false, Cml = true, Ccol = true, Cafl = false, ReqPumpC = noReq/reqon/reqoff), State = On)$

   iv. $((Hwl = false, Cml = true, Ccol = true, Cafl = true, ReqPumpC = noReq/reqon/reqoff), State = On)$

(d) Doing an import chain on the relation set: $((InformLwl = true, InformPCM = false, X), State = On)$ We get:

   i. $((Lwl = true, Cml = false, Ccol = false, Cafl = false, ReqPumpC = noReq/reqon/reqoff), State = On)$

   ii. $((Lwl = true, Cml = false, Ccol = false, Cafl = true, ReqPumpC = noReq/reqon/reqoff), State = On)$

iii. $((Lwl = true, Cml = false, Ccol = true, Cafl = false, ReqPumpC = noReq/reqon/reqoff), State = On)$

iv. $((Lwl = true, Cml = false, Ccol = true, Cafl = true, ReqPumpC = noReq/reqon/reqoff)State = On)$

(e) Importing the chains for the relation $((InformLwl = false, InformPCM = false, X), State = On)$ we get :

i. $((Lwl = false, Cml = false, Ccol = false, Cafl = false, ReqPumpC = noReq/reqon/reqoff), State = On)$

ii. $((Lwl = false, Cml = false, Ccol = false, Cafl = true, ReqPumpC = noReq/reqon/reqoff), State = On)$

iii. $((Lwl = false, Cml = false, Ccol = true, Cafl = false, ReqPumpC = noReq/reqon/reqoff), State = On)$

iv. $((Lwl = false, Cml = false, Ccol = true, Cafl = true, ReqPumpC = noReq/reqon/reqoff), State = on)$

(f) The chain import of the relation $((InformLwl = false, InformPCM = true, X), State = On)$generates the followings set:

i. $((Lwl = false, Cml = true, Ccol = false, Cafl = false, ReqPumpC = noReq/reqon/reqoff), State = on)$

ii. $((Lwl = false, Cml = true, Ccol = false, Cafl = true, ReqPumpC = noReq/reqon/reqoff), State = on)$

iii. $((Lwl = false, Cml = true, Ccol = true, Cafl = false, ReqPumpC = noReq/reqon/reqoff), State = on)$

iv. $((Lwl = false, Cml = true, Ccol = true, Cafl = true, ReqPumpC = noReq/reqon/reqoff), State = on)$

(g) Importing the relationship chain for $((ReqPumpC = reqon, Y, Z)$ we get:

i. $((Lwl/Hwl = false/true, Cml = true/false, Ccol = false/true, Cafl = false/true, ReqPumpC = reqon), State = on)$

ii. $((Lwl = true, Cml = true/false, Ccol = false/true, Cafl = true/false, ReqPumpC = reqon), State = off)$

iii. $((Hwl = true, Cml = true/false, Ccol = false/true, Cafl = true/false, ReqPumpC = reqon), State = off)$

iv. $((Lwl/Hwl = false/true, Cml = true, Ccol = true/false, Cafl = false/true, ReqPumpC = reqon), State = off)$

v. $((Lwl/Hwl = false/true, Cml = true/false, Ccol = true, Cafl = true/false, ReqPumpC = reqon), State = off)$

vi. $((Lwl/Hwl = false/true, Cml = true/false, Ccol = true/false, Cafl = true, ReqPumpC = reqon), State = off)$

(h) Finally, importing the chain for $((ReqPumpC = reqoff, Y), Z)$ We get

i. $((Lwl/Hwl = false/true, Cml = true/false, Ccol = false/true, Cafl = false/true, ReqPumpC = reqoff), State = off)$

ii. $((Lwl = true, Cml = true/false, Ccol = false/true, Cafl = true/false, ReqPumpC = reqoff), State = on)$

iii. $((Hwl = true, Cml = true/false, Ccol = false/true, Cafl = true/false, ReqPumpC = reqoff), State = on)$

iv. $((Lwl/Hwl = false/true, Cml = true, Ccol = true/false, Cafl = false/true, ReqPumpC = reqoff), State = on)$

v. $((Lwl/Hwl = false/true, Cml = true/false, Ccol = true, Cafl = true/false, ReqPumpC = reqoff), State = on)$

vi. $((Lwl/Hwl = false/true, Cml = true/false, Ccol = true/false, Cafl = true, ReqPumpC = reqoff), State = on)$

The variable set $(InformPCW = true, InformPCM = false, X)$ holds when water level is equal to Hwl. Considering all the different values which X variable may take to satisfy the above condition, X could be variables $Cml, Ccol, Cafl$ and $ReqPumpC$ and their associated values to make a separate test case, such that the variable set $(State = Off)$ still satisfies.

2. **Step 2:** Based on the test cases generated in **Step 1**, the tool filters out the test cases which are meaningless and irrelevant. This is achieved by using *a priori* knowledge of the system such that variable-value pairs that are not attainable/possible with respect to the system specification are not included in the resulting test cases. For example, the three combinations for the variable *ReqPumpC* in Step 1 (a) are not required. The values *noReq* and *reqon* are the only significant values when the pump is in off state i.e., *State=off.* It would be irrelevant to have value *reqoff* for *ReqPumpC* when the variable state is already *State=off.* Likewise, we eliminate all such conditions and arrive at a smaller set of test cases.

3. **Step 3:** The reduced set of test cases are parsed again with respect to the constraints to generate final relevant test cases. For example, as shown below in$(a)i$, the final test case which we get, consists of value *noReq* for variable *ReqPumpC,* which is achieved by parsing the test cases generated in **Step 2** with respect to set of contracts. Note that these resulting test cases form a complete test suite for critical functional testing of the mine-pump.

   (a) We have,

      i. $((Hwl = true, Cml = false, Ccol = false, Cafl = false, ReqPumpC = noReq), State = Off) \rightarrow \Diamond(PState = on))$

      ii. $((Hwl = true, Cml = false, Ccol = false, Cafl = true, ReqPumpC = noReq), State = Off) \rightarrow \Diamond(PState = off))$

iii. $((Hwl = true, Cml = false, Ccol = true, Cafl = false, ReqPumpC = noReq), State = Off) \to \Diamond(PState = off))$

iv. $((Hwl = true, Cml = false, Ccol = true, Cafl = true, ReqPumpC = noReq), State = Off) \to \Diamond(PState = off))$

(b) We have,

i. $((Hwl = false, Cml = true, Ccol = false, Cafl = false, ReqPumpC = noReq), State = Off) \to \Diamond(PState = off))$

ii. $((Hwl = false, Cml = true, Ccol = false, Cafl = true, ReqPumpC = noReq), State = Off) \to \Diamond(PState = off))$

iii. $((Hwl = false, Cml = true, Ccol = true, Cafl = false, ReqPumpC = noReq), State = Off) \to \Diamond(PState = off))$

iv. $((Hwl = false, Cml = true, Ccol = true, Cafl = true, ReqPumpC = noReq), State = Off) \to \Diamond(PState = off))$

(c) We have,

i. $((Hwl = false, Cml = true, Ccol = false, Cafl = false, ReqPumpC = noReq), State = on) \to \Diamond(PState = on))$

ii. $((Hwl = false, Cml = true, Ccol = false, Cafl = true, ReqPumpC = noReq), State = on) \to \Diamond(PState = on))$

iii. $((Hwl = false, Cml = true, Ccol = true, Cafl = false, ReqPumpC = noReq), State = on) \to \Diamond(PState = on))$

iv. $((Hwl = false, Cml = true, Ccol = true, Cafl = true, ReqPumpC = noReq), State = on) \to \Diamond(PState = on))$

(d) We have,

i. $((Lwl = true, Cml = false, Ccol = false, Cafl = false, ReqPumpC = noReq), State = on) \to \Diamond(PState = off))$

ii. $((Lwl = true, Cml = false, Ccol = false, Cafl = true, ReqPumpC = noReq), State = on) \rightarrow \Diamond(PState = off))$

iii. $((Lwl = true, Cml = false, Ccol = true, Cafl = false, ReqPumpC = noReq), State = on) \rightarrow \Diamond(PState = off))$

iv. $((Lwl = true, Cml = false, Ccol = true, Cafl = true, ReqPumpC = noReq), State = on) \rightarrow \Diamond(PState = off))$

(e) We have,

i. $((Lwl = false, Cml = false, Ccol = false, Cafl = false, ReqPumpC = noReq), State = on) \rightarrow \Diamond(PState = off))$

ii. $((Lwl = false, Cml = false, Ccol = false, Cafl = true, ReqPumpC = noReq), State = on) \rightarrow \Diamond(PState = off))$

iii. $((Lwl = false, Cml = false, Ccol = true, Cafl = false, ReqPumpC = noReq), State = on) \rightarrow \Diamond(PState = off))$

iv. $((Lwl = false, Cml = false, Ccol = true, Cafl = true, ReqPumpC = noReq), State = on) \rightarrow \Diamond(PState = off))$

(f) We have,

i. $((Lwl = false, Cml = true, Ccol = false, Cafl = false, ReqPumpC = noReq), State = on) \rightarrow \Diamond(PState = off))$

ii. $((Lwl = false, Cml = true, Ccol = false, Cafl = true, ReqPumpC = noReq), State = on) \rightarrow \Diamond(PState = off))$

iii. $((Lwl = false, Cml = true, Ccol = true, Cafl = false, ReqPumpC = noReq), State = on) \rightarrow \Diamond(PState = off))$

iv. $((Lwl = false, Cml = true, Ccol = true, Cafl = true, ReqPumpC = noReq), State = on) \rightarrow \Diamond(PState = off))$

(g) We have,

i. $((Lwl/Hwl = false, Cml = false, Ccol = false, Cafl = false, ReqPumpC = reqon), State = off) \rightarrow \Diamond(PState = on))$

ii. $((Lwl = true, Cml = true/false, Ccol = false/true, Cafl = true/false, ReqPumpC = reqon), State = off) \rightarrow \Diamond(PState = off))$

iii. $((Hwl = true, Cml = false, Ccol = false, Cafl = false, ReqPumpC = reqon), State = off) \rightarrow \Diamond(PState = on))$

iv. $((Lwl/Hwl = false/true, Cml = true, Ccol = true/false, Cafl = false/true, ReqPumpC = reqon), State = off) \rightarrow \Diamond(PState = off))$

v. $((Lwl/Hwl = false/true, Cml = true/false, Ccol = true, Cafl = true/false, ReqPumpC = reqon), State = off) \rightarrow \Diamond(PState = off))$

vi. $((Lwl/Hwl = false/true, Cml = true/false, Ccol = true/false, Cafl = true, ReqPumpC = reqon), State = off) \rightarrow \Diamond(PState = off))$

(h) We have,

i. $((Lwl/Hwl = false/true, Cml = true/false, Ccol = false/true, Cafl = false/true, ReqPumpC = reqoff), State = on) \rightarrow \Diamond(PState = off))$

ii. $((Lwl = true, Cml = true/false, Ccol = false/true, Cafl = true/false, ReqPumpC = reqoff), State = on) \rightarrow \Diamond(PState = off))$

iii. $((Hwl = true, Cml = true/false, Ccol = false/true, Cafl = true/false, ReqPumpC = reqoff), State = on) \rightarrow \Diamond(PState = on))$

iv. $((Lwl/Hwl = false/true, Cml = true, Ccol = true/false,$

$Cafl = false/true, ReqPumpC = reqoff), State = on) \rightarrow \Diamond(PState = off))$

v. $((Lwl/Hwl = false/true, Cml = true/false, Ccol = true, Cafl = true/false, ReqPumpC = reqoff), State = on) \rightarrow \Diamond(PState = off))$

vi. $((Lwl/Hwl = false/true, Cml = true/false, Ccol = true/false, Cafl = true, ReqPumpC = reqoff), State = on) \rightarrow \Diamond(PState = off))$

Appendix B

Appendix

AirFlowSensor	Carbondioxide	Manual	MethaneSensor	PumpController	WaterSensor
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformLWL=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=True

Table B.1: TestCaseGeneration - 1

AirFlowSensor	Carbondioxide	Manual	MethaneSensor	PumpController	WaterSensor
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=True

Table B.2: TestCaseGeneration - 2

AirFlowSensor	Carbondioxide	Manual	MethaneSensor	PumpController	WaterSensor
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=True

Table B.3: TestCaseGeneration - 3

AirFlowSensor	Carbondioxide	Manual	MethaneSensor	PumpController	WaterSensor
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=On	InformPCW=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=On	InformLWL=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=On	InformPCW=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=On	InformLWL=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=On	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=On	InformLWL=False

Table B.4: TestCaseGeneration - 4

AirFlowSensor	**Carbondioxide**	**Manual**	**MethaneSensor**	**PumpController**	**WaterSensor**
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=On	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=On	InformPCW=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformLWL=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformLWL=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=True
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformLWL=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=True

Table B.5: TestCaseGeneration - 5

AirFlowSensor	Carbondioxide	Manual	MethaneSensor	PumpController	WaterSensor
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformLWL=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=True
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformLWL=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformLWL=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformLWL=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformLWL=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformLWL=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformLWL=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=Off	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=False
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=True
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=Off	InformLWL=False

Table B.6: TestCaseGeneration - 6

Appendix C

Appendix

AirFlowSensor	Carbondioxide	Manual	MethaneSensor	PController	WaterSensor	OutPut
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True	PState=On
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True	PState=On
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=Off	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=False	PState=On
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=False	State=On	InformLWL=False	PState=Off

Table C.1: ResultantTestCase - 1

AirFlowSensor	Carbondioxide	Manual	MethaneSensor	PController	WaterSensor	OutPut
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=On	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=On	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=NoReq	InformPCM=True	State=On	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=NoReq	InformPCM=True	State=On	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=False	PState=On
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=False	PState=On
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True	PState=On
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=True	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformLWL=True	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=False	State=Off	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformLWL=True	PState=Off

Table C.2: ResultantTestCase - 2

AirFlowSensor	Carbondioxide	Manual	MethaneSensor	PController	WaterSensor	OutPut
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOn	InformPCM=True	State=Off	InformPCW=True	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=True	PState=On
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=True	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=True	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=True	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=False	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=True	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=False	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformLWL=True	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=False	State=On	InformPCW=True	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformLWL=True	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=False	PState=Off
InfoSupAf=True	InfoSupCo=True	ReqPumpC=ReqOff	InformPCM=True	State=On	InformPCW=True	PState=Off

Table C.3: ResultantTestCase - 3

Bibliography

[Amla92] Nina Amla and Paul Ammann. Using Z Specifications in Category Partition Testing. Proceeding of COMPASS 1992, Seventh Annual Conference on Computer Assurance, pages 310, 1992.

[Berg96] L. Bergmans, M. Akshit, Composing Synchronization and Real-Time Constraints, Journal of Parallel and Distributed Computing, 36(1), pp. 32-52, 1996.

[Chen03] T.Y. Chen, Pak-Lok Poon, T.H. Tse, A Choice Relation Framework for Supporting Category-Partition Test Case Generation , IEEE Trans. On Software Engg, June, 2003.

[Dick93] J. Dick and A. Faivre, Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In J. C. P. Woodcock and P. G. Larsen, editors, FME'93: Industrial-Strength Formal Methods, Formal Methods Europe, Springer-Verlag, Lecture Notes in Computer Science 670, pp. 268-284, April 1993.

[Doye97] N.J. Doye, Order Sorted Computer Algebra and Coercions, Ph.D. Thesis,University of Bath, 1997.

[Dsso97] C. Bourhfir, R. Dssouli, El M. Aboulhamid and N. Rico, Automatic Executable Test Case Generation for Extended Finite State Machine Protocols, IFIP International Workshop on Testing Communicating Systems, IFIP IWTCS'97, Korea, 1997.

[Edwa96] Edward Kit, Software Testing in the Real World Improving the Process, ACM Press - Addison- Wesley, 1996.

[Ehrig90] H.Ehrig and B. Mahr, Fundamentals of Alg. Spec.2, Module Specifications and Constraints, New York: Springer-Verlag,1990.

[Good75] John B. Goodenough and Susan L. Gerhart, Towards a Theory of Test Data Selection, IEEE Transactions on Software Engineering, 1(2), pp 156-173, June 1975.

[Groch93] Matthias Grochtmann and Klaus Grimm, Classification Trees for Partition Testing, Software Testing, Verification and Reliability, pp. 63-82, 1993.

[Guo02] Jiang Guo, Using Category Theory to Model Software Component Dependencies, 9th Annual IEEE Intl. Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2002), pp. 185-192, April 2002.

[Hall88] P. A. V. Hall, Towards Testing with Respect to Formal Specifications, Second IEE/BCS Conference on Software Engineering, pp. 159-163, 1988.

[Hayes86] Ian Hayes, Specification Directed Module Testing, IEEE Transactions on Software Engineering, 12(1), pp. 124-133, January 1986.

[Hoare89] C.A.R. Hoare, Notes on an Approach to Category Theory for Computer Scientists, In M. Broy, editor, Constructive Methods in Computing Science, pages 245{305. International Summer School directed by F.L. Bauer [et al.], Springer Verlag, 1989. NATO Advanced Science Institute Series (Series F: Computer and System Sciences Vol. 55).

[Jose97] M. Joseph, Real-Time Systems: Spec., Verification and Analysis , Prentice Hall, London, 1997

[Lee96] D. Lee and M. Yannakakis, Principles and Methods of Testing Finite State Machines-A Survey. Proceedings of the IEEE, vol. 84, no. 8. pp. 1090-1123, 1996.

[Mand95] D. Mandrioli, S. Morasca, A. Morzenti, Generating Test Cases for Real-Time Systems from Logic Specifications, ACM Trans. On Computer Systems, Vol 13, Issue 4, pp. 365-398, 1995.

[Nait81] S. Naito and M. Tsunoyama, Fault Detection for Sequential Machines by Transition-Tours, Proceedings of 11th IEEE Fault Tolerant Computing Conference, pp. 238-243, 1981.

[Ostr88] Thomas J. Ostrand and Marc J. Balcer, The Category-Partition Method for Specifying and Generating Functional Tests, Communications of the ACM, 31(6), pp. 676-686, June 1988.

[Pier91] B.C. Pierce, Basic Category Theory for Computer Scientists, Cambridge, MA: M.I.T. Press, 1991.

[Raus02] Andreas Rausch, Design by Contract + Componentware-Design by Signed Contract, Journal of Object Technology, 1(3), pp. 19-36, 2002.

[Sidh89] Deepinder P. Sidhu and Ting-Kau Leung, Formal Methods for Protocol Testing: A Detailed Study, IEEE Trans. Soft. Eng., Vol. SE-15, pp. 413-426 , April 1989.

[Srini96] Y.V. Srinivas and J.L. McDonald, The Architecture of SPECWARE, a Formal Software Development System, Technical Report, Kestrel Institute, 1996.

[Sinha99] P. Sinha, N. Suri, Identification of Test Cases Using a Formal Approach, FTCS-29, Madison, USA, pp.314-321, 1999.

[Stock96] P. Stocks, D. Carrington, A Framework for Specification-Based Testing, IEEE Transactions on Software Engg., 22(11), pp 777-793, 1996.

[Szyp97] C. Szyperski, Component Software, Beyond Object-Oriented Programming, Addison Wesley Longman Limited ,1997.

[Varm03] N. Varma, S. Kanade, P. Sinha, Composition of Modules with Synchronization and Real-Time Constraints, Technical Report at Concordia University, Montreal, Canada., Jan 2003.

[Willi99] K. Williamson, M Healy, Industrial Applications of Software Synthesis via Category Theory, 14th IEEE International Conference on Automated Software Engineering, pp. 35, Oct. 1999.