ABSTRACT

Title of Dissertation: DOMAIN SPECIFIC TEST CASE GENERATION USING HIGHER ORDERED TYPED LANGUAGES FOR SPECIFICATION

Avik Sinha, Ph.D 2005

Dissertation Directed By: Dr. Carol S. Smidts,
Associate Professor,
Department of Mechanical Engineering.

Model based testing is an approach for automatic generation of test cases on the basis of a representative model of the system. Recent studies show that model based testing has many possible advantages over manual test generation techniques including a gain in effectiveness, efficiency and reuse.

The effectiveness (ability to uncover faults in a system) of a model based testing process is determined by the correctness of the model and by the number of requirements represented in the model. In practice, test models for model based test automation techniques are usually created from requirement or design specifications of the software and hence, these techniques overtly rely on such specifications for the completeness of the test models. This may lead to failure in testing some critical requirements specific to the application domain because the user, who helps in defining the requirements, may fail to consider certain domain specific requirements. To him some may appear to be too trivial to be specified explicitly in the

requirements document and the others, he may forget. Even if the requirement is complete with domain specific requirements, testers may not realize criticality of such requirements or may find them too complex to model. In all such cases, testing is incomplete and ineffective.

This dissertation describes a new model based testing technique developed to remedy such situations. The new technique is based on modeling the system under test using a strongly typed domain specific language (DSL). In the new technique, information about domain specific requirements of an application are captured automatically by exploiting properties of the DSL and are subsequently introduced in the test model. The new technique is applied to generate test cases for the applications interfacing with relational databases and the example DSL chosen for that purpose is HaskellDB. Test suites generated using the new technique are enriched with test cases addressing domain specific implicit requirements and therefore, are more effective in finding faults.

This dissertation will present details of the technique and describe an experiment and a case study to explore its effectiveness, efficiency, usability and industrial applicability.

DOMAIN SPECIFIC TEST CASE GENERATION USING HIGHER ORDERED TYPED LANGUAGES FOR SPECIFICATION

By

Avik Sinha

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2005

Advisory Committee:
Professor Carol Smidts, Chair
Professor Ali Mosleh
Professor Mohammad Modarres
Professor Marvin Zelkowitz
Professor Michel Cukier
Dr. Clay Williams

Acknowledgements

I wish to express my sincere gratitude to Dr. Carol Smidts for without her immense help in guiding my research, this dissertation would have been impossible. As an advisor she assisted me in every aspect from research brainstorming to writing this dissertation.

I would like to thank Drs. Peter Santhanam and Clay Williams at IBM TJ Watson Research Center for their help and support for the scalability study of the methodology presented in this dissertation.

I would also like to thank Drs. Marvin Zelkowitz and Michel Cukier for their help in reviewing prior publications from this research.

I am fortunate to have been able to work on this project with a talented and dedicated team of UMD researchers consisting of Dr. Ming Li, Dr. Bin Li, Susmita Ghose, Dongfeng Zhu, Yuan Wei, Anand Ladda and Wende Kong. Special thanks are presented to them for their help and the support they provided to this project.

I wish to acknowledge the support of Dr. Jim Widmaier and the National Security Agency during the earlier phases of this research. I am also grateful to Dr. Andrew Moran at Galois Connections Inc. for helping me to learn HaskellDB.

Finally, I would like to thank Dr. Mohammad Modarres, Dr. Ali Mosleh, Dr. Marvin Zelkowitz, Dr. Michel Cukier and Dr. Clay Williams for agreeing to be on my committee.

Table of Contents

Acknowledgements ii

Table of Contents iii

List of Tables ix

List of Figures xi

Chapter 1 Introduction 1

1.1 Research Objective 1

1.2 Research Statement 1

1.3 Approach 3

1.4 Content 4

1.5 Summary of Contributions 6

1.6 References 7

Chapter 2 Literature Review 9

2.1 Model Based Testing 9

2.2 Domain Specific Testing 12

2.3 Motivation 13

2.4 Summary 14

2.5 References 15

Chapter 3 Domain Specific Languages and Model Based Testing 18

3.1 Application Specific Needs and Languages 18

3.2 What is a DSL? 19

3.3 Advantages of DSL 21

3.4 Challenges of using DSL 23
3.5 DSLs and Model Based Testing 24
3.6 Types and Functions of HaskellDB 24
3.6.1 HaskellDB Types 25
3.6.2 HaskellDB Functions 26
3.7 HaskellDB Axioms and Domain Specific Requirements 27
3.8 Summary 31
3.9 References 32
Chapter 4 Derivation of a Structural Representation 34
4.1 Derivation of EFSM 35
4.1.1 Three styles. 35
4.1.2 Specification to Actual Flow 38
4.2 Derivation of EFSMA 42
4.2.1 Treatment of Parameters 42
4.3 Specification with Conditional Flow 47
4.4 Special Case of Recursion 51
4.5 Summary 53
4.6 References 53
Chapter 5 Tool Support for HOTTest 54
5.1 Modeling of SSP 55
5.2 Component 1: Hugs Interpreter 58
5.3 Component 2: Call Graph Generator 58
5.4 Component 3: EFSM Generator: 61

5.5 Component 4: Test Generator 63
5.6 Summary 63
5.7 References 63
Chapter 6 Experimental Validation of Usability and Performance 65
6.1 The Experiment Design 66
6.1.1 The Research Question 66
6.1.2 Variables 66
6.1.3 Measurement Models 67
6.1.4 Hypothesis 71
6.1.5 Design 73
6.1.6 Threats to Validity 74
6.2 Experiment Preparation 75
6.2.1 Subjects 75
6.2.2 Applications 76
6.2.3 Requirement Parsing 77
6.3 Experiment 78
6.4 Measurement and Analysis 80
6.4.1 Usability -Learning 81
6.4.2 Usability-Efficiency 83
6.4.3 Usability- Error 85
6.4.4 Usability- Satisfaction and Ease 85
6.4.5 Performance-Effectiveness 86
6.4.6 Performance- Efficiency 88

6.5 Results and Discussion 90
6.6 Summary 93
6.7 References 94
Chapter 7 Industrial Applicability of HOTTest and Other Test Generation Tools 96
7.1 Description of the Test Design Tools 96
7.1.1 Archetest 96
7.1.2 ASMLT 98
7.1.3 EFSM Based Test Generation 100
7.2 Design of Case Study 101
7.2.1 Design of the Measurement Framework 101
7.2.2 Case Study Instruments 108
7.2.3 Case Study: Process 110
7.2.4 Threats to Validity 112
7.3 Case Study Results 114
7.3.1 Complexity of the Modeling Process 114
7.3.2 Ease of Learning 124
7.3.3 Effectiveness 127
7.3.4 Efficiency 129
7.3.5 Scalability 132
7.4 Analysis of the Results 135
7.5 Summary 139
7.6 References 139
Chapter 8 Extension to other Domains 142

8.1 Steps for Extension 142
8.1.1 Domain Analysis 142
8.1.2 Design/ Choice of a Domain Specific Language 143
8.1.3 Associate Requirements to DSL constructs 144
8.2 Extension to the domain of Graphical User Interface 146
8.2.1 Domain Analysis 146
8.2.2 Choice of Domain Specific Language 147
8.2.3 Associating Requirements to DSL constructs 149
8.3 Summary 152
8.4 References 152
Chapter 9 Conclusion and Future Work 153
9.1 Advantages of HOTTest 153
9.2 Limitations of HOTTest 155
9.3 Future Research 157
Appendix A: HaskellDB 159
Appendix B: HaskellDB Axioms 179
Appendix C: TclHaskell Axioms 182
Appendix D: Experiment Data for Usability Experiment 189
Bibliography 191

List of Tables

Table 3.1: Axioms derived from HaskellDB 31

Table 6.1: Dependent Variables of the Experiment 67

Table 6.2: Measurement models used in the experiment 71

Table 6.3: Students' experience profile 76

Table 6.4: Number of Requirements in the Requirements List 77

Table 6.5: Schedule for the experiment 78

Table 6.6: A descriptive statistics of learnability data 82

Table 6.7: A descriptive statistics of efficiency data 84

Table 6.8: A descriptive statistics of data on *error index* 85

Table 6.9: A descriptive statistics of subjective attributes of usability data 86

Table 6.10: A descriptive statistics of effectiveness data 87

Table 6.11: A descriptive statistics of effectiveness of test suites for generic requirements 88

Table 6.12: A descriptive statistics of efficiency data 89

Table 6.13: An Overview of the Analysis 90

Table 6.14: Comparison of Performance between first and second rounds of assignment 93

Table 7.1: List of Concepts for the tools 115

Table 7.2: Learnability Data for the tools 124

Table 7.3: Effectiveness and Domain-specific effectiveness calculations 128

Table 7.4: The data for testing effort 129

Table 7.5: Efficiency measurements 132

Table 7.6: The data for testing effort for SearchPUBS 133

Table 7.7: Absolute and Normalized Scalability Values for the Tools 133

Table 7.8: Summary of Measurement Results. 135

Table 7.9: Classification of the domain specific requirements missed by the techniques 137

Table 7.10: Efficiency Calculations considering only the Contributions of Manual Effort 138

Table 7.11: Test effort data from IBM projects 138

Table 8.1: The function categories of TclHaskell and the associated requirements. 152

Table D.1: Results of Statistical Analysis of Data 190

List of Figures

Figure 1.1: The Test Framework using HOTTest 4

Figure 4.1: The derived EFSM 41

Figure 4.2: The EFSM with embedded variables 45

Figure 4.3: The EFSM with axioms introduced, EFSMA. 47

Figure 4.4: EFSM with the embedded variables 51

Figure 4.5: Model Depicting Recursion 53

Figure 5.1: The Architecture of HOTTest 54

Figure 5.2: A screenshot from SSP Application 56

Figure 5.3: HaskellDB Model for SSP 57

Figure 5.4: Screenshot Depicting Hugs Interface 58

Figure 5.5: Algorithm for Call Graph Generation 60

Figure 5.6: A Screenshot from Call Graph Generator 61

Figure 6.1: The Experiment Design 73

Figure 6.2: Box plots of the data for the two test techniques. 83

Figure 7.1: The Test Framework using Archetest 98

Figure 7.2: The Test Framework using ASMLT 100

Figure 7.3: The Test Framework using EFSM based modeling 101

Figure 7.4: Different phases of the case study 112

Figure 7.5 Semantic Dependency Graph for HOTTest- Novice's Perspective 116

Figure 7.6 Semantic Dependency Graph for HOTTest- Expert's Perspective 117

Figure 7.7 Semantic Dependency Graph for Archetest- Novice's Perspective 118

Figure 7.8 Semantic Dependency Graph for Archetest- Expert's Perspective 119

Figure 7.10 Semantic Dependency Graph for ASML- Expert's Perspective 121

Figure 7.11 Semantic Dependency Graph for EFSM- Novice's Perspective..... 122

Figure 7.12 Semantic Dependency Graph for EFSM- Expert's Perspective..... 122

Figure 7.13: Comparative analysis of complexity of the tools 123

Figure 7.14: Comparative plots of the absolute proficiencies attained by the users. 125

Figure 7.15: A comparative plot of the learning time..... 126

Figure 7.16: A comparative plot of the normalized values of learnability 126

Figure 7.17: Comparative plots of effectiveness values 129

Figure 7.18: A comparative plot of the net effort and its contributors 131

Figure 7.19: A Comparative Plot of Efficiency Calculations 132

Figure 7.20: Comparative Plots of Normalized Scalability 135

Chapter 1 Introduction

1.1 Research Objective

The objective of this research is to develop an automatic test generation technique that can generate test cases automatically from specification documents while accounting for domain specific requirements of applications. This research proposes a new model based test design technique that achieves this objective. Through this research we also provide proofs of the fact that such a test generation technique satisfies the usability requirements of the test case generation processes used in the industry and is scalable to large scale industrial problems.

1.2 Research Statement

Testing is a vital part of the software development lifecycle and is necessary to ensure software correctness. Proper testing is necessary to ensure and enhance reliability of software. Test Automation is the process of automating the test generation and execution process to make it effective and efficient. Test automation techniques can be white-box or black-box test automation techniques depending on whether or not the automation process needs access to the source code. White-box test automation techniques are used for unit-testing and similar testing assignments where the size of the code is small.[2] For larger and complex codes white-box testing is inappropriate, and one has to resort to black box test automation techniques. Black-box testing is also important for system level testing of applications.

Automating black box testing requires generating test cases on the basis of a representative model of the system called the test model. These techniques are, therefore, collectively known as model based test automation techniques. Models not

only enhance the understanding of a product and its architecture, but enable one to semi-automatically derive test cases at an early development stage.[3] Model based test automation techniques help in making the test-generation process faster and make it less susceptible to human error by automating routine and error prone tasks. They also help in making the test process more reproducible by making the process less dependent on human interpretation. With suitable enhancements models can be used to generate scripts for executing test cases using the commercially available test harnesses like WinRunner[7], SilkTest [6] or RationalXDE[5].

Test models for model based test automation techniques are usually created from requirement or design specifications of the software and hence, these techniques overtly rely on the specification for the completeness of the test models. This may lead to failure in testing some critical requirements specific to the application domain because the user, who is familiar with the domain and defines the requirements, may consider certain domain specific requirements to be too trivial to be specified explicitly in the requirements document. The tester and the developer may not have the necessary domain knowledge and hence, may never realize that such a requirement is missing. Even if the tester is aware of some domain specific requirements, due to the complexity of the application, it might be difficult for the tester to generate test cases for such requirements. For example consider the domain of database applications and consider an application for querying a static relational database. It is possible that there will be no explicit requirement in the specification document that entails the application to throw warnings on domain specific errors like generating queries for a non-existent field. This requirement is nevertheless

important, more so when the application interfaces with other applications. Therefore, it is necessary to test the application for such requirements.

1.3 Approach

In this research we develop a model based testing technique called HOTTest, which reduces such testing errors and makes testing more effective. HOTTest is an acronym for Higher Ordered Typed specification-based testing. It uses a higher-ordered-typed domain-specific specification language (e.g. HaskellDB) to model the system. This enables HOTTest to develop the test oracle automatically. Further, HOTTest can extract domain specific axioms from the model to create additional test cases.

A *domain-specific language* (DSL) is a small, usually declarative language that offers expressive power focused on a particular problem domain [1]. Through suitable abstractions, through embedded types and through specific library functions, the DSL imports domain knowledge into any application. Information about domain specific requirements can be captured automatically by exploiting properties of the DSL.

Higher Order Programming is the ability to use functions as values [4]. Using a higher ordered language one can pass functions as arguments to other functions and functions can be the return value of other functions. This style of programming is mostly used in functional programming, but it can also be very useful in other forms of programming. Because of a declarative nature of higher ordered languages, they can serve as ideal languages for specifying systems. A higher-ordered typed language is any higher ordered language which is strongly typed. A strongly typed

language allows us to derive system axioms[1] based on type constraints. If the types are so embedded that they capture domain concepts, then these axioms provide useful information about domain specific requirements (DSRs). In this dissertation we show how one can extract the axioms and use them to enrich the test suite with domain specific test cases. The process of test automation in HOTTest involves translation of a system specification written in a Higher-Ordered Typed language into an intermediate representation similar to extended finite state machine (EFSM) based representations of the system. The intermediate representation can then be used as input to all EFSM based test design tools, which can generate test cases from the test model automatically while satisfying various coverage and adequacy criteria.

Figure 1.1 presents an overview of the test generation framework

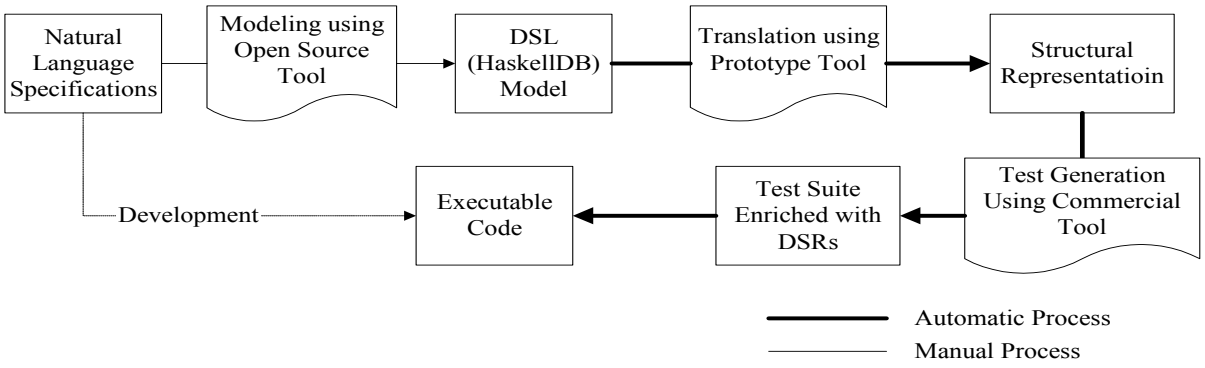

Figure 1.1: The Test Framework using HOTTest

1.4 Content

Chapter 2 of this thesis presents an outline of the related work. Related research in the fields of model based testing and domain specific testing are explored. Based on the state of the current research the motivation for this research is developed.

[1] Axioms are properties that are true for any system specified in HaskellDB.

In Chapter 3 we define domain specific languages and discuss their advantages and disadvantages. A brief description of HaskellDB is provided and the types and functions of HaskellDB are introduced. In later sections of the chapter we identify the domain specific axioms for applications interfacing with relational databases and then we demonstrate how the domain specific axioms can be associated with specific constructs of HaskellDB.

Chapter 4 explains the methodology for derivation of the structural representation (EFSM) from the DSL based representation of the system. This step is necessary in order to define the coverage and adequacy criteria and to generate test cases using tools for EFSM based test case generation. The chapter describes the process of flow derivation, the process of parameter mapping and the process of axiom embedding. Later sections of the chapter illustrate how the conditional flows are translated into test model constraints and how the special case of recursion is handled.

The implementation of the prototype tool is described in Chapter 5. The tool architecture and the primary modules of HOTTest implementation are discussed.

In order to study the usability of HOTTest and to compare it with other model based test design techniques an in-vitro experiment was designed. Chapter 6 reports the design of the experiment and describes the results of the experiment.

Chapter 7 describes a case study conducted in an industrial setting to assess the scalability of HOTTest to industry scale problems and also to compare its performance with other model based test design techniques.

In Chapter 8 we define the general principles of extending HOTTest to the other domains of interest. The principles are explained with the example domain of applications based on graphical user interfaces.

Chapter 9 concludes this thesis by highlighting the advantages and limitations of HOTTest as a model based test design tool. Possible avenues for future research are also discussed.

1.5 Summary of Contributions

The significant contributions of this dissertation are as follows:

1. Development of a new model based test generation technique: Through this research a new model-based test generation technique is developed that can test for domain specific implicit requirements along with other generic functional requirements. The new technique uses a higher ordered strongly typed language for modeling a system and thus introduces a new modeling tool to the software modeling community. The technique also mitigates the shortcomings of other model based test generation techniques that fail to test for domain specific requirements.
2. Development of a methodology for derivation of EFSM from functional specifications: This research develops a methodology for translating a requirements specification written in a strongly typed functional language to a finite state machine based representation. Thus, any application specified in a Higher Ordered language can now be used for test case generation or model checking by utilizing the tools available for finite state machine based representation of the system.

3. Usability, Performance and Scalability assessment of the model based test: This research also addresses the usability, performance and scalability aspects of modeling software using higher ordered languages. Results of such study can assist software modeler in choosing the right tool and technique to achieve their modeling objectives.

This research is a first step in use of higher ordered specification languages for modeling of software applications. It will help in enhancing the state of the art for automatic test generation techniques. It will allow programmers of Higher Ordered languages to access the tools and techniques available for software verification and validation. It will also help in understanding the specific needs of application domains with a view to test case generation.

1.6 References

[1] A. Deursen, P. Klint and J. Visser, "Domain-specific Languages: An annotated bibliography," in *ACM SIGPLAN Notices*, vol.35, no. 6, pp. 26-36, 2000 http://www.cwi.nl/projects/dsl

[2] B. Beizer, *Software Testing Technique* .Boston, USA: International Thomson Computer Press, 1990.

[3] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, "Model-Based Testing with AsmL.NET," *in Proc. 1st European Conference on Model-Driven Software Engineering ,* December 2003.

[4] P. Hudak, *The Haskell School of Expression*. New York: Cambridge University Press, 2000.

[5] *Rational XDE Tester User's Guide,* IBM Corporation., New York, NY, 2004

[6] *Silktest User's Guide*, Version 6.5, Segue Software Inc., Lexington, MA, 2002

[7] *WinRunner User's Guide,* Version 7.01, Mercury Interactive Inc., Sunnyvale, CA, 2001.

Chapter 2 Literature Review

Arguably all software testing activity is model based, since any test case must be designed using some mental model of the application under test. In recent years the use of explicit models for software development has expanded greatly. The use of these models for the generation of test cases in the IT industry is still in its infancy, although a significant part of the telecommunications, aerospace, and micro-electronics industries have been experimenting with models for verification and test generation for over a decade. Recently, the research community has also expressed interest in testing applications that have special requirements associated to their application domains. In this chapter we present an overview of recently published research works on model based testing and domain specific testing.

2.1 Model Based Testing

Model based testing is a test design technique where the basis for test generation for any application is its model representation and not the application itself. The coverage and adequacy criteria are defined on the basis of the model. The test cases are derived from such models and the applications are tested against them. Thus, in effect model based testing is comparison of the system's implementation to its representation (model).

Several current researchers have discussed a variety of model based test generation techniques and have highlighted the advantages of using such techniques. Barnett et al [16] discuss a technique to generate test cases from Abstract State Machine models of systems. In their technique, abstract state machine representations of systems are grouped into hyper states and corresponding finite state machines (FSM) are

produced. The FSM's are then used for generating test cases using graph coverage algorithms. A general pitfall for all finite state machine based test generation techniques is that the state space increases exponentially with increase in system size. Therefore, these model based testing techniques are not suitable for large scale industrial applications.

UML or the Unified Modeling Language is a very common modeling technique used in the industry during software design. Williams [4] shows how one can derive test cases from UML use case specifications. The domain model is a class diagram, and serves to indicate the domain classes that can be instantiated by the system. The use cases are formalized to produce a use case specification by adding five key concepts to standard use cases. A state chart is produced from the activity diagram, which forms the basis for test case generation. The advantage of using an UML based test model is that the model developed for design purposes can easily be enhanced for test case generation, but such effectiveness of testing is dependent on the skill of the tester and completeness of the requirements specification.

Another very popular model based test design technique is based on Extended Finite State Machine (EFSM) models of systems. Savage et. al.[17] discuss an EFSM based test design technique and discuss its use in designing test cases for software in the aerospace industry. EFSMs describe a system's dynamic behavior using hierarchically arranged events, states, and transitions. An event causes a change of state; the state describes a condition of the system; and the transition visually describes the system's new state as a result of a triggering event. In an EFSM the state machine notation is enriched by adding context (history), predicates

(requirements-based behavior control), constraints (test output control), test information (test execution system instructions), nested state machine models (hierarchies of models, or sub-models), and the path flow language (syntax used to control the model's behavior). Other publications like the one by Wang et al [5]and the one by Dsoulli et al [19] discuss industrial use of model based test design. They show how EFSMs can be used for testing various communication protocols. EFSM based testing is superior to finite state machine based testing for the states can be hierarchically grouped and the transitions can be constrained to perform test generation by batches. This allows testers to generate test cases for large scale applications. However, modeling in EFSM itself becomes complicated with increase in application size. This has a very adverse effect on scalability of EFSM based test generation techniques.

Some model based testing techniques are also known as formal-specification-based testing as the test model is a formal specification of the system. Jagadeesan et al [14] provide a nice example for automatic testing of reactive systems. Their methodology is based on specifying the safety requirements of reactive systems using temporal logic. Finite state machine oracles corresponding to the safety properties are generated automatically and test harnesses are built. Hall [18] demonstrates a technique for deriving tests from a Z specification as a system model. Dick et al [11] provide an interesting technique for extraction of test cases from models of the system in VDM. Tretmans et al [12] discuss another technique for generating test cases for applications from Promella, a language used for modeling distributed systems. Some other model based test design techniques employ program

documentation to generate test models. Peters et al [7] propose one such technique for automatic generation of test oracles. The formal methods based test generation techniques do not lend themselves readily for industrial use. Several papers explore the possible reasons for formal methods not being popular in the industry. Luqi et al [15] and Knight et al [10] cite multiple reasons for less use of formal methods in the industry. One of the primary reasons that they cite is that the formal notations are difficult to understand for the practicing programmers. Another reason cited is that the formal methods do not scale up to the industrial problems. Finney [13] also demonstrates through an experiment that the formal notations are difficult to understand for industry practitioners.

2.2 Domain Specific Testing

Computer applications can be broadly grouped into application domains depending on the type of functions they perform. These domains can be named based on a particular feature of that domain eg., the domain of relational databases, the domain of graphical user interface etc. A general problem with the regular test generation techniques is that they don't explicitly account for Domain Specific requirements. We define a domain specific functional requirement (DSR) as a requirement for an application which arises from the knowledge about the application domain and is usually non-significant for applications from other domains. Recent papers have addressed the need for test generation for specific application domains.

Reyes et al [1] provide a framework for developing domain specific testing tools. Their framework provides a library of domain independent components that could be integrated with existing test design techniques to support domain specific

test automation. The process relies on the testers' ability to identify specific domain requirements and introduce ad-hoc libraries for generating test cases. The effectiveness of this work hence is largely dependent on the ability of the testers.

Specific needs of the application domain with regards to testing are highlighted in some other recent publications. Chays et al [6] highlight the issues with testing database applications. More specifically, they provide a test data generation framework for testing requirements specific to database applications. Another research by Memon et al [3] address testing graphical user interface requirements for applications. However, none of these techniques provide a model based framework for capturing domain specific requirements automatically.

2.3 Motivation

In a recent report published by NIST on the economic impacts of software testing [9], the gross annual cost incurred due to insufficient testing of software is estimated to be $59.1 billion. The report also maintains that $22.2 billion of this cost can be recovered by improving the software testing infrastructure through invention of superior test automation techniques and integration of such techniques to existing software development processes. Model based test techniques have many potential benefits, but the present-day state of the art lacks the capability to address domain specific requirements.

The reason for existing model based test design techniques failing to account for DSRs lies in the fact that many of the DSRs are not explicitly discussed in specification documents, as they are considered to be too trivial by the users, who

have extensive knowledge of the domain. A solution to this is a test generation framework that does not rely on the specification document to account for all DSRs.

It is argued that domain specific languages (DSLs) can be effectively used to specify domain specific applications as they also capture DSRs while expressing the functional requirements.[2] As we discuss in the next chapter, DSLs are designed by experts of an application domain and use of DSL in specification imports such expertise into the specification. Leijen et al [8] demonstrate how types can be embedded in a higher ordered typed language like Haskell in order to address specific concerns of relational database based applications. The outcome of their work is HaskellDB, a higher ordered, strongly typed, domain specific and functional language. If such a language is used to model an application interfacing with relational databases, then the model corresponds to the constraints prescribed by the domain experts. Such a model can thus be used to automatically generate test cases for certain domain specific requirements which are not specified explicitly. Further, as is discussed in the next chapter domain specific languages have focused expressive power. This means that modeling domain specific applications in a DSL is relatively easy. This is the motivation for our research. We want to develop a model based test generation technique that is user friendly and which addresses domain specific requirements along with the generic functional requirements.

2.4 Summary

In summary we can say that many of the current researchers identify the benefits of Model Based Testing and a variety of model based test design techniques can be found in the literature. A general concern for model based test design

techniques is that they do not account explicitly for domain specific requirements. Strongly typed domain specific languages can be used as an effective tool for automatically importing domain specific expertise into applications and therefore, can be used for developing test models enriched with domain knowledge.

2.5 References

[1] A. A. Reyes and D. J. Richardson, "Siddhartha:A Method for Generating Domain-Specific Test Driver Generators," in *Proc. 14th IEEE International Conference on Automated Software Engineering*, Cocoa Beach, FL, 1999, pp. 81-90.

[2] A. Deursen, P. Klint and J. Visser, "Domain-specific Languages: An annotated bibliography," in *ACM SIGPLAN Notices*, vol.35, no. 6, pp. 26-36, 2000 http://www.cwi.nl/projects/dsl

[3] A. M. Memon, M. E. Pollack and M. L. Soffa, "Hierarchical GUI test case generation using automated planning," in *IEEE Transactions on Software Engineering*, Volume: 27 Issue: 2, Feb. 2001 pp: 144 –155.

[4] C. E. Williams, "Toward a test-ready meta-model for use cases," *in Proc. Workshop on Practical UML-based Rigorous Development Methods,* Toronto, CA, 2001, 270–287.

[5] C. J. Wang and M. T. Liu, "Generating Test Cases for EFSM with Given Fault Models," in *Proc. IEEE Infocom*, 2, pp. 774-781, 1993.

[6] D. Chays, S. Dan, P.G. Frankl, F.I. Vokolos and E.J. Weyuker, "A Framework for Testing Database Applications," in *Proc. ISSTA 2000*, Portland, 2000, pp. 147-157.

[7] D. K. Peter and D. L. Parnas, "Using Test Oracles Generated from Program Documentation," in *IEEE Transactions on Software Engineering*, Vol 24, No. 3, pp. 161-173,1998.

[8] D. Leijen and E. Meijer, "Domain Specific Embedded Compilers," in *Proc.2nd USENIX Conference on DSL*, Austin, 1999, pp. 109-122.

[9] G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing," in *Planning Report 02-3*. Prepared by RTI for the National Institute of Standards and Technology (NIST), May 2002: see http://www.nist.gov/director/prog-ofc/report02-3.pdf

[10] J. C. Knight, C. L. DeJong, M. S. Gibble and L.G. Nakano, "Why Are Formal Methods Not Used More Widely?" in Proc. 4th NASA Formal Methods Workshop, Hampton, VA, 1997.

[11] J. Dick and A. Faivre, "Automating the Generation and Sequencing of Test Cases from Model-Based Specifications," in *Proc. First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, Odense, 1993, pp.268-284.

[12] J. Tretmans and A. Belinfante, "Automatic testing with formal methods," in *Proc. EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November pp. 8-12, 1999. EuroStar Conferences, Galway, Ireland.

[13] K. Finney, "Mathematical notation in formal specification: Too difficult for the masses?" in IEEE Transactions on Software Engineering, Volume 22, No 2, pp.158-159, 1996

[14] L. J. Jagadeesan, A. Porter, C. Puchol, C. J. Ramming and L. G. Votta, "Specification-based testing of reactive software: tools and experiments," in *Proc.19th International Conference on Software Engineering,* Boston, 1997, pp. 525–535.

[15] Luqi and J. Goguen, "Formal Methods: Problems and Promises," in IEEE Software, Volume 14, No 1, pp 73-85, 1997.

[16] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, "Model-Based Testing with AsmL.NET," *in Proc. 1st European Conference on Model-Driven Software Engineering ,* December 2003.

[17] P. Savage, S. Walters and M. Stephenson, " Automated Test Methodology for Operational Flight Programs," in *Proc. IEEE Aerospace Conference,* vol.4, pp. 293-305, 1997.

[18] P.A.V. Hall, “Relations between Specifications and Testing,” in *Information and Software Technology*, vol.33, no. 1, pp. 47-52, 1991.

[19] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir, “Test Development For Communication Protocols: Towards Automation,” in *Computer Networks*, 31, 1999, pp. 1835-1872.

Chapter 3 Domain Specific Languages and Model Based Testing

A scientific approach may be classified into a generic approach or into a specific approach based on the range of problems the approach addresses. A generic approach attacks a large group of problems. It provides a solution for a spectrum of questions in a certain area. These solutions are often not optimal. On the other hand, a specific approach provides an optimized solution for a smaller set of problems. The same philosophy when extended to the field of computer languages, results in what we classify as generic languages and domain specific languages (DSL). This chapter defines DSLs and identifies the advantages and disadvantages of using a DSL. We also describe how embedded domain specific languages that are strongly typed can be used as effective modeling techniques for test generation.

3.1 Application Specific Needs and Languages

Most of the existing programming languages like COBOL, FORTRAN, and Lisp came into existence as languages dedicated to solve a certain class of problems. COBOL started off as a language to address commerce and business oriented languages. FORTRAN on the other hand, had started off as a language to solve problems regarding formulae translations. Lisp was targeted to attack specific problems in language processing and list handling.

The need for application specific languages arises whenever there is a new application domain. Certain functionalities need to be implemented frequently or they may require special skills to be implemented. Various solutions to this question exist and are tried from time to time: -

- *Function libraries* contain in-built functions that perform related tasks in well-defined domains like, for instance, differential equations, graphics, user-interfaces and databases.
- *Object oriented frameworks* like the Java development environment are just extensions of the previous idea. The objects have some built-in methods and classes that make the programmer's task easy. Classical libraries have a flat structure, and the application can invoke any of the library functions from any part of the code. In object-oriented frameworks, however, a definite hierarchy is followed and there are certain restrictions regarding the accessibilities of various methods.[2]
- A *domain specific language* is a language designed to address the needs of a very specific class of problems.

Although many domain specific languages have been designed and used over the years, the systematic study of domain specific languages has only started in late 90s. An annotated bibliography of DSLs can be found in [2] .

3.2 What is a DSL?

The exact definition for DSLs is debatable. This is mainly because of the range of DSLs that exist today and the various forms in which they exist. DSL or a domain specific language is a programming or specification language dedicated to a particular domain or problem. Deursen et al, define DSL as follows: [2]

> "A *domain specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and

abstractions, expressive power focused on, and usually restricted to, a particular problem domain."

DSLs express just as much as is needed, no more no less. DSLs are often small, offering only a restricted suite of notations and abstractions. Sometimes, they are referred to as micro-languages or little languages in the literature.

Sometimes, DSLs are enhancements of a general programming language. They modify an existing general programming language to make it more suitable to any problem domain. Often known as embedded DSLs they evolve the generic language by inserting types, functions and subroutines into the existing library and by making them specific to any domain. For example HaskellDB is an embedded DSL made out of the functional programming language called Haskell. HaskellDB is a DSL designed to generate type safe SQL queries for any relational database.

Domain specific languages are often claimed to be more declarative than imperative. Imperative languages allow the programmer to define the state variables and to guide the application through various states as he wishes to. Declarative languages on the other hand do not have any explicit states. The states are declared and transformed implicitly [11].

Since DSLs are mostly declarative, they are often also used as specification languages. Since they are designed with the specific application in mind, they often are more efficient in capturing the specifications of any requirement. They are ideal for generating efficient and consistent requirement specifications; efficient because they capture more through a very few constructs and consistent because they generally have in-built consistency checks. Such a requirement specification can be

used as a model for the purpose of model based testing as is described in the later chapters.

Some DSLs are mainly used to generate applications for a certain domain of interest. Such DSLs are some time referred to as application specific languages and the corresponding DSL compilers are called the application generators.

Yet another class of DSLs actually does not aim at generating a complete application. Instead they are aimed at generating libraries/classes for assisting the main application. A common example of such an example could be Lex and Yacc commonly used for the purpose of generation of libraries for compilers.

Numerous examples of DSLs could be found. Some of them are widely used worldwide and are often confused with general programming languages e.g., SQL, Unix shell language, Matlab, etc. Common domains that DSLs address include graphics [9], financial products [8], telephone switching systems [2], protocols [10], operating systems[2], device drivers [3], routers in networks [2], database queries[5], robot languages [6].

3.3 Advantages of DSL

Following are some of the advantages of DSL over generic languages:

1. Ease of learning: DSLs are designed with a specific domain in mind and the DSL design process involves active interaction with domain experts. Therefore one of the advantages is that it is seeped with knowledge specific to the domain and thus any user who is conversant with the domain learns the language easily.

2. Easier Programming: In a DSL the level of abstraction is appropriate for expressing the domain application. Also, constructs and special functions are provided in order to help the users define applications of a specific domain. Therefore, the programs are easier to write and are usually more concise and readable than the ones in general programming languages.

   If the DSL is declarative then the user has to think of what to implement rather than how to implement. It is easier for the programmer to implement any algorithm. This makes the entire development process shorter and simpler. Specific optimization strategies are sometimes implemented in DSL compilers to enhance performance and also to systematize the code. This allows the user to get rid of all the complex optimization algorithms. Therefore the applications are easy, small and more maintenance friendly.

3. Systematic Reuse: The DSLs contain a large number of in-built functions but are usually quite restricted in terms of the variety available in program constructs. Hence, in a way the user is forced to use a lot of library functions. Also, most of the functions are explicitly parameterized, which enforces the usage style of the function. This helps the user to make use of the domain expertise that goes into the design of the DSL.

4. Improved Dependability: In contrast to a general programming language, the semantics of a DSL are often restricted to enforce check on certain properties, which are critical to any domain. DSLs help increase the testability of the code and hence produce more reliable code as compared to the general

programming languages. Thus, in a nutshell, DSLs enhance productivity, reliability, maintainability and portability.

5. Solutions easily interpreted: DSLs also allow the solutions to be in a format easily understood by the domain experts. This allows easy and correct interpretation of the solutions.
6. Validation and Optimization: DSLs allow validation and optimization at the domain level. Apostle a domain specific language for a range of device drivers is used to do static checking on the domain level and to determine applicability of optimizations. HaskellDB allows type check of every function before actual implementation and thus provides some handy axioms. We will see in later sections how we can use these axioms to enhance test generation.

3.4 Challenges of using DSL

In spite of the potential benefits, the use of DSLs is limited owing to the following factors:

- The costs of designing, implementing and maintaining a DSL.
- The costs of education for DSL users.
- The limited availability of DSLs.
- The difficulty of balancing between domain specificity and general-purpose programming language constructs.
- The potential for a tower of Babel, a potential language for every other domain.

3.5 DSLs and Model Based Testing

Software specifications written in a domain specific language are ideal candidates for test models in model based test automation. Since the specification captures the requirements at the level of abstraction appropriate for a domain, they help the users to express the requirements efficiently. Further, if the domain specific language is strongly typed or has in built consistency checks, the specification can be checked for various specification level errors like ambiguity, completeness, consistency etc. Thus the test cases derived from such a specification as a test model, have better coverage and more effective. Also domain specific languages import domain expertise into specifications. This allows the test generation process to generate test cases capturing requirements specific to the application domain.

This thesis demonstrates the use of domain specific languages in model based test generation. The principle is demonstrated through HaskellDB, a domain specific language for applications querying relational database applications. HaskellDB was designed as a domain specific language to produce type safe SQL queries. When used to write specifications of applications, the specifications can be type checked using the Hugs interpreter for Haskell. HaskellDB types capture information about the interfacing database and the inbuilt functions of HaskellDB that access the database conforms to the respective type constraints. This enables us to capture test cases specific to the database and its connections automatically.

3.6 Types and Functions of HaskellDB

HaskellDB is an embedded DSL derived from Haskell. Haskell is a functional language based on lambda calculus. [7] HaskellDB provides a finite set of operators and an optimum level of abstraction. The goal of HaskellDB is to guarantee a type

safe embedding of database queries and operations within Haskell. The underlying idea is that instead of sending plain SQL strings to a database, queries should be expressed with normal Haskell functions. Haskell's type system is then used to check the queries at compile time. Instead of getting a runtime error message saying that a field name doesn't exist, a type error is given at compile time that points to the location where the error might have originated. Queries are performed through the HaskellDB query monad which is a first-class value and can be stored in data structures, passed as argument or can take typed parameters.

3.6.1 HaskellDB Types

HaskellDB possesses embedded types that define the elements of a relational database. Each query generation operator is defined on these types. Unless the type specification of the database entries matches the input type of the query generator, one cannot define a legal query operation. Following are some of the embedded types in HaskellDB:

1. **Relation**: A relation groups together a collection of attributed values. It is represented by the abstract type `Rel` .
2. **Table**: Relational databases represent relations via tables, and HaskellDB defines a `Table` type that is parameterized over the type of the relation.
3. **Attributes**:A relation associates a collection of attributed values. In HaskellDB, attributes are first-class values, all of which have the `Attr` type.
4. **Expressions**: **Expr** is essentially an *abstract syntax tree* representation of possible SQL expressions. It is a data type whose values correspond directly to

SQL expressions. The role of the type parameter is analogous to that played by types in most programming languages. It prevents us from constructing **`Expr`** values that correspond to ill-formed SQL expressions.

Thus in HaskellDB a database or table of type **`Table r`**, consists of a collection of rows or relations, of type **`Rel r,`** where each column or attribute or field, of type **`Attr (Expr t),`** is named and contains a value, of type **`Expr t`**.

3.6.2 HaskellDB Functions

HaskellDB provides the user with a monad called the query monad[2] to build up a query or relational expression. It provides the following basic operations:

`data Query a` – *abstract data structure for SQL queries*

`returnQ :: a -> Query a` — *returns a query element*

`bindQ :: Query a -> (a -> Query b) -> Query b –` *binds two query --generating functions*

`table :: Table r -> Query (Rel r)`—*abstracts a table in the database*

`restrict :: Expr Bool -> Query()`— *restrict in relational DB*

`project :: r -> Query (Rel r)`—*project in relational DB*

By using a monad, HaskellDB code can then be phrased using Haskell's overloaded notation for monads (the "**`do`**" notation). Here's an example query:

-- *project out all the names from the phone book.*

```
names = do ph <- table phBookTable
           project ( ph ! name )
```

[2] A monad is a way to structure computations in terms of values and sequences of computations using those values, thus allowing the programmer to build up computations using sequential building blocks. In Haskell, monads are data types that encapsulate the functional I/O-activity, in such a manner that the side-effects of IO are not allowed to spread out of the part of the program that is not functional (imperative).

In order to construct the complex query operations the user has to use the combinators and operators provided by HaskellDB. Details of HaskellDB combinators and operators can be found in Appendix A.

3.7 HaskellDB Axioms and Domain Specific Requirements

Chays et al [4] argue that applications interfacing with databases need to satisfy some specific properties in order to make them function properly. These properties are significant only for database applications and hence may be called domain-specific properties. After a preliminary investigation we found that the most important database specific properties could be grouped under the following headings:

Req. Set 1. **Connection Specific Properties:** A vital property to ensure for all database applications is that a correct connection is established to the right database. Without a proper connection the application will fail either due to inability to handle query requests or due to inability to generate correct results for a query.

Req. Set 2. **Field Related Properties:** The database that an application is linked to, should consist of the appropriate fields. In other words, the application should not throw queries for non-existent tables or attributes and therefore, every attempt of doing so should be arrested. Such requirements are sometimes managed by the database management systems (DBMS), but it is equally important for the application to satisfy such requirements. This requirement becomes more important when the application interfaces with other applications.

Req. Set 3. **Type Related Properties:** It is also important for the application to throw queries of the right type. A string field in the database should not be treated as an integer field and vice versa. Any such attempt should be arrested by the application.

Req. Set 4. **Integrity Related Properties:** Any application which can possibly modify the database, is subject to integrity related properties. A change in the database should only be allowed in case the user has the privilege to do that. Sometimes data retrieval is also privilege specific. All such requirements are vitally important for a database application.

Req. Set 5. **Constraint Related Properties:** All insertions, deletions and updates performed by the application are subject to the following constraints:

a. Domain Constraints: These are the constraints on possible values assumed by any attribute.

b. Uniqueness Constraints: These are the constraints that prohibit multiple occurrences of certain attributes.

c. Referential Integrity Constraints: These are the constraints that ensure that the relation between various tables of the database is maintained.

d. Not Null Constraints: This is the constraint that prohibits a certain attribute from taking null values.

e. Semantic Integrity Constraints: These are the constraints that express constraints on the values of the attributes.

To be able to produce domain specific test cases the model needs to capture these requirements. Testers may not have access to the real database and even if they do, they may not be able to generate test cases that will address all the above requirements.

In a HaskellDB model of the system, the information about the database is captured through its type. A syntactically correct HaskellDB model ensures that the functions and the combinators used in the model access the correct fields of the database and that they do not perform operations violating a type constraint. Thus, any syntactically correct HaskellDB model can confirm certain properties of the software system. These we call the type axioms of the function. We can relate the domain specific properties specified above to individual functions based on the axioms they offer. Any instance of the functions in the specification thus asserts the domain specific properties via the axioms.

Our technique extracts these requirements (axioms) from a HaskellDB specification of the system and uses them to enhance the test model with domain specific test cases. The axioms are identified for every use of HaskellDB operators in the specification. For instance whenever the operator **`restrict`** (a relational database operator) is used in Haskell , it guarantees that the parameter is a boolean expression. This in turn demands that the relational operator for the Boolean expression has comparable variables on either side. When one side of the operator is a database field, extracted by the bang (!) operator we know that the fields accessed through it do exist in the database. In absence of such fields in the database, a HaskellDB specification will throw a type error.

Assume that there is a table in the database called 'Home" which is listed as

Name	Author_id	Title_pub
Carol	1	"Welcome to Software Testing"
Andy	2	"Functional Programming Rules"
Avik	3	"The world of Haskell"

Hence the table can be restricted to the second row with the **`restrict`** operation as shown below

```
x <- table Home
restrict(x!Name.==.constant"Andy")
```

The output should return a query for the second row that is

Andy	2	"Functional Programming Rules"

Thus if we follow the derivation process as described above, the axiom for the operator **`restrict`**, implies that the field 'Name' must exist in the table 'Home' otherwise the operation would not be possible. The domain specific requirement that builds on this axiom states that "*the application should always throw an error/ warning message that prevents formation of a query for the field 'Name' when it doesn't exist in the table 'Home'.*" Our algorithm extracts (see Chapter 4) all such domain specific requirements for every operator use and embeds corresponding states in the extended finite state model to enhance it. Table 3.1 lists a set of axioms derived from various types of HaskellDB functions and the requirements that we can associate on the basis of those axioms. This list is not complete and an exhaustive list can be found in Appendix B.

	Operator/ Function	Axiom	Associated Set of Requirements
1	Assignment Operator	The object on the RHS of the assignment operator must be defined.	Req Set 1, Req Set 2
2	Relational DB operators	All fields accessed by the operator must exist in the corresponding database.	Req Set 1
		The Boolean predicate for restriction and projection operation must be type consistent.	Req Set 3
3.	Connection Operators	Connection parameters are predefined and are of the right type.	Req Set 1
4.	Update Operators	The update operator does not violate any table type.	Req Set 4, Req Set 5
3	Comparison Operators	The compared fields on either side of the comparison operators must be comparable.	Req Set 3
4.	Boolean Connectors	The arguments must all be SQL Boolean expressions.	Req Set 3
5.	Bang/Extraction Operator	The field being extracted out of the relation argument must exist.	Req Set 2
6	Set Operators	The database arguments must be of the same shape (that is, each table has the same fields, and the corresponding fields store values of the same Expr type).	Req Set 4, Req Set 5
7	Arithmetic Operators	The arguments must all be arithmetic expression.	Req Set 3
8	String Operators	The arguments must all be string literals.	Req Set 3

Table 3.1: Axioms derived from HaskellDB

3.8 Summary

In summary it can be said that domain specific languages have the ability to provide the right level of abstraction and inbuilt functionalities to aid in specifying

applications of the particular domain. Strongly typed domain specific languages can be developed by embedding domain specific types and functionalities. These languages can be used to model systems for model based testing. By doing so, one can derive type related system axioms. These axioms can then be used to derive domain specific test cases.

3.9 References

[1] A. Deursen and P. Klint, "Little languages: Little maintenance?" *Journal of Software Maintenance*, 10:75-92, 1998.

[2] A. Deursen, P. Klint, and J. Visser., "Domain-specific Languages: An annotated bibliography", Net publication, http://www.cwi.nl/projects/dsl

[3] D. Bruce, " What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation".

[4] D. Chays, S. Dan, P.G. Frankl, F.I. Vokolos and E.J. Weyuker, "A Framework for Testing Database Applications," in Proc. ISSTA 2000, Portland, 2000, pp. 147-157.

[5] D. Leijen and E. Meijer, "Domain Specific Embedded Compilers," in *Proc.2nd USENIX Conference on DSL*, Austin, 1999, pp. 109-122.

[6] G. Arango., "Domain analysis: From art form to engineering discipline." *Fifth International Workshop on Software Specification and Design*, pages 152-159, May 1989.

[7] Hudak P., "The Haskell School of Expression", Cambridge University Press, NY, 2000.

[8] M. Antoniotti and A. Göllü, "SHIFT and SMART-AHS: A language for hybrid system engineering modeling and simulation." Ramming J. C., editor, *Proceedings of the USENIX Conference on Domain-Specific Languages*, Berkeley, CA, October 15-17 1997, pages 171-182.

[9] M. Van den Brand, A. Van Deursen, P. Klint, S. Klusener, and E. Van der Meulen, "Industrial applications of ASF+SDF", Wirsing M. and Nivat M., editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 9-18. Springer-Verlag, 1996.

[10] S. Chandra, B. Richards, and J. R. Larus, "Teapot: A domain-specific language for writing cache coherence protocols", *IEEE Transactions on Software Engineering*, 25(3), May/June 1999, pages 317-333.

[11] University of Massachusetts, CS 530 Lecture Slides, 2001.

Chapter 4 Derivation of a Structural Representation

In order to generate test cases satisfying a given test coverage criterion, it is necessary to derive a structural representation of the system model. The structural representation chosen in this case is called an extended finite state machine (EFSM)[3]. In this representation, every state represents a state of the system under test (SUT). The states are linked through transformations that signify change of state for the SUT. A state transformation can only be possible through triggering of a transition. A transition is called trigger-able if the system is at a state which can be transformed by the transition. The transitions can be constrained or un-constrained. Constrained transitions can only be triggered if the constraint is satisfied and the transition is trigger-able from the current system state. An un-constrained transition need not satisfy any constraint and can be executed as soon as the system reaches a state for which the transition is trigger-able. A group of states and transitions can be isolated as a model when none of the states in a group can be transformed to the states of the system outside the group. Introduction of models into the representation allows a hierarchical arrangement of states. Each model can represent a function at a higher level of abstraction and its child functions will then define the states of the model. A model can subsequently contain other models.

We define the EFSM representation of the system by a set $m0 = \{\Omega 0, \tau 0, St_{Start0}, St_{Exit0}\}$ where $\Omega 0$ is a set of states and sub-models in $m0$ whereas, $\tau 0$ is the set of transitions in $m0$ mapping each state or sub-model in $\Omega 0$ through various

[3] By choosing EFSM as the structural representation, commercially available tools for test design like TestMaster [1] could be used for test generation.

system transformations. St_{Start0} is the starting state and St_{Exit0} is the exit state for *m0.* A test case in such a representation of the system is a list of transitions that transforms St_{Start0} to the state $St_{Exit0.}$

The derivation of the Extended Finite State Machine is done in three different steps:

1) Derivation of an Extended Finite State Machine (EFSM) without Axioms or Predicates

2) Derivation of an Extended Finite State Machine, EFSMA, which accounts for Axioms and not the predicates

3) Derivation of an Extended Finite State Machine, EFSMAP, accounting for Axioms and Predicate

4.1 Derivation of EFSM

To derive EFSM we first need to introduce to the reader the three approaches one can use to write a specification in Haskell DB. The control flow and actual order of execution will then be extracted from these three specification styles.

4.1.1 Three styles.

HaskellDB specification can have the following three identified specification styles:

a) Specifying using the do-monad: Following is an extract of specification using do-monad.

F1=Do- {…

F2

F3

F4

…

}

This do-monad is used to specify the fact that one wishes to perform function *F2*, then *F3*, then *F4* whenever there is a call to function *F1*. In other words, functions specified with a do-monad will need to be performed in the order specified under the do. Also, *F2, F3* and *F4* are the children of function *F1* and are at a hierarchically lower level than *F1*. We denote a do- sequence of functions as *F1{SDO[F2,…,Fk]}* which implies that *F1* is the function specified using a do-notation and calls functions *F2…Fk* in sequence.

b) Specifying using Functional Composition: Functional composition is the second possible HaskellDB specification style. Following is an example of functional composition: *F1•F2•F3(..)* where • is the functional composition operator. This notation is used to specify that one wishes to first perform *F3(..)* , then *F2* on the result of that operation, i.e. *F2(F3(..))*, and finally *F1* on the result of *F2* and *F3*, i.e. *F1(F2(F3(…)))*.We denote a composition of functions as *SCOMP[F1,F2,…,Fk]* which implies that the function *Fk* will be called first. Function *Fk-1* will be called with the return value of function *Fk* and so on till function *F1* is called with the return value of function *F2*.

c) Specifying using a Sequential Juxtaposition of Functional Terms: Using the

third specification style, sequential juxtaposition, functions can be specified in any order. To then reconstitute the actual implied specification one has to understand in what order functions call each other. The following is an extract of a possible sequential juxtaposition specification:

F1 F4 F6

If Fi_c denotes the fact that function *i* is called, and if the completion of execution of function *i* is denoted by Fi_d, then the sequence of operations for the above extract of specification is given by, $F4_c\ F4_d\ F6_c\ F6_d\ F1_c\ F1_d$. In other words function *F1* is called with parameters which are the outputs of functions *F4* and *F6.* So prior to a call of *F1*, *F4* and *F6* need to be evaluated. HaskellDB supports a lazy binding of functions. This is taken into account while translating the test model to EFSM. We denote a sequential juxtaposition of functions *SSEQ[F1F2,…,Fj Fk..Fk+m,.., Fn].*

Any HaskellDB specification is a combination of do monads, sequential juxtapositions and compositions. For example consider the following specification:

```
query = do{ x <- table home
          ; restrict(x!name .==. constant John)
          }
show = search.execute
main = show query
```

It contains a combination of the three styles discussed above. For translation purposes, an embedded use of a specification style is denoted by explicit reference to the style. Thus the above specification may be denoted as

S =main{SSEQ[show{SCOMP[search, execute]} ,query{SDO[SSEQ [assign x SSEQ[table home]], SSEQ [restrict SSEQ[eq SSEQ[bang x name] SSEQ[constant John]]]]]}} .

4.1.2 Specification to Actual Flow

Hence taking a generic specification one can easily map it to the actual order in which functions should be executed.

Let us define an ordering function Order that will reorder the functions in *S* to generate a new specification *S'*. Within *S'*:

1. A function *F*, child function of function *G*, is denoted by *G{F}*

2. A function *F* executed after a function *G* is denoted by *{G;F}*.

The *Order* function orders the actual sequence of operation based on the specification style. The following defines *Order* in cases for the three specification styles and for their combinations:

Case I. For a do-notation we have *Order (S)=S'* where *S=SDO[F1…Fk]* and *S'* is the specification reordered by actual sequence of operation. Thus *S'={F1;F2;…Fk}.*

Case II. For a composition of functions, *Order (S) =S'* where *S=SCOMP[F1,F2…Fk]* and *S'={Fk;Fk-1; …F1;}.*

Case III. Similarly, for a sequential juxtaposition of functions we have *Order (S) =S'* where *S=SSEQ[F0 F1 F2,…Fk]* and *S'={F1;F2;…Fk;F0}* .

Case IV. The Order function is distributed over the different fragments of specification written in different styles contained in a complete specification. Thus if *S1[F1,…Fk]* and *S2[G1,…,Gn]* are two fragments of the complete specification

adhering to different styles (e.g., *S1* could be a sequential juxtaposition operator and *S2* could be a composition operator) then:

1. *Order(S1[F1,…,Fj{S2[G1,…,Gn]},Fj+1,…,Fk])=Order(S1[F1,,Fj{Order(S2[G1,…, Gn]}),Fj+1,,Fk]).*
2. *Order([S1[F1,….Fk]; S2[G1,…Gn]]) = { Order(S1[F1,….Fk]); Order(S2[G1,…Gn])}*

For instance, consider the specification introduced in section 4.1.1:

S = main{SSEQ[show{SCOMP[search, execute]} query{SDO[SSEQ [assign x SSEQ[table home]], SSEQ [restrict SSEQ[eq SSEQ[bang x name] SSEQ[constant John]]]]}]}

Then, the remapped specification is:

S'=Order(S)= main{Order(SSEQ[show{SCOMP[search, execute]} query{SDO[SSEQ [assign x SSEQ[table home]], SSEQ [restrict SSEQ[eq SSEQ[bang x name] SSEQ[constant John]]]]}])}

=main{query{Order(SDO[SSEQ [assign x SSEQ[table home]], SSEQ [restrict SSEQ[eq SSEQ[bang x name] SSEQ[constant John]]]]};show{Order(SCOMP[search.execute])}}

This subsequently reduces to:

=main{query{x; home; table; assign;x;name;bang;John;constant;eq;restrict};show{execute;search}}

4.1.3 Derivation of the EFSM.

Having obtained the reordered specification we proceed to develop the EFSM. *S'* is the basis for the EFSM. Let us define a function *StateMap* such that, *StateMap(S')= m0*, where *m0* is the desired EFSM and is defined as *m0= {Ω0, τ0,StStart0,StExit0}* as before.

In $\mathcal{S}'$, functions listed within a pair of curly brackets represent a group of functions, $\mathcal{G}i$, all at the same level of hierarchy in a fixed execution sequence. Each such group $\mathcal{G}i$ is translated into a sub-model mi defined as $mi = \{\Omega i, \tau i, StStarti, StExiti\}$.

StateMap translates each function $\mathcal{F}j$ in $\mathcal{S}'$ to a transition tj in the corresponding EFSM based on the following rules:

Rule 1. If $\mathcal{F}j$ is the first function in the group $\mathcal{G}i$ (e.g.,$\{\mathcal{F}j;\ldots\}$) then a transition tj is created from $\mathcal{S}tStarti$ to $\mathcal{S}tj$ such that $\mathcal{S}tj \in \Omega i$ and $tj \in \tau i$. This is because for each model the transition starts from $\mathcal{S}t_{Start}$ and also, execution of the first function in the group signifies the first transition of states in the model.

Rule 2. If $\mathcal{F}j$ is the last function in the group $\mathcal{G}i$ (e.g., $\{\ldots;\mathcal{F}j\}$) then a transition tj is created from the current state to St_{Exiti} and $t \in \tau i$. This justifies the fact that each group terminates with the execution of the terminal function and that for a model the transitions must end at $\mathcal{S}t_{Exit}$.

Rule 3. If $\mathcal{F}j$ is the function in the group $\mathcal{G}i\text{-}1$ that calls a group $\mathcal{G}i$, (e.g., $\mathcal{F}j\{\mathcal{G}i\}$) then a transition tj is created to the model m_i where $m_i \in \Omega i\text{-}1$, $tj \in \tau i\text{-}1$.

Rule 4. If $\mathcal{F}k$ follows $\mathcal{F}j$ in sequence in group $\mathcal{G}i$ (e.g., $\{\ldots;\mathcal{F}j;\mathcal{F}k;\ldots\}$) then a transition tk is created to $\mathcal{S}tk$ from $\mathcal{S}tj$ such that $\{\mathcal{S}tj, \mathcal{S}tk\} \subset \Omega i$, $tk \in \tau i$

Rule 5. If $\mathcal{F}k$ immediately follows the call of group $\mathcal{G}i$ in a group $\mathcal{G}i\text{-}1$ (e.g., $\{\ldots;\{\mathcal{G}i\};\mathcal{F}k;\ldots\}$) then a transition ti' is created to $\mathcal{S}tk$ from mi such that $\{mi, \mathcal{S}tk\} \subset \Omega i\text{-}1$, $ti' \in \tau i\text{-}1$

Any *S'* is uniquely defined by a set of functions *{F0,…Fn}* and an underlying control flow which is expressed through various combinations of sequences of functions and the embedded function calls. The function *StateMap* addresses all possible flows in *S'*, which are necessary and sufficient to derive EFSM from any *S'*. This also implies that the *StateMap* function is a unique mapping function that differentiates between specifications differing in either the set of functions or the underlying control flow.

Since the specification contains a finite number of functions, the number of transitions in the resulting test model is finite. This also ensures that the number of states in the state machine is finite. The special case of recursive calls is discussed later in this paper.

Consider, our running example :

S'=main{query{x; home; table; assign;x;name;bang;John;constant;eq;restrict};show{execute;search}}. We will have *StateMap(S')= m0= {Ω0, τ0,StStart0,StExit0}* where *τ0={query,show}* . (Figure 4.1)

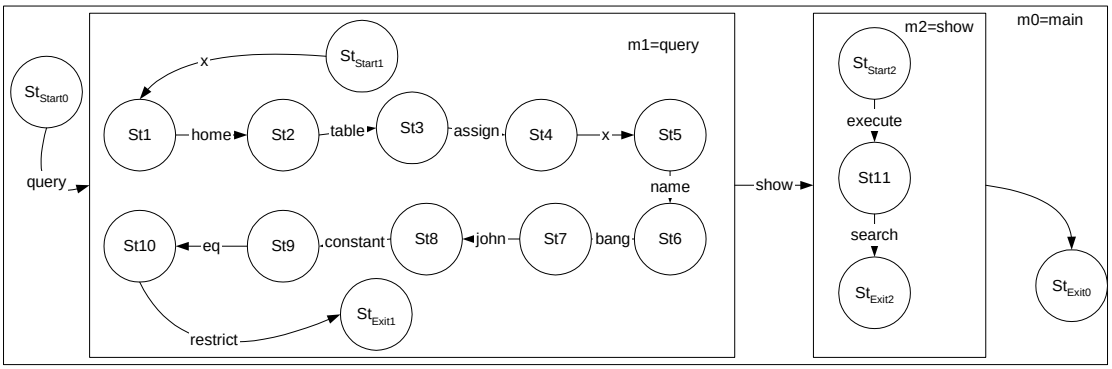

Figure 4.1: The derived EFSM

4.2 Derivation of EFSMA

The test model obtained in section 4.1 captures the requirements imposed by the initial specification S. To allow the test model to capture and test the implicit domain properties, it is necessary to embed additional states and derive EFSMA. To do so, parameters on which functions act need to be introduced explicitly. Indeed, axioms are properties that must be true of the parameters of a function. First let us consider the impact of explicit treatment of parameters on the three styles of specification defined in section 4.1.1.

4.2.1 Treatment of Parameters

In a functional language there is no difference between functions and their parameters. Functions can be passed on as parameters or parts of lists and other data structures. A parameter of a function is essentially the value returned by some other function. The arguments map uniquely to each parameter and the values are passed by reference. The values returned by functions are assigned to the parameters and they have the same type as the return type of the functions. Let $\mathcal{S}\chi$ denote a specification with explicit reference to parameters.

In the following, we define a function *ParameterMap*[4], which reorders the functions in $\mathcal{S}\chi$ while accounting for variables. $\mathcal{S}\chi'$ is the resulting specification.

To do so, parameters are categorized according to the following:

[4] *ParameterMap* is a composition of two operators: an operator that tags the variables in the specification according to their use i.e. d, c, and an extension of *Order*, that reorders the functions in the same fashion as *Order* while mapping variables used in function calls.

1. *Definition:* The parameter is said to be defined if the function using it associates a value and a type to the parameter. A definition of the parameter is denoted by x_d.

2. *Computation:* The parameter is said to be used in computation when the parameter is used by a function to define any other parameter. A computational use *of* x is denoted by x_c

In order to differentiate between a return value of a function and the call to a function we denote a return value of F as $\underline{F}$.

Considering the example specification in section 4.1.1 with the variables introduced we may have

Sx = main(maind)[SSEQ[show(showd, searchc)[SCOMP[search(searchd,execcutec), execute(executed, queryc)]] query(queryd)[SDO[SSEQ [assign(xd,tablec) x(xd) SSEQ[table(tabled, homec) home(homed)]], SSEQ [restrict(xd, eqc) SSEQ[eq(eqd,bangc,constantc) SSEQ[bang(bangd,xc,namec) x(xc) name(named)] SSEQ[constant(constantd, Johnc) John(Johnd)]]]]]]]

The remapped specification is then

S'_x=*ParameterMap* (S_x)

=*main*($\underline{main_d}$)[*query*($\underline{query_d}$)[*x*($\underline{x_d}$);*home*($\underline{home_d}$);*table*($\underline{table_d}$,$\underline{home_c}$);*assign*($\underline{x_d}$,$\underline{table_c}$);*x*($\underline{x_c}$);*name*($\underline{name_d}$);*bang*($\underline{bang_d}$,$\underline{x_c}$,$\underline{name_c}$);*John*($\underline{John_d}$);*constant*($\underline{constant_d}$,$\underline{John_c}$);*eq*($\underline{eq_d}$,$\underline{bang_c}$,$\underline{constant_c}$); *restrict*($\underline{x_d}$, $\underline{eq_c}$)]; *show*($\underline{show_d}$, $\underline{search_c}$)[*execute*($\underline{execute_d}$, $\underline{query_c}$);*search*($\underline{search_d}$,$\underline{execute_c}$)]]

4.2.2 Derivation of EFSMA

The first step in the process of embedding axioms is application of a function Filter that differentiates functions that belong to the standard HaskellDB library from functions that do not. Indeed these are the functions to which axioms are associated.

Let $S'_X = \mathcal{F}0(x_d)\{\ldots \mathcal{F}i(y_d, x_c)\{\ldots \mathcal{F}j\text{-}1(y_d, x_c);\ \mathcal{F}j(y_d, x_c);\ \mathcal{F}j{+}1(y_d, x_c)\ \ldots \mathcal{F}n(y_d, x_c)\}\}$ be an ordered specification such that $\mathcal{F}j$ is a library function, $Filter(S') = S_X''$ where $S_X'' = \mathcal{F}0(x_d)\{\ldots \mathcal{F}i(y_d, x_c)\{\ldots \mathcal{F}j\text{-}1(y_d, x_c)\ ;\&A0(y_d, x_c);\ \mathcal{F}j{+}1(y_d, x_c)\ \ldots \mathcal{F}n(y_d, x_c)\}\}$, $\&A0$ is an action representing the call to the standard library function $\mathcal{F}j$.

Thus for our example specification, S'_X in section 4.2.1

$S''_X = Filter(S'_X)$

$= main(\underline{main_d})\{query(\underline{query_d})\{x(\underline{x_d}); home(\underline{home_d}); \&table(\underline{table_d}, \underline{home_c}); \&assign(\underline{x_d}, \underline{table_c}); x(\underline{x_c});$

$name(\underline{name_d}); \&bang(\underline{bang_d}, \underline{x_c}, \underline{name_c}); John(\underline{John_d}); constant(\underline{constant_d}, \underline{John_c}); \&eq(\underline{eq_d}, \underline{bang_c}, \underline{constant_c}); \&restrict(\underline{x_d}, \underline{eq_c})\}; show(\underline{show_d}, \underline{search_c})\{execute(\underline{execute_d}, \underline{query_c}); search(\underline{search_d}, \underline{execute_c})\}\}$

Having derived S''_X, the reordered specification with embedded parameters and identified library functions, we proceed to extract the EFSMA. We now define a function $StateMap_X$ such that $StateMap_X(S''_X) = m0_X$ where $m0_X$ is our EFSMA with parameters and is given by $m0_X = \{\Omega 0, \tau 0, St_{Start}, St_{Exit}, V0\}$. $StateMap_X$ is an extension of $StateMap$ such that $m0_X = m0 \cup \{V0\}$. $V0$ is the set of variables for states in the set $\Omega 0$. If Vti_0 is the set of variables in transition ti_0 (i.e. the variables used in the corresponding

function) then $\mathcal{V}_0 = \mathcal{V}t1_0 \cup \mathcal{V}t2_0 \cup \ldots \cup \mathcal{V}tn_0$, and $\{t1_0, t2_0, \ldots, tn_0\} = \tau 0$. Similarly we derive $\mathcal{V}$ for the sub-models in $m0_x$.

Thus, for our example specification we have $StateMap_x(S''_x) = m0_x = \{\Omega 0, \tau 0, St_{Start}, St_{Exit}, \mathcal{V}0\}$ as presented graphically in Figure 4.2.

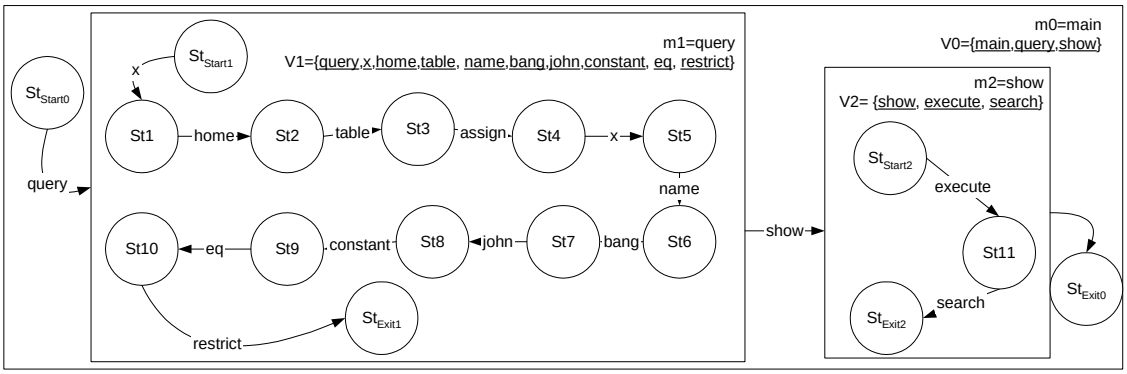

Figure 4.2: The EFSM with embedded variables

4.2.3 Embedding the Axioms

Having introduced the variables in the test model, we proceed to embed the related axioms into $m0_x$ to derive EFSMA. Each call to a HaskellDB library function associates one or more properties with the variables on which the function is acting. As discussed before all calls to the HaskellDB library functions are represented by actions in the filtered specification S''_x. These properties were derived in Chapter 3 and are listed in Appendix B. Thus each action in S''_x possesses a set of properties which hold true for the parameters on which it operates (parameters in computational use). These properties can either be satisfied by the implementation or not. The test model should test each case. Thus two additional states in the test model need to be created. The first one will generate a test case which does satisfy the axiom, hence a

normal functioning of the system. The second one generates (if possible) a test case that violates the axiom and will allow us to verify whether the application is protected against this violation or not.

Let X represent the set of variables on which a library function Ai acts in computational use. Thus for each $\mathcal{A}i(X)$ in $\mathcal{S}x''$ a set of properties given by $\mathcal{P}i(X)=\{p1_i(X),\ldots pn_i(X)\}$ hold. Let us define a function *TestProperty* such that $TestProperty(m_{\mathcal{X}}, \mathcal{P}_i(X)) = \tau_i^x$., where $\tau_i^x = \coprod_{j=1}^{n} \{t_j, \bar{t}_j, t_{j,Exit}\}$ and $m_{\mathcal{X}}$ represents the test model or any of the sub-models. If $St_{i,j}$ is the system state where the property pj_i for action $\mathcal{A}i$ holds and $St'_{i,j}$ is the sate where it does not hold, then t_j is a transition to the state $St_{i,j}$ from the state $St_{i,j-1}$ and $\bar{t}_j$ is a transition to the state $St'_{i,j}$ from the state $St_{i,j-1}$. Further, $t_{j,Exit}$ is a transition from $St_{i,j-1}$ to St_{Exit}. Also we have, $\Omega_i^x = \coprod_{j=1}^{n} \{St_{i,j}, St'_{i,j}\}$.

If there are n transitions representing the actions $\mathcal{A}1,\ldots\mathcal{A}n$ in $m_{\mathcal{X}}$, we define $\tau_{\mathcal{A}xioms} = \coprod_{i=1}^{n} \tau_i^x$ and $\Omega_{\mathcal{A}xioms} = \coprod_{i=1}^{n} \Omega_i^x$ [5].

EFSMA is then obtained easily by defining a function *AxiomMap* such that if $m_{\mathcal{X}}=\{\Omega, \tau, St_{Start}, St_{Exit}, V\}$, then $AxiomMap(m_{\mathcal{X}}) = m_{\mathcal{X}}'$, where $m_{\mathcal{X}}'=\{\Omega', \tau', St_{Start}, St_{Exit}, V\}$; $\tau' = \tau \cup \tau_{\mathcal{A}xioms}$ and $\Omega' = \Omega \cup \Omega_{\mathcal{A}xioms}$. Figure 4.3 depicts the effect of introducing the axioms into the test models.

[5] The set of properties for a given library function is finite. Therefore the number of states added is finite and EFSMA remains a finite state machine.

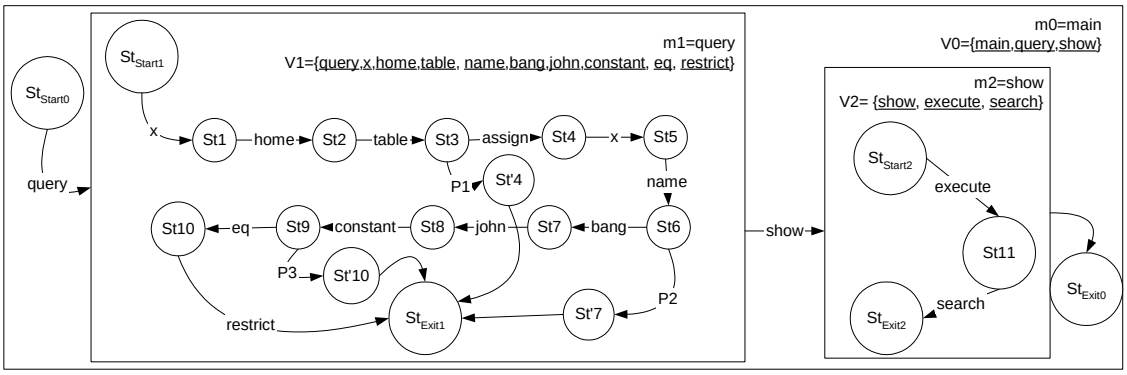

Figure 4.3: The EFSM with axioms introduced, EFSMA.

4.3 Specification with Conditional Flow

HaskellDB Specifications contain conditional calls of the functions. Conditional flow can exist only in a *SSEQ* or *SDO* specification.

In the following we extend the function *ParameterMap* to account for the conditional flow. A conditional flow in HaskellDB can be created in via three possible constructions. For each conditional constructor the function *ParameterMap* introduces a condition *Ci (X)* to the corresponding specification fragments. *Ci (X)* can be uniquely described by the variable *X* and the constraint associated with it. If the constraint is satisfied by the variable, then *Ci (X)= True* else *Ci (X)=False*. The constraint is composed of a relational operator and a constraint value. Hence, we formally represent a condition as:

P(X) = (X RO Value) where *RO ϵ {.==.,!=., .>=.,. <=.,.>.,.<.}*

ParameterMap extracts the condition for each conditional construction in HaskellDB as described below. Let *S0* represent the example specification, which has specification fragments *S1, S2,… Sn*, each adhering to any one of the three

specification styles described as defined in section 4.1.1 Also, let X represent the set of variables, x represent any particular variable and x_i represent any value assumed by the variable x.

1. **If then else:** The if-then-else construct of HaskellDB is the counterpart of the if-then-else in procedural languages. The general syntax for if-then-else is given as :

   if (Pi(X)) then (Consequence) else (Alternative)

   Pi(X) is a boolean expression on X that can either be True or False. The Consequence and the Alternative are HaskellDB specification fragments. The variable used in the predicate needs to be defined before its use.

   Let us consider an example specification, *S0* with an if-then-else construct such that:

   S0=SSEQ[F0(x){ if (P1(x)) then S1[…] else S2[…]}]

   Thus *ParameterMap (S0)=S0'* such that

S0'=F0(x_d){C1(x)=>ParameterMap(S1[…])}{C2(x)=>ParameterMap(S2[…])}:C1(x)=P1(x),C2(x)= $\overline{P1(x)}$.

2. **Case constructor:** The case constructor of Haskell has the following syntax:

   Case (X) of
   (Value 1) -> Consequence 1
   (Value 2) -> Consequence 2
   …
   (Value n) -> Consequence n

   Value i is a possible value set assumed by the set of variable X. A value of '_' signifies the default value of the set of variable. The consequences have to be a valid HaskellDB specification fragment. Now consider an example specification with an explicit reference to the case constructor.

   S0 = SSEQ[F0 (x) { Case (x) of

(x_1) -> S 1[…]
(x_2) -> S 2[…]
(_) ->S3[…]
}]

Thus $\mathcal{P}arameterMap\ (S0) = S0'$ such that

$S0'=\mathcal{F}0(x_d)\{C1(x)=>\mathcal{P}arameterMap(S1)\}\{C2(x)=>\mathcal{P}arameterMap(S2)\}\{C3(x)$

$=>\mathcal{P}arameterMap(S3)\}$, $C1(x)=(x.==.\ x1)$, $C2(x)=(x.==.x2)$ and

$C3(x)= (\overline{C1(x)}\ \text{AND}\ \overline{C2(x)}) = \overline{C1(x)}\ \text{OR}\ \overline{C2(x)} =((x!=x1)\ OR(x!=x2))$

3. **Pattern Matching:** Pattern matching is a unique way for introducing conditional flow in Haskell specifications. If a function $f\ (X)$ is described using a pattern matching fragment, then the function sequentially scans for matching patterns on X until one is found. This is a way of overloading functions in Haskell. In case of ambiguity the first available pattern is adopted. Pattern matching can bear any of the following two allowable syntaxes in Haskell.

- $\mathcal{F}\ (x)\{$
  $|\mathcal{P}attern(x)_1 \rightarrow Consequence\ 1$
  $|\mathcal{P}attern(x)_2 \rightarrow Consequence\ 2$
  …
  $|\mathcal{P}attern(x)_n \rightarrow Consequence\ n$
  $\}$
- $\mathcal{F}\ (\mathcal{P}attern(x)_1) = Consequence\ 1$
  $\mathcal{F}(\mathcal{P}attern(x)_2) = Consequence\ 2$
  …
  $\mathcal{F}(\mathcal{P}attern(x)_n) = Consequence\ n$

  Again, the consequences need to be valid HaskellDB specification fragments.

$\mathcal{P}attern(x)$ is an abstract way of representing the list of parameters and their types. It transforms a parameter pattern into a regular constraint with a relational operator and a constrain-value defined on the variable x. Let us consider an example specification with an explicit reference to the pattern matching constructor.

$S0 = SSEQ[\ F0\ (x)\ \{$
$\quad |\ Pattern(x_1) \rightarrow S\ 1[\ldots]$
$\quad |\ Pattern(x_2) \rightarrow S\ 2[\ldots]$
$\quad \}]$

Thus $ParameterMap(S0) = S0'$ such that $S0'=F0(x_d)\{C1(x)=>ParameterMap(S1)\}\{C2(x) => ParameterMap\ (S2)\}$, $C1(x)=Pattern(x)_1$, and $C2(x)= Pattern(x)_2$

4.3.1 State Machine

The function Filter is then applied to the re-ordered specification to differentiate the functions that belong to the standard HaskellDB library from the functions that do not. We then apply the $StateMap\chi$ function to the re-ordered specification. EFSMAP is an extension of the EFSMA notation for *TestModels* with each transition also characterized by a governing predicate. The default value of these predicates is set to NULL. The $StateMap\chi$ function is extended to handle the specification while accounting for the conditional flow by adding a rule 6 and also by modifying rule number 3 in order to account for functions passing control to multiple groups:

Rule 3 (modified): If Fj is the function in the group Gi that calls k groups $G_{i+1},..G_{i+k}$, (e.g., $Fj\{G_{i+1}\}\{G_{i+2}\}\ldots\{Gi_{+k}\}$)then k transitions $t_{j1}, t_{j2},\ldots t_{jk}$ are created to the models $m_{j+1},m_{j+2},\ldots,m_{j+k}$ where $\{m_{i+1},m_{i+2},\ldots,m_{i+k}\}\subset\Omega i$, $\{ t_{j1}, t_{j2},\ldots t_{jk}\}\subset\tau i$.

Rule 6: If group Gi is initiated on condition $Cj(x)$ being True and if tk is the transition that corresponds to initiation of Gi, then $tk.predicate=Cj(x)$.

Let S'' , be the reordered specification accounting for the variables and for the conditional branching of functions with identified library functions.

Thus, if we have $S'=\{F0(x_d)\{C1(x)=>F1(y_d, x_c)\ ;F2(z_d, y_c)\ \{A3(y_d, zc)\}\}\{C2(x)=>F3(x_d, y_c)\}\}$, then $TestMap_x(S''_x) = m0_x=\{\Omega 0, \tau 0, St_{Start}, St_{Exit}, V0\}$ as presented graphically in Figure 4.4.

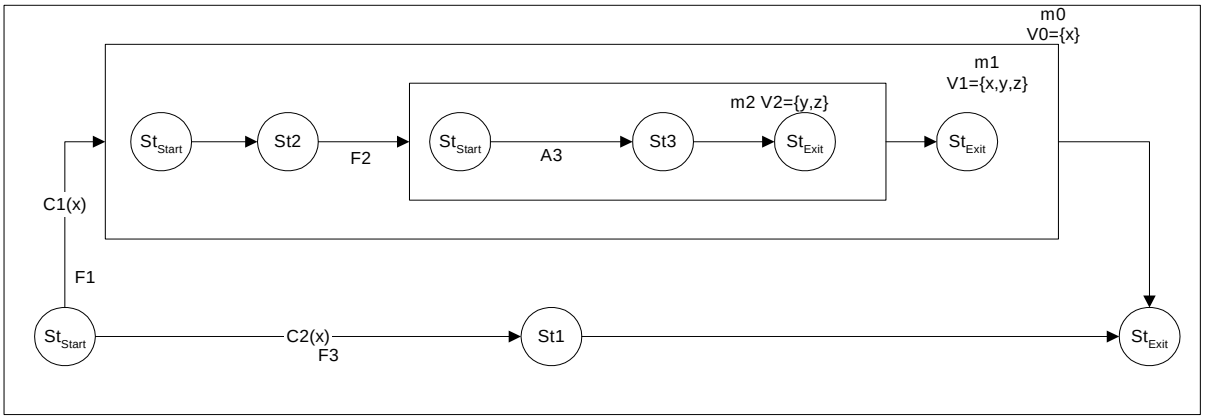

Figure 4.4: EFSM with the embedded variables

The test model derived above can be enriched with domain specific test cases as before by the application of the function *Axiom*.

4.4 Special Case of Recursion

Recursion is the only means of introducing the repetitive tasks in HaskellDB. Recursions are characterized by the base case definition and the recursive relation. A recursion can not be modeled explicitly into state machines primarily because the number of states and transitions depend on the input to the function and hence is not predetermined for the test models. Consider the following example for calculating factorials by recursion:

Fact 0= 1
Fact x= x Fact x-1*

This can be written in a more generic form as :

SSEQ[F1 (x){

|x.==.x1->y1 base case
|x.!=.x1->SSEQ[F2(x)[SCOMP[F1,F3(x)]]] recursive rule
}

where *F1=Fact*, *F2= multiplication operator* and *F3= unary subtractor*. Here the number of calls to *F1* depends on the value of *x* and hence with the input varying, the number of states in the model will vary. *F3* determines how *x* is modified with each recursive call. The recursion will halt iff $F3^n(x) = x1$ for some finite *n*. (Termination Rule) which is true for this case as $F3^x(x)=0=x1$.

Now, suppose we need to validate *F1(x)*. Let us consider *x2* such that $F3^n\ x2=x1$. If by assuming that *F1 F3(x2)* is correct, we can validate *F1(x2)* then by induction we can say that *F1(x)* is validated, provided that the base case, *F1 (x1)= y1*, is true.

In order to keep the number of states finite and pre-determined, a similar technique is adopted for test modeling. It can be assumed that for a correct HaskellDB specification the termination rule is satisfied (or else the static checker will throw an error). The test-model models the base case and the recursive rule. If a function is correct for the recursive rule and the base case, then it can be said that the function is correctly implemented for any input. The recursive rule can be validated by modeling the function for one iteration. A test suite for testing iteration must exercise the base case at least once.

For example for the above specification we have the test model as follows:

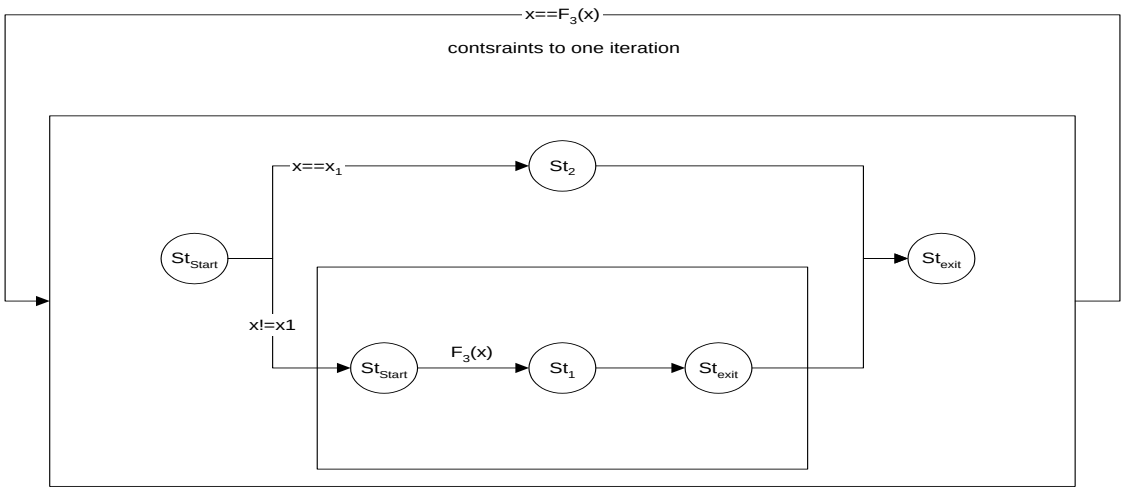

Figure 4.5: Model Depicting Recursion

4.5 Summary

There are three possible functional flows in Haskell (and similar higher ordered typed languages). By identifying the flows it is possible to embed states corresponding to every call and return of the functions. The states can be grouped hierarchically by identifying the hierarch in function calls. Thus a functional specification can be translated into an extended finite state machine representation of the system. The EFSM notation can then be enriched by embedding variable and predicates. Additional states can also be embedded in the EFSM to account for domain specific axioms.

4.6 References

[1] *Test Master User's Guide*, Release 1.9.5, Empirix Inc., New Hampshire, 1999.

Chapter 5 Tool Support for HOTTest

This chapter describes the tools supporting test generation using HOTTest. The primary components of HOTTest's implementation are identified and described in detail. We also show screenshots from various phases of generation of test cases from HaskellDB models. The translation process is described using a model for a small search program (SSP).

Figure 5.1 presents the primary tools that implement HOTTest.

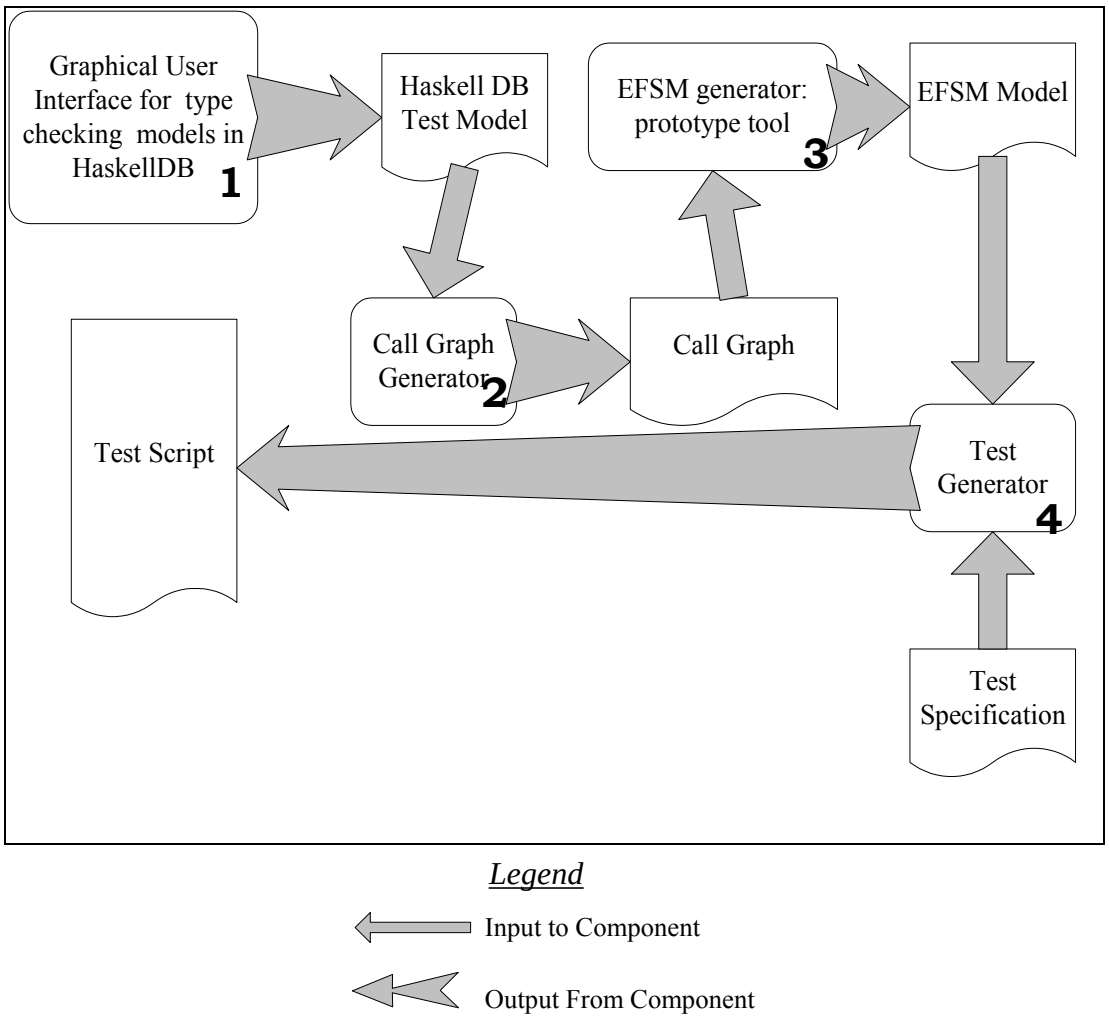

Figure 5.1: The Architecture of HOTTest

The user manually describes the system using any text editor like "Notepad[6]". The graphical user interface, Hugs is used to type check the HaskellDB specification. The type correct HaskellDB specification is the input to the HaskellDB parser. The parsed tree produced by the parser is used to generate a call graph. The call graph lists the call sequence of the functions and also lists the parameters passed and returned during the calls to the function. The call-graph is then used to generate the EFSM and embed the related domain specific axioms. The EFSM is finally imported to TestMaster which is a commercial tool for test case generation from EFSM models of systems.

5.1 Modeling of SSP

SSP is an application for generating queries for the PUBS database. PUBS is a database created in MS Access with information about authors and their publications. The database has three tables named authors, titles and titleauthor. SSP is required to generate and execute queries on PUBS. SSP generates the queries based on the search options specified by the user. The user may opt to search by first or last name of the author, by the author's city, or by the title of the book written by the author. User is then supposed to provide the search string for the selected option, viz. name of the author, or city of the author or the title of the book. If an entry corresponding to the query exists in PUBS, the application returns the Name of the author(s) and their respective publication(s) or else the application returns a message saying "No such entry".

[6] "Notepad" is a text editor available with the Windows O/S. Other text editors can be used for modeling in HaskellDB, if "Notepad" is not available.

Figure 5.2. depicts a screenshot of SSP application implemented using Visual C++. The GUI is implemented using dialog boxes of the Microsoft Foundation Classes (MFC).

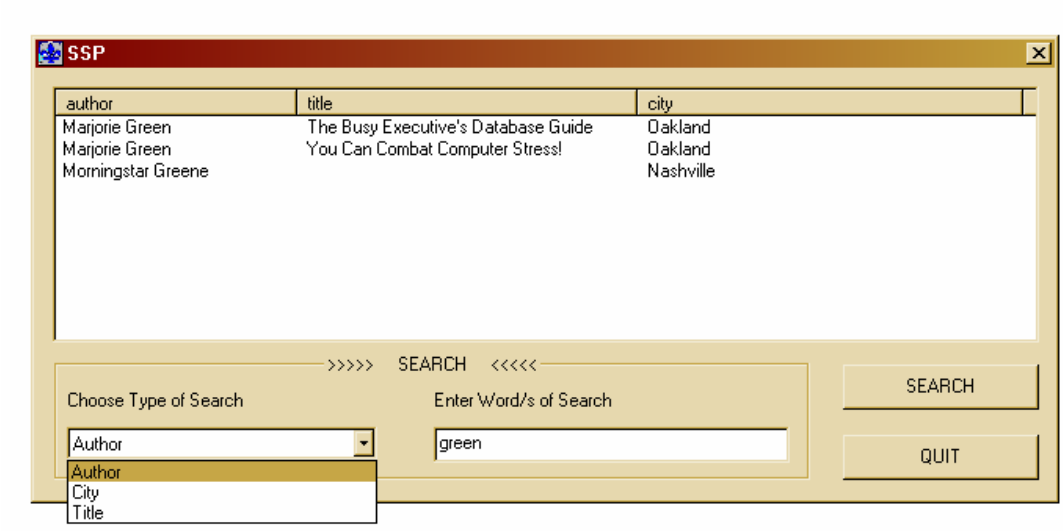

Figure 5.2: A screenshot from SSP Application

To model an application in HaskellDB, one can use any text editor like notepad.exe. The model is an executable specification of the application under test. To enable database specific type checking and to access the domain specific functions one has to import the module named HaskellDB into the specification. Figure 5.3 presents the model for SSP in HaskellDB.

```
module Search_example where
import Trex
import HaskellDB
import Ado
import Pubs


search sf q = doQuery (printQ sf) q

printQ sf = putStr . unlines . perform sf

perform f [] = ["No such entry"]
perform f rs = map f rs

doQuery action q
            = adoRun $
```

```
        adoConnect (adoDSN "Pubs") $ \pubs ->
        do{ rows <- query pubs q
          ; action rows
          }

main = do putStr "Choose search criterion:\n\t(1) author\n\t(2) city\n\t(3) title\n\t(q) quit \nMake your choice: "
                  choose

choose = do choice <- getChar
          case choice of
               '1' -> do doAuthor
                         main
               '2' -> do doCity
                         main
               '3' -> do doTitle
                         main
               'q' -> return ()
               _  ->  do putStr "\nNo such choice; try again!\n"
                         main


doAuthor = do putStr "\n Which author name do you want to search for? "
            name <-getLine
            search showResult (authorQ name)

doCity   = do putStr "\n Which city do you want to search for? "
            name <-getLine
            search showResult (cityQ name)

doTitle  = do putStr "\n Which title do you want to search for? "
            name <-getLine
            search showResult (titleQ name)

authorQ name = do{ x <- table authors
                         ; y <- table titleauthor
                         ; z <- table titles
                         ; restrict ((x!au_lname .==. constant name) .||.(x!au_fname .==. constant name))
                         ; restrict (y ! title_id .==. z!title_id)
                         ; project (au_fname = x!au_fname .++. x!au_lname, city= x!city, title = z!title)
                         }

cityQ name = do{ x <- table authors
                         ; y <- table titleauthor
                         ; z <- table titles
                         ; restrict (y ! title_id .==. z!title_id)
                         ; restrict (x!city .==. constant name)
                         ; project (au_fname = x!au_fname .++. x!au_lname, city= x!city, title = z!title)
                         }


titleQ t
             = do{ x <- table authors
                ; y <- table titleauthor
                ; z <- table titles
                ; restrict (x!au_id .==. y!au_id)
                ; restrict (y!title_id  .==. z!title_id)
                ; restrict (like (z!title) (constant ("%" ++ t ++ "%")))
                ; project (au_fname = x!au_fname .++. x!au_lname, city= x!city, title = z!title)
                }

showResult r
             =  "Name =" ++ r!. au_fname ++ "City =" r!.city ++ "Title =" r!.title
```

Figure 5.3: HaskellDB Model for SSP

5.2 Component 1: Hugs Interpreter

At the front end we have Hugs, a Graphical User Interface for validating models in HaskellDB. Hugs is an interpreter for Haskell, available freely for non-commercial use. Hugs supports static type checking and can also be used to execute the specification. The HaskellDB type library needs to be loaded before Hugs can check models for HaskellDB types. Hugs can track the errors and provides suggestions for the modeler for possible error locations. An example screenshot of the Hugs interface is depicted in Figure 5.4.

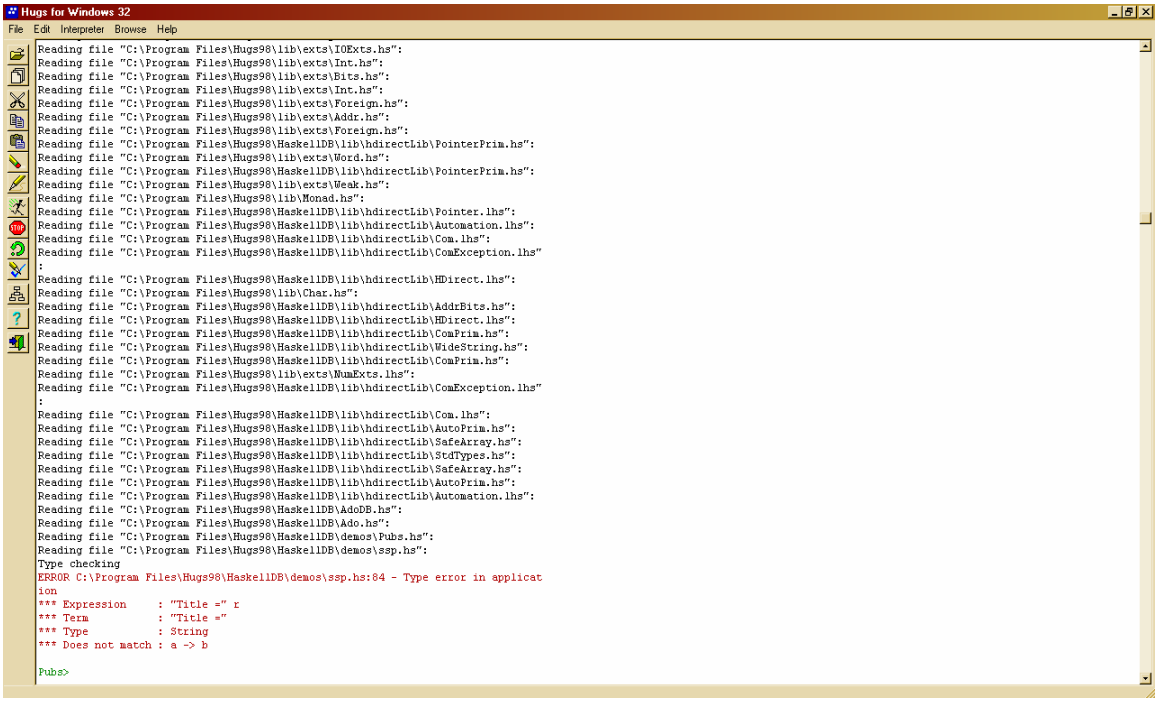

Figure 5.4: Screenshot Depicting Hugs Interface

5.3 Component 2: Call Graph Generator

The callgraph generator component is composed of a HaskellDB parser and a call graph generator tool. The HaskellDB parser is implemented using HSparser, a parser generator tool (e.g.,YACC for C++) written in Haskell. The parser takes a

syntactically correct HaskellDB specification and generates a parse tree for it. The parse tree is analyzed using a Haskell module called CallGraph.hs that outputs the call sequence of the functions and the parameters that are passed and returned during the call of the functions.

The algorithm for CallGraph.hs (Depicted in Figure 5.5) distinguishes three kinds of declaration: IO procedures, HaskellDB queries, and everything else. This allows it to concentrate on those declarations of interest. The 'main' function first labels each node in the parse tree with its declaration type (IO procedure, query, other), and then extracts from those nodes the monitored variables (those whose initial values can affect the control flow) and controlled variables (those whose final values constitute the output of the program). The monitored and the controlled variables together constitute the set of "important variables". Given a set of "important" variables, and a set of labeled declarations, the module CallGraph.hs produces a graph representing the control flow for those declarations. It does this by considering each node in turn, and constructing a control graph node containing the following information:

- The name of the node
- A list of its input variables (built from the parameters of the declaration, and any of its free variables which are also "important" variables).
- A list of output variables
- The control flow emanating from this node. This can be one of:
    - Go to <node>, with parameters <ps>.
    - Perform IO action.

- Branch on <name>, with possibilities given as pairs of values
- Exit this node.

```
CallGraph(parseTree P)
{
        flowGraph f = empty;
        P, f = mark(f,P);
        develop_flow_graph(f,p);
}

mark(flowGraph f, parseTree P)
{
        for each node n in P
        {
                if (n.type = IO)||(n.type = Query)
                        {
                                n.important = TRUE;
                                node node_in_flowGraph = new node;
                                node_in_flowGraph.input = n.parameterlist;
                                if (n.type = IO )
                                        node_in_flowGraph.output = Empty;
                                if (n.type = Query)
                                        node_in_flowGraph.output = Rows;
                                f.addNode(node_in_flowgraph);
                    }
                else
                        n.important = FALSE;
        }
        return P, f;
}

develop_flow_graph( flowGraph f, parseTree P)
{
        for each node n in f
        {
                node declaration_in_parseTree = P.getNode(n);
                for each functionCall f in declaration_in_parsetree
                {
                        if (f = bindingAction)          //binding action is definition of local variables in
terms
                                                        //of inputs or other local variables
                         {   n.actionList.add(f);
                             n.localVariables.add(f.return);
                          }
                        if (f = non_bindingAction);     //non binding action is the action where no local
                                                        //variable is defined. Only such action in our
                                                        // declarations of interest is an output action
                           n.actionList.add(output_action);
                        if (f = association)             // association is renaming of variables.
                           do nothing;
                        if (f = final_action)            // Signifies a termination of a do_sequence.
Usually
                                                        //termination,branching or a jump.
                            if declaration_in_parseTree for f(X) is important
                                    n.add(Goto G with X);
                            if f.type = variableInvocation && variable.scope =global
                                    n.add (Goto Variable);
                            if f.type = conditional && predicate has localVariables
                                    n.add ( branches on);
                }
        }
}
```

Figure 5.5: Algorithm for Call Graph Generation

A screenshot of an output from the module CallGraph.hs is depicted in Figure 5.6.

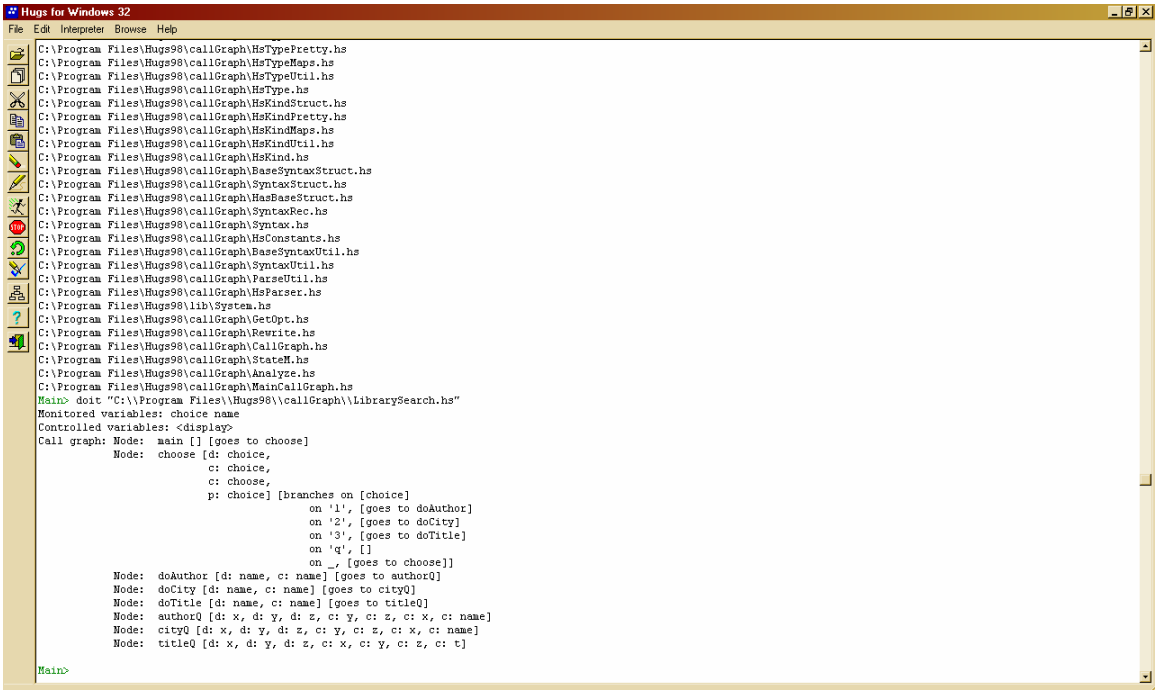

Figure 5.6: A Screenshot from Call Graph Generator

5.4 Component 3: EFSM Generator:

EFSM generator is a prototype tool that implements the methodology discussed in the previous chapter. The EFSM generator has four main modules as follows:

1. ParameterMap: This module implements the algorithms for ordering functions and extracting the parameters.
2. Filter: This is the module that identifies and isolates the HaskellDB standard library functions.
3. TestMap: This module translates the specification into the extended finite state machine.

4. AxiomMap: It is the module that embeds states to capture domain specific requirements on the basis of the HaskellDB axioms.

The EFSM generator generates the EFSM representation of the model in accordance to an API for the commercial test generation tool. Figure 5.7 shows a flowchart for the EFSM generator component of HOTTest.

Figure 5.7: A flowchart for EFSM generation

The prototype tool generates the EFSM in "tmi" format which corresponds to the API provided in TestMaster [1]. Files in "tmi" format can be readily uploaded to TestMaster for test case generation.

5.5 Component 4: Test Generator

The test generator is the component that derives test cases on the basis of the generated EFSM. The test generator is a commercial test generation tool (TestMaster [1]). Apart from the generated EFSM, the test generator takes as input a *test specification*. A test specification contains information on *test objectives*, *test profile* and *test constraints*. The test generator performs a depth first search on the test model for the exit states starting from the entry state of the main model on all enabled edges. An edge is said to be enabled if the test constraints are satisfied before traveling the edge. The test constraints contain constraints based on profile, number of iterations(in case of recursion) and behavioral constraints. Each edge of the EFSM is enriched with information called TestInfo. This is the information that contains the script for a certain test harness (in our case it was WinRunner[2]). While generating the test case the test generator concatenates the test info stored in these edges in sequence to produce test scripts in accordance to a certain coverage criteria.

5.6 Summary

In summary, to generate test cases using the HOTTest technique one needs to use four independent components, viz. Hugs Interface, Call-Graph Generator, EFSM generator and TestMaster, an EFSM based Test Case Generator. Prototype components have been implemented for the call-graph generator and the EFSM generator. Hugs is freely available for non-commercial use and TestMaster is a commercial tool by Empirix Inc.

5.7 References

[1] *Test Master User's Guide*, Release 1.9.5, Empirix Inc., New Hampshire, 1999.

[2] *WinRunner User's Guide,* Version 7.01, Mercury Interactive Inc., Sunnyvale, CA, 2001.

Chapter 6 Experimental Validation of Usability and Performance

To be industrially viable, a test generation technique should excel both in performance and usability. Since, HOTTest uses a functional domain specific language with limited use in the past, it is uncertain how will it compare to usability of other test generation tools. Programmers of imperative languages are not conversant with the functional style of specification and may find it difficult to use HaskellDB to define the systems. Also, it is argued that because of its syntax and its type system, Haskell is hard to learn. Since, HaskellDB is derived from Haskell and uses the same syntax, it is feared that it will raise similar concerns. Therefore, there is a need for a study that characterizes the use of functional languages for system modeling and thereby provides empirical support for HOTTest's usability. Additionally, we also seek empirical evidence to support claims with regards to performance of HOTTest.

In this chapter an in-vitro experiment is designed that studies the usability of HOTTest by comparing it with another model based test design technique. This study will identify the advantages and disadvantages of using Higher Ordered Typed specification languages for model based testing of applications. The model based test design technique that is chosen as the basis for comparison is the Extended Finite State Machine (EFSM) based test design technique. This technique was chosen after an extensive survey of test design techniques used in the software industry. It was found that Extended Finite State Machine (EFSM) is a very popular technique for modeling state-based systems including web-applications, database query systems,

computer communications, industrial control system, etc.[5][11][12] Tools like TestMaster [1] use EFSM based test models for generation of test cases. EFSM based test generation is widely used in the telecommunication industry for automatic test case generation and requirements tracking. Both HOTTest and EFSM based techniques are functional test design techniques and they have tools to support test generation. Further, both of these techniques work with behavioral representation of the system. Therefore, EFSM based test generation lends enough similarity in the modeling process for comparison.

6.1 The Experiment Design

6.1.1 The Research Question

In this chapter a formal investigation is carried out to answer the following research questions:

- Q1- Is the HOTTest based test generation technique comparable in usability to the model based test generation technique using EFSMs?
- Q2- Is the test suite generated using the test model of HOTTest superior in performance than the one generated using the EFSM model of the system?

6.1.2 Variables

The independent, controlled and dependent variables for the experiment are as follows:

- Independent Variable- The independent variable is the test modeling technique used. The experiment groups used either the HaskellDB specifications or the EFSMs for modeling the application.

- Controlled Variable- The controlled variable is the knowledge and experience of the students and it is measured on an ordinal scale.
- Dependent Variable- The dependent variables are performance and usability indicators of the test design techniques and performance indicator of the test models. Some of them are direct measures and the others are calculated using some other direct measures. (See Table 6.1.)

No.	Indicator of	Dependent Variable	Type	Symbol
1.	Usability	Learning	Indirect	***Learn***
2.	Usability	Ease of Learning	Indirect	***Eff***
3.	Usability	Errors	Direct	***EI***
4.	Usability	Satisfaction	Direct	***Sat***
5.	Usability	Ease	Direct	***Ease***
6.	Performance	Effectiveness	Indirect	***EffectP***
7.	Performance	Efficiency of Model	Indirect	***EffP***

Table 6.1: Dependent Variables of the Experiment

6.1.3 Measurement Models

The measurement-models used for measuring performance indicators of the models and usability indicators of the techniques are described below.

6.1.3.1 Usability

Traditionally the measures of usability [3], [9] are defined for software applications or more specifically for their graphical user-interfaces. This study extends the concept to the processes of test modeling. The measurement model for usability is based on the following four main attributes suggested by [9]:

1. Learnability: Learnability is the most fundamental usability attribute. [9]Learnability is the measure of ease of learning the technique. The higher

the learnability, the easier it is to learn the technique. Therefore, learnability is defined as:

$$Learnability = \frac{Proficiency\ Level}{Effort}.$$

The *Effort* for learning is measured by recording the time needed to reach the *Proficiency Level*. The *Proficiency Level* of a user is a relative measure dependent on the number of errors committed by him/ her for a given task.

2. Efficiency: Efficiency of the technique is a measure of performance of a user after he/she has achieved a specified level of proficiency[9]. It depicts the productivity of the technique. The efficiency is measured as:

$$Efficiency = \frac{Test\ Suite\ Size}{Effort}.$$

The size of the test suite can be computed by counting the number of test cases it has and the effort can be computed by recording the time to develop the model.

3. Errors: The technique needs to have a low error rate, so that the users commit fewer errors and the criticality of the errors is low. An error is called a *significant error* if it results in generation of a wrong test case or in a missing test for a requirement. Also an error is said to have a *workaround* if there exists methods to eliminate the effect of the error without modifying the model. Criticalities of errors are evaluated based in accordance to the following definitions:

    a. *Level1:* A significant error that does not have a workaround.

    b. *Level2:* A significant error which has a workaround.

c. *Level3:* A non-significant error.

A measure, Error Index, is defined to account for different types of errors. Error index of an application is a weighted sum of the errors in the application. Thus the error indices for the two test generation processes are defined as

$$Error\ Index = 3*(No.of\ \ Level\ 1Errors) + 2*(No.\ \ of\ \ Level\ 2\ Errors) + No.of\ \ Level\ 1\ Errors$$

The levels of errors are assigned weights of 3, 2 and 1 respectively. This scale helps us to derive an aggregate value for the errors for comparison but it might introduce construct related threats. To mitigate the effect, a comparison of level 1, level 2 and level 3 errors is also presented in this chapter along with the aggregate.

4. Satisfaction: This is a subjective measure which describes how pleasant the technique is for use. Satisfaction is measured on a four point *semantic differential scale*: 1- Frustrating, 2-Unpleasant, 3- Likeable and 4- Pleasant.
5. Ease: This is another subjective assessment. The measure shows the difficulty felt by the user in achieving a task. Similar to satisfaction, this measure is also measured on a four point *semantic differential scale*: 4- Very Easy, 3- Easy, 2-Moderately Difficult, 1-Very Difficult.

6.1.3.2 Performance

The performances of the models are measured using the test suites generated from the test models by adopting the same coverage scheme. The performance metrics of the models are thus the same as the performance metrics of the test suites. Two traditional metrics to measure performances of the test suites are effectiveness and

efficiency.[6], [8], [10] The measurement models for these metrics are defined as follows:

1. Effectiveness: The effectiveness of the test model is determined by computing the fraction of the net requirements covered by the test suite generated using the technique. Thus effectiveness for a test suite is defined as:

$$Effectiveness = \frac{Number\ of\ \ Requirements\ Covered}{Total\ Number\ of\ \ Requirements\ in\ the\ Application}.$$

2. Efficiency: The efficiency of the test suite is the measure that determines the effectiveness achieved per unit cost of developing the test suite. Thus efficiency is calculated as

$$Efficiency = \frac{Effectiveness\ of\ \ the\ Test\ Suite}{Cost\ to\ Develop\ \ the\ Test\ Suite}$$

The cost to develop the test suite is a function of the time spent in developing the test model and the time spent in generating the test suites from the test models. The time spent to generate the test suite from the test models is negligible in comparison to the time spent in developing the test models. (See Section 6.6) Thus:

$$Efficiency \cong \frac{Effectiveness\ of\ \ the\ Test\ Suite}{Time\ to\ Develop\ \ the\ Test\ Model}$$

Table 6.2 lists all the measurement models used in the experiment.

Aspect	Contributor	Model
Usability	Learnability	$Learnability = \frac{Proficiency\ Level}{Effort}$
	Efficiency of tool	$Efficiency = \frac{Test\ Suite\ Size}{Effort}$

	Error Index	$Error\ \text{Index} = 3*(\text{No. of Level 1Errors}) + 2*(\text{No. of Level 2 Errors}) + \text{No. of Level 1 Errors}$
Performance	Effectiveness	$Effectiveness = \frac{Number\ of\ Requirements\ Covered}{Total\ Number\ of\ Requirements\ in\ the\ Application}$
	Efficiency	$Efficiency \cong \frac{Effectiveness\ of\ the\ Test\ Suite}{Time\ to\ Develop\ the\ Test\ Model}$

Table 6.2: Measurement models used in the experiment

6.1.4 Hypothesis

The general hypothesis of the experiment is that the test suite generated using HOTTest (***T1)*** is superior in *performance* than the one generated using EFSMs (***T2***). Also, since ***T1*** is based on a formal functional specification of an application, it is hypothesized that the usability of ***T2*** is higher than usability of ***T1***. As discussed before, usability is characterized by five main aspects: Learnability(***Learn***), Efficiency(***Eff***), Error Index (***EI***) ,Satisfaction (***Sat***) and Ease(***Ease***) and the *performance* of the techniques are characterized by two main aspects: Effectiveness (***EffectP***) and efficiency (***EffP***[7]). The hypotheses for the individual variables are defined as follows:

- $H_{0\ Learn}$: There is no difference in learnability of test techniques ***T1*** and ***T2***. Both techniques require equal effort to learn.
- $H_{A\ Learn}$: The difference in learnability of test techniques ***T1*** and ***T2*** is significant.

[7] The efficiency metric of usability, ***Eff*** should not be confused with the efficiency metric for performance, ***EffP***. While, ***Eff*** measures how efficient the tool is in producing a test-model, ***EffP*** measures how efficient the technique is in generating a test-suite of certain effectiveness.

- $H_{0\ Eff}$: There is no difference in efficiency of use between the two testing techniques ***T1*** and ***T2*** .Both techniques require same amount of effort from the users to produce test models of similar size.
- $H_{A\ Eff}$: The difference in efficiency of use for test techniques ***T1*** and ***T2*** is significant.
- $H_{0\ EI}$: There is no difference between error indices of the two testing techniques ***T1*** and ***T2*** .Both techniques are equally error prone.
- $H_{A\ EI}$: The difference in error index of test techniques ***T1*** and ***T2*** is significant.
- $H_{0\ Ease}$: There is no difference in ease of use between the two testing techniques ***T1*** and ***T2*** .Users feel that both techniques are equally easy to work with.
- $H_{A\ Ease}$: The difference between ease of use for test techniques ***T1*** and ***T2*** is significant.
- $H_{0\ Sat}$: There is no difference in level of user satisfaction between the two testing techniques ***T1*** and ***T2***. Users feel that both techniques are equally satisfying.
- $H_{A\ Sat}$: The difference in user satisfaction for test techniques ***T1*** and ***T2*** is significant.
- $H_{0\ EffectP}$: There is no difference between effectiveness of the test suites generated from the test models produced using the two testing techniques ***T1*** and ***T2*** .Test suites generated by the techniques are equally capable of uncovering faults.

- $H_{A\ EffectP}$: The difference in effectiveness of test suites generated through test techniques ***T1*** and ***T2*** is significant.
- H_{0EffP}: There is no difference between efficiencies of the test suites generated from the test models produced using the two testing techniques ***T1*** and ***T2*** .Test suites generated by the techniques are equally costly in achieving similar effectiveness.
- $H_{A\ EffP}$: The difference in efficiencies of the test suites generated through the test techniques ***T1*** and ***T2*** is significant.

6.1.5 Design

An in-vitro experiment was designed to answer the research questions described in section 6.1.1. The experiment was conducted in a class on Software Testing offered at University of Maryland during Fall 2003.The class was divided into two groups and there were two rounds of trainings and observations. The experiment design is shown below in Figure 6.1.

Group 1	***R H A1 A2 T A3 A4 Q M***
Group 2	***R T A1 A2 H A3 A4 Q M***

R – Randomization
H – Training for HaskellDB specification based test modeling
T – Training for extended finite state machine based testing
Ax – Assignment number x {A1 & A3: Smaller Project; A2& A4: Larger Project}
Q – Questionnaire on Satisfaction and Ease
M – Measurement and Analysis.

Figure 6.1: The Experiment Design

The instrumentation of the experiment consists of requirements documents for the applications and log-sheets used for to recording daily progress. Log-sheets were

designed in order to record every event of a student's work session. There were two rounds of assignments. Each round of assignment had a smaller and a larger project. A follow-up questionnaire was designed to assess the subjective attributes of Usability. The measurements on the models and the usability of the techniques were performed later in order to minimize the internal threats.[5]

6.1.6 Threats to Validity

The experiment design minimizes the effects of the threats to internal validity.[5] Biases resulting in differential *selection* of respondents for the comparison groups are eliminated through blocking of the effect of the controlled variable through randomization. The effect of *instrumentation* is minimized because the measurements are performed by a single person at the end of the experiment. The students were monitored through log sheets on their daily performances. This gave us a chance to observe any effects due to *History* and *Maturation*. Post-mortem of the log sheets gave us no indication of such effects. Data was collected through log-sheets and *self reporting* of the data poses another threat to the experiment design. Log-files were automatically generated by the test design tools that reported the time and the activity. Computer generated log-files were used to cross check the log-sheets submitted by the students. Erroneous log sheets were not used for measurements.

Concerning the external validity, the use of students as subjects is a threat. However, the students are senior level undergraduate students of computer engineering, computer sciences and electrical engineering, and many of them had part-time or full time jobs in software companies. (See section 6.2.1) Another threat to the external validity is the requirement specification used in the experiment. The

size of the application is in the smaller range of the real world problems. The scalability and other related issues to HOTTest thus remain unanswered at this point.

The choice of the representative test design technique is an issue for external threats. EFSM based modeling was chosen as the representative after a comprehensive survey, which revealed that the techniques is one of the most frequently used tools. It was found to have the best tool support in all finite-state-machine based test design tools.

6.2 Experiment Preparation

This section describes the preparation needed to conduct the experiment and the subjects acting in the experiment.

6.2.1 Subjects

The subjects for the experiment were students of a senior-level undergraduate course on Software Testing offered at University of Maryland. A total of 28 students were part of the experiment. The undergraduate students were senior year students from Computer Engineering (23 out of 28), Computer Sciences (3 out of 28) and Electrical Engineering (2 out of 28). Table 6.3 provides information on backgrounds of the students. Almost 30% of the students were working in the software industry as part-time employees and another 35% had past experiences in the industry. Some students had significant experiences in software development through their research work. A few of them had full-time job experiences as developers in software companies and a few others as Co-ops/ Interns in the software industry. All of them had extensive programming experiences in the past through various courses and course projects as part of their curriculum. This means that the students were

experienced and, to some extent, comparable to fresh software engineers in the industry.

Profile	Number of Students
Full Time Jobs in Software Companies in the Past	2
Internships in Software Companies in the Past	7
Currently, Part-Time Job in Software Companies	6
Currently, Conducting Research in Industry at Present	2
Currently, Conducting Research in School at Present	5
Currently, Involved in Some Research Projects in School at Present	4
Experience only through Class Projects in the Past	2

Table 6.3: Students' experience profile

The students were not notified about the experiment to ensure that they do not get influenced by the knowledge of the experiment. The experiment was presented as a class project that was mandatory for the course ensuring the necessary motivation. Preventive steps were taken to ensure that the students had no un-wanted communications during the course. The experiment served the educational objective of teaching students a popular finite state machine based test-modeling tool and a formal software specification language, both required by the course curriculum.

6.2.2 Applications

The experiment needed two database applications of different sizes. The applications were designed to generate and execute queries on a static database called PUBS. PUBS is a database created in Microsoft Access with information about authors and their publications. The database has three tables named *authors*, *titles* and *titleauthor*. The applications ask the user for search options and generate SQL queries on the basis of the search options. The smaller application has two possible search options and the bigger has twelve.

Requirement documents were developed for the applications in natural language and the applications were implemented in C++. Requirements documents for the applications were formatted according to the IEEE specification standards. [7] The requirements were analyzed for defects prior to the experiment by two independent inspectors. This was necessary because the requirements were assumed to be correct for the purpose of the experiment. The implementation of the smaller application had 1KSLOC and that of the larger application had 4.2 KSLOC.

6.2.3 Requirement Parsing

In order to measure requirement coverage and effectiveness of the test models, the natural language requirements were thoroughly examined and a list of atomic requirements was produced for the two applications. The list was prepared by an individual who was unaware of the experiment but was very familiar with the applications. This step was taken to avoid any bias arising from personal discretion and the knowledge of the experiment.

The requirements were then classified as general, functional and non-functional requirements. The functional requirements were further classified as domain-specific (DSR) and non-domain-specific requirements. The number of atomic requirements in the two applications is shown in Table 6.4.

		Smaller Application(***A1/A3***)	Larger Application (***A2/A4***)
General Requirements		5	18
Functional Requirements	Domain Specific	27	65
	Generic	26	66
Non-Functional Requirements		2	13
Total		**60**	**162**

Table 6.4: Number of Requirements in the Requirements List

6.3 Experiment

The experiment was run for a span of eleven weeks. Using the controlled variable to get a block design, the students were initially divided into four groups and then randomized into two groups. A questionnaire with eight questions was used to explore the students' experience in functional programming, software testing and database programming. The questionnaire showed that students had four different types of backgrounds. Therefore, it was necessary to divide them into these groups and thereby mitigate the effect of the experience factor from the experiment. After regrouping each group had 12 students. The schedule of the experiment is shown in Table 6.5.

	Group 1	**Group 2**
Day 1 Session 1	Introduction to FSM and Test Modeling	Introduction to Haskell and Functional Programming
Day 1 Session 2	Tool Demonstration	SQL, Relational DB, Haskell DB demonstration
Day 8 Session 3	Tool Tutorial Assignment 1 assigned.	HaskellDB Tutorial Assignment 1 assigned.
Day 8 Session 4	In Class Tutorial	In Class Tutorial
Day 15	Assignment 1 Submitted. Project 1 assigned.	
Day 26	Project 1 submitted.	
Day 27 Session5	Introduction to Haskell and Functional Programming	Introduction to FSM and Test Modeling
Day 28 Session 6	SQL, Relational DB, Haskell DB demonstration	Tool Demonstration
Day 35 Session 7	HaskellDB Tutorial	Tool Tutorial
Day 35 Session 8	In Class Tutorial	In Class Tutorial
Day 42	Assignment 2	
Day 63	Project 2	
Day 77	Questionnaire on Satisfaction and Ease	

Table 6.5: Schedule for the experiment

The students were trained before each round of assignments. There were eight training sessions in all, each of length 1 hr 10 minutes. Apart from the theory presentations, the sessions consisted of in class assignments and practical demonstrations of the techniques and the associated tools. Questions were encouraged during the class but no interactions were allowed among students outside the class. All questions to the instructor, outside the class were through e-mails or through help sessions. Events in lecture and the help sessions were recorded and so were the questions through e-mails. During the 4^{th} and the 8^{th} study sessions, the students were given in-class assignments and they were trained on how to use the log sheets.

The study sessions 4 and 8 followed up with the application of the technique on the smaller project. The project was assigned at the end of the sessions and was due in a week. The students were instructed to work independently and record every event. The experiment details were recorded on log-sheets. The students recorded the time taken, nature and the possible cause of any events in the log-sheets. While designing the log-sheet, it was ensured that it is very easy to fill up and that it is not ambiguous. This ensured that the extra burden on test design because of the log-sheets was minimal. Students were strictly instructed to avoid outside-class communications. There were extra credits for log-sheets which provided them the necessary motivation. The bigger project was assigned a week after the smaller project was submitted and was due after three weeks. After each student turned in the projects and log-sheets, the data was briefly examined for errors and missing information.

A follow-up questionnaire was sent to the students at the end of the experiment to assess the subjective measures of usability viz., satisfaction and ease. Test models were used to generate test cases following the full coverage scheme. Full cover is similar to the all-path coverage schemes used in structural testing. The choice of the coverage scheme was important. Different test coverage schemes have different schemas for test selection. Since full coverage produces an exhaustive set of tests for the system under test, it could be ensured that the performance measurement is not affected by test case selection schemas.

6.4 Measurement and Analysis

The dependent variables were measured and analyzed for inferring on the hypotheses. In the following sections we will address the HOTTest based technique as ***T1*** and the EFSM based modeling technique as ***T2***. The observation sets used for measurement are as follows:

1. ***O1, O3***: The observations conducted on the smaller projects (***A1*** / ***A3***). (See section 6.1.5) The instruments included logsheets, submitted models/specs, log of help sessions and log of e-mail interaction with students.
2. ***O2, O4***: The observations conducted on the larger projects. (***A2***/***A4***). (See section 6.1.5) The instruments included logsheets, submitted models/specs, log of help sessions and log of e-mail interaction with students.
3. ***O5***: The answers to the questionnaire (***Q***) designed to assess satisfaction and ease.
4. $O1_{modified}$, $O3_{modified}$: The observations conducted on smaller assignments after minor corrections and elimination of incorrigible models and specs.

5. ***$O2_{modified}$, $O4_{modified}$***: The observations conducted on larger projects after minor corrections and elimination of incorrigible models and specs.

The statistical significance tests were conducted with an $\alpha = 0.05$. Detailed results from the statistical analysis are presented in the appendix. Box-plots of the data sets were created to identify the outliers and to see the overall trend of the population. For each data-set Kolmogorov Smirnov (K-S) tests are performed to assess the normality of the data. If the data is normal a *dependent t-test* is performed and if the data is not normal, *Wilcoxon Signed Rank* (W-S) tests are performed to infer on the null-hypothesis. An effect size[8] was then calculated for the test statistic using the *Karl-Pearson Correlation Coefficient(r).*

6.4.1 Usability -Learning

The learnability for the techniques was calculated by using ***O1, O3***. The two parameters of interest for learnability calculations are:

1. Time to learn: The time to learn is calculated by adding the time spent in training sessions, time spent during the help sessions and the time spent on study materials. The time spent on study materials was recorded from the log sheets of the students.
2. Proficiency: In order to measure proficiency, the number of errors committed by each student was counted. The proficiency for individual students was then calculated using the following equation:

$$Proficiency\ of\ a\ Student\ X = 1 - \frac{Number\ of\ Errors\ Committed\ by\ Student\ X}{Maximum\ Number\ of\ Errors\ Committed\ by\ any\ Student}$$

[8] Effect size is an objective measure of the importance of the finding; the higher the effect size the higher is the importance of the finding. [28]

The learnability was calculated in accordance to the measurement model specified in section 6.1.3.1 using the two parameters described above. The data collected for ***T1*** and ***T2*** assignments was analyzed for inferring on the null hypothesis $H_{0\ Learn}$. The descriptive statistics for the data set are presented in Table 6.6.

	Proficiency		**Time to Learn(min.)**		**Learnability(*Learn*)**	
	T1	***T2***	***T1***	***T2***	***T1***	***T2***
Number of Data Points	26	28	26	28	26	28
Mean	.8950	.4737	315.5000	307.5000	.0028	.0015
Std. Deviation	.21897	.31155	18.58578	14.50862	.00069	.00100

Table 6.6: A descriptive statistics of learnability data

The number of data points in ***T1*** is 26 as the statistical outliers observed in box-plots (Figure 6.2 (a)) were not considered for analysis. The mean time to learn for ***T1*** was 2.6% higher than that for ***T2***, but the mean level of proficiency for ***T1*** was higher by 88.94% and it over-shadowed the effect of time on learnability calculations. The observed learnability of ***T1*** was 86.67% higher than that of ***T2***. As the data was normal (see Table 15 in Appendix), a *dependent t-test* was performed for comparing the means of the two techniques. The results from the *t- tests* reject the null hypothesis $H_{0\ Learn}$. This implies that the mean values of *learnability* differed significantly.

Further, the effect size of the test statistic using the *correlation coefficient* is 0.61, signifying that the finding is substantive. Figure 6.2 (a) shows a box-plot of learnability values for the two test techniques, which shows a marked difference between learnability of the test techniques. The median values denoted by the horizontal line in the center of the boxes are significantly different.

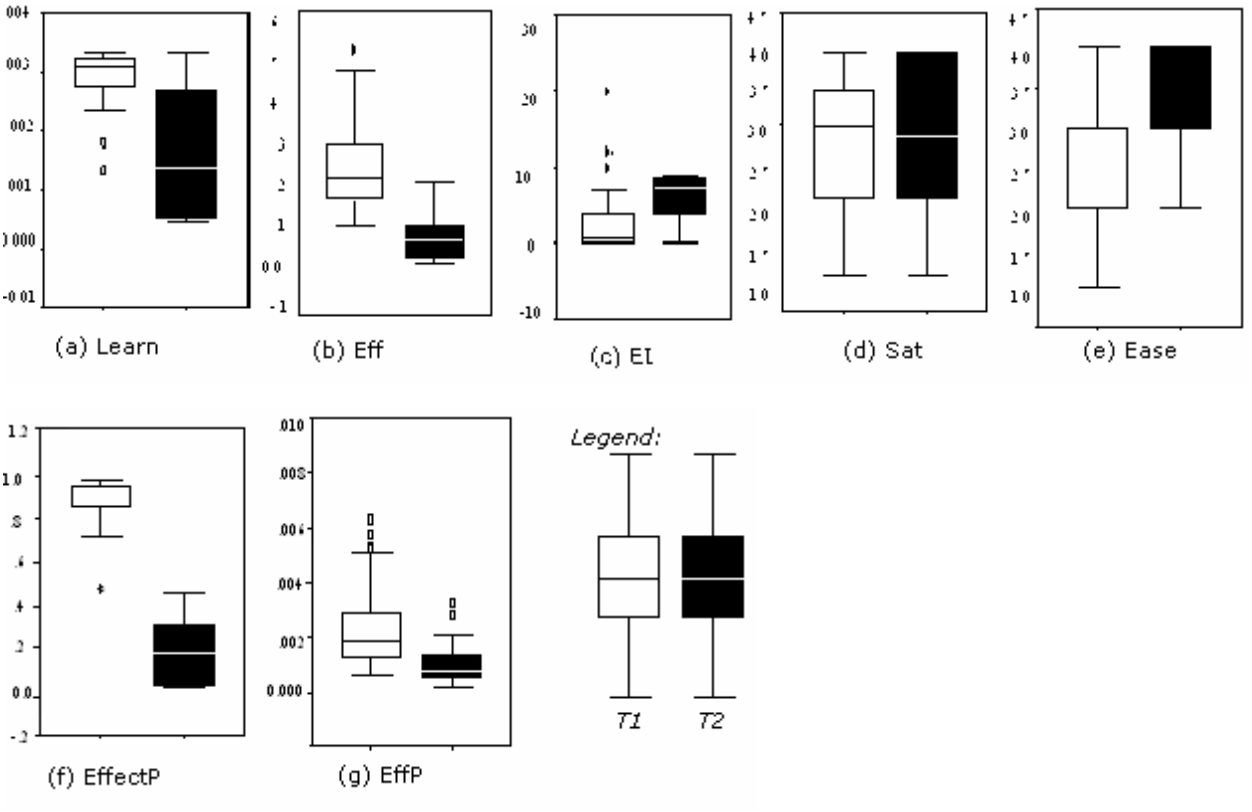

Figure 6.2: Box plots of the data for the two test techniques.

6.4.2 Usability-Efficiency

The efficiency aspect of the usability is calculated using ***O2*** and ***O4.*** All students who did not achieve enough proficiency in ***O1*** and ***O3*** were filtered out while considering the data in ***O2*** and ***O4*** respectively. This was necessary to filter out the effect of proficiency on the results of efficiency. The two parameters of interest for efficiency calculations are:

1. Effort: The time to develop the test model was considered as the measure of effort. This value was directly available from the log-sheets.
2. Size of the Test Suite: The test suites produced from the test models were used for measuring the size of the test suites. The number of test cases in a test suite is not a very good indicator of its size because the sizes of different test

cases themselves vary depending on the number of requirements they test for. Thus a good indicator of the test suite size is the number of atomic requirements covered by the test models. The parsed requirements-list was used as a basis for measuring the number of atomic requirements covered by the test models. The number of test cases in a test suite is a function of the number of the requirements covered and is directly proportional to it.

The efficiency values for the test generation techniques are calculated using the above parameters. The descriptive statistics are presented in Table 6.7.

	No. of Requirements Covered		**Time to Develop(min.)**		**Efficiency(*Eff*)**	
	T1	***T2***	***T1***	***T2***	***T1***	***T2***
Number of Data Points	23	24	22	24	22	24
Mean	117.5217	30.38	568.046	497.46	0.25	0.084
Std. Deviation	3.47549	10.172	252.689	310.962	0.115	0.053

Table 6.7: A descriptive statistics of efficiency data

The number of requirements covered by ***T1*** is 286% higher than the number of requirements covered by ***T2*** and as the time to develop the model is only 14% higher, the mean efficiency for ***T1*** is considerably higher(197.62%) than that of ***T2.*** Also, the t-test rejects the null hypothesis $H_{0\ Eff}$, which implies that the means differ significantly. (See Table 15 in Appendix) The effect size of the test statistic is calculated to be 0.75 that signifies that the effect of the test is large. [1]Figure 6.2 (b) shows the box plots for the efficiency of production using the two test design techniques. ***T1*** is superior to ***T2*** for almost all data points. The median value for ***T1*** is three times that of ***T2***.

6.4.3 Usability- Error

The direct measure error is counted from ***O1*** and ***O3***. The errors are assigned criticality on the basis of the rules specified in section 6.1.3.1 and corresponding *error index* was calculated. Descriptive statistics for the errors are mentioned in Table 6.8.

	T1				***T2***			
Level	1	2	3	**EI**	1	2	3	**EI**
Number of Data Points	22	22	22	**22**	20	20	20	**20**
Mean	.64	.45	.00	**2.82**	1.25	.95	.45	**6.10**
Std. Deviation	1.529	.739	.000	**5.197**	.786	.510	.510	**2.882**

Table 6.8: A descriptive statistics of data on *error index*

The mean error index for ***T1*** is 116.3% lower than that of ***T2***. The frequencies of level 1, level 2 and level 3 errors for ***T1*** is uniformly lower than those for ***T2.*** The data is not a part of normal distribution. (See K-S tests in Table 13 in Appendix) Hence, to test for sufficiency W-S tests [2] are conducted instead of the usual *t test*s. The W-S tests' significance values were 0.004 ($p < 0.05$), implying that the null hypothesis $H_{0\ Ei}$ is rejected. Further, the effect size is calculated to be (- 0.39) which implies a medium sized effect for the findings. Figure 6.2 (c) shows the box plot of the error index of the two testing techniques. The median and the upper limit for the error index of ***T1*** is less than that of ***T2*** and the error indices for ***T2*** are uniformly higher than that for ***T1.***

6.4.4 Usability- Satisfaction and Ease

The satisfaction and ease aspects of the usability were measured using ***O5***. Table 6.9 presents a descriptive analysis of the results. The number of data points is

less than 28 because of less number of respondents to the questionnaire. All data sets pass the normality tests. (K-S significance value >0.05) The test of hypothesis is conducted using dependent t-tests. The mean satisfaction with ***T2*** is marginally higher (3.92%) than that with ***T1***. However, the t-test significance levels >0.05 means that we accept hypotheses $H_{0\ Sat}$. This signifies that the population means do not differ sufficiently. Further the effect size for the test findings on Satisfaction is 0.06, signifying a small effect. 0.

	Satisfaction (*Sat*)		**Ease(*Ease*)**	
	T1	***T2***	***T1***	***T2***
Number of data points	21	18	21	18
Mean	2.8333	2.9444	2.7381	3.3611
Std. Deviation	.99163	.87260	.86051	.58926

Table 6.9: A descriptive statistics of subjective attributes of usability data

The mean value assessment for ease of ***T2*** is 23% higher than ***T1*** and the test on hypothesis $H_{0\ Ease}$ fails, ($p < 0.05$). This implies that the difference in subjective assessment of ease is significant. The effect size for the t-statistic on Ease is 0.39 signifying a medium sized effect. Figure 6.2 (d) and Figure 6.2 (e) show the box plots of ease for ***T1*** and ***T2***. There is a significant difference in subjective assessment of ease for the two techniques, and there is little or no difference in satisfaction assessment for the two techniques.

6.4.5 Performance-Effectiveness

The performances of the models were evaluated from $O1_{modified}$, $O2_{modified}$, $O3_{modified}$ and $O4_{modified}$. The test models and specification were corrected for minor mistakes if there were any. The modifications were mainly to ensure that each of the models could be used for generating test cases following the full-cover scheme.

Since, effectiveness is independent of the proficiency level of the students, ***O1*** and ***O2*** could be used for measurement after minor corrections. The average number of corrections per model is 0.12. The modifications do not affect the measurements as they did not alter the test suites. The two parameters of interest for effectiveness calculations are:

1. Total Number of Requirements: The total number of requirements was computed from the parsed requirements.
2. Number of Requirements Covered: All functional requirements affect the output of the application. A requirement is said to be verified if at least one test case checks for the variation in the output from the application due to a missing requirement or due to a faulty implementation of the requirement. The total number of requirements covered is computed after examining the test suite generated from various test models following the full-cover scheme.

Effectiveness of a test model is measured by calculating the fraction of the requirements covered through the generated test suites. Table 6.10 presents the descriptive statistics for effectiveness calculations.

	Effectiveness(*EffectP*)	
	T1	***T2***
Number of Data Points	45	43
Mean	**.8668**	**.1197**
Std. Deviation	.08440	.1201

Table 6.10: A descriptive statistics of effectiveness data

The mean effectiveness of the test models derived from ***T1*** are 624.14% superior to those derived from ***T2***. Again, the K-S tests for normality are negative for the ***T2*** data. So for checking the null hypothesis, $H_{0\ EffectP}$, the Wilcoxon Signed Rank

Test was used. The sig value for the W-S tests <0.05 resulting in rejection of the null hypothesis $H_{0\ EffectP}$. This signifies that the means of the samples differ significantly. Also, the effect size for the test statistic was found as -0.87. This implies a very high effect size and a substantive finding. The box plots of the data show that the effectiveness values of the two test suites have significant differences. (Figure 6.2 (f)) The lowest value of effectiveness for ***T1*** is greater than the highest value of effectiveness for ***T2***.

	Effectiveness (*EffectP*$_{generic}$)	
	T1	***T2***
Number of data points	45	45
Mean	.884	.491
Std. Deviation	.0683	.1882

Table 6.11: A descriptive statistics of effectiveness of test suites for generic requirements

A functional requirement can be further classified as domain specific requirement or as a generic requirement. A second round of measurement was conducted considering just the generic requirements. ***T1*** again faired over ***T2*** by 80%(See Table 6.11). The Wilcoxon Signed Rank test rejected the null hypothesis again, and therefore the means differed significantly. Also, the effect size of the test is -0.86 signifying a large effect.

6.4.6 Performance- Efficiency

In order to measure the efficiency of the models observations ***O2***$_{modified}$ and ***O4***$_{modified}$ were used. Unlike in effectiveness measurements, ***O1*** and ***O3*** could not be used in efficiency measurements because the measure is dependent on time to develop the test model. Time to develop is a true indicator of effort only when the

user has achieved a specified level of proficiency. The efficiencies were calculated by measuring the following parameters:

- Effectiveness: Effectiveness was calculated as before. (See section 6.5)
- Time to develop the Test Model: The time to develop the test model or the time to write the equivalent specification was considered as the measure of effort. This value was directly available from the log-sheets.

The efficiency for the test model is calculated in accordance to the equation specified in section 6.1.3.2. using the above parameters. The descriptive statistics are presented in Table 6.12

	Efficiency (*EffP*)	
	T1	***T2***
Number of Data Points	22	24
Mean	**.0018**	**.0007**
Std. Deviation	.00085	.00043

Table 6.12: A descriptive statistics of efficiency data

The mean efficiency for ***T1*** is 157% higher than that of ***T2***. Since the data sets are parts of a normal distribution, to test for sufficiency t-tests are conducted as before. The t-test rejects the null hypothesis $H_{0\ Eff}$, which implies that the difference in means is statistically significant. So it can be said that the *efficiency* for ***T1*** is significantly higher than that of ***T2***. The effect size of the test statistic is calculated to be 0.75, which signifies that the effect of the test is large.[1] Figure 6.2 (g) shows the box plots for the efficiency in learning for the two test-design techniques. ***T1*** is superior than ***T2*** for almost all data points. This is true also for the outliers.

6.5 Results and Discussion

A summary of the analysis of the experimental data is listed in Table 6.13. The analysis of the experimental data as presented in the previous section rejected six out of seven hypotheses. This implies that differences in traits for the two test techniques ***T1*** and ***T2*** are significant. Further, the effect sizes for the test statistic are mostly large, implying that the findings are substantive.

Variable	Trait	Superior Technique	Hypothesis accepted?	Effect Size
Usability	Learnability	***T1***	No	Large
	Efficiency	***T1***	No	Large
	Error Index	***T1***	No	Medium
	Satisfaction	***T2***	Yes	Small
	Ease	***T2***	No	Medium
Performance	Effectiveness	***T1***	No	Large
	Efficiency	***T1***	No	Large

Table 6.13: An Overview of the Analysis

It can be inferred from the results that the technique ***T1*** based on Higher-Ordered-Typed functional specification of the application is superior in effectiveness to ***T2*** which is based on EFSM based system modeling. There is a gain in effectiveness by a factor of 7.3, while using ***T1***. Such behavior can be ascribed mainly to the following reasons:

1. ***T1*** captures a greater number of requirements by including those which are domain specific.
2. ***T1*** users translate the specification in natural language to another textual representation. This aids in achieving a systematic abstraction process. The higher effectiveness observed in case of the generic requirements is an indicator of the fact that the abstraction process used by the users in HOTTest is better than that of EFSM based modeling.

3. ***T1*** tool is supported by static type checker of the Haskell type system. This eliminates many human errors and ensures that the model is a closer representation of the system.

A similar verdict can be reached, while considering efficiencies of the test suites. The efficiency of the test suites produced using ***T1*** is 2.6 times higher than that of the test suites produced using ***T2***. The average time needed to develop HaskellDB model is higher than the time needed to create test models for EFSM based technique, but the gain in effectiveness offsets the loss in time.

The subjective assessment of *ease* was in favor of ***T2*** showing that the subjects of the experiment like any other programmer of imperative languages had difficulty in changing the perspective. Only two of the subjects said that ***T1*** is extremely easy to use. Both subjects had prior experience in coding applications using ML. ML is a functional language like Haskell and therefore, it was easier for them to translate the concepts.

Further, the satisfaction measure for the students was in favor of ***T2*** but the null-hypothesis was accepted. This implied that the differences in the mean were not significant enough. Even the box plots showed a similar data for both test design techniques.

The surprising discovery was the fact that for all the objective measures of usability, ***T1*** performed better than ***T2***. For all such measures the null hypotheses were rejected showing a significant difference in means. Also, for all such measures the effect sizes of the test statistics are large showing that the findings are substantial. This indicates that although the users don't feel good about using formal functional

specification based modeling techniques, they actually perform better with them. They learn it better, and they commit errors less frequently. The two possible reasons for such behavior are:

1. The functional specification method's strong typed-ness inadvertently forces the users to concentrate more on the task.
2. The functional specification method has a better error feedback mechanism (in terms of static type checking) to prevent users from committing recurrent errors. The strong typed specification language made error tracking easier.

A comparison between first round of assignments (observation sets ***O1*** and ***O2***) with second round of assignments (observation sets ***O3*** and ***O4*)** is presented in Table 6.14. It is observed that there is a slight increase in the mean values for most variables but the difference is not statistically significant. This implies that the students' knowledge of the system has a minimal effect on the variables of the experiment.

Test Technique	Variable	Descriptive Statistics			Comparison of Means(t-test)			
		Round	Mean	Std. Deviation	t	Df	significance	Mean difference
T1	EffectP	1	0.8517	.04217	2.530	21	0.079	0.0360
		2	0.8157	.02455				
	EffP	1	0.0015	.00087	-1.450	20	0.162	-0.0005
		2	0.0020	.00078				
	Eff	1	0.1546	.08124	-1.649	20	0.115	-0.0568
		2	0.2114	.08042				
	Learn	1	0.0025	.00104	-1.339	20	0.195	-0.0005
		2	0.0030	.00050				
	EI	1	3.8000	6.82805	0.802	20	0.432	1.8000
		2	2.0000	3.43776				
T2	EffectP	1	0.2198	.0742	-1.550	22	0.135	-0.0504
		2	0.2701	.0847				
	EffP	1	0.0006	.0004	-0.717	22	0.481	-0.0001
		2	0.0007	.0004				
	Eff	1	0.0214	.0171	-0.634	22	0.533	-0.0047
		2	0.0262	.0199				

	Learn	1	0.0012	.0009	-0.729	18	0.476	-0.0003
		2	0.0015	.0009				
	EI	1	6.60	2.797	0.767	18	0.453	1.00
		2	5.60	3.026				

Table 6.14: Comparison of Performance between first and second rounds of assignment

6.6 Summary

The experiment presented in this chapter compares HOTTest, a Higher Ordered Typed Domain Specific Language based test design technique with a test design technique based on creation of EFSM models for the software. The performance aspects of the test suites generated using the two techniques were compared along with the usability aspect of the respective techniques.

The main result from the analysis is that HOTTest provides enhanced performance of the test suites without any compromise on usability. The important results from the experiment are:

- Performance of Test Suites: HOTTest can generate test suites that are more effective and efficient than the test suites generated from the EFSM based test models. The gain in effectiveness is by a factor of 7.3 and in efficiency by a factor of 2.6.
- Usability of the Technique: The users feel HOTTest is more difficult to use than the EFSM based test design tool used in the industry but they learn it faster and produce more effective models in less time. Further, while using HOTTest the users commit fewer errors and the errors have a lower criticality.

6.7 References

[1] A. Field and G. Hole, *How to Design and Report Experiments* .London, UK: SAGE Publications, 2003

[2] A. Hughes and D. Grawoig, *Statistics: A Foundation for Analysis* .Reading, MA: Addition Wesley Publishing Company, 1971.

[3] B. E. John, "Evaluating Usability Evaluation Techniques," in *ACM Computing Surveys*, [Online] 28 (4es).Available: http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a139-john/

[4] C. J. Wang and M. T. Liu, "Generating Test Cases for EFSM with Given Fault Models," in *Proc. IEEE Infocom*, 2, pp. 774-781, 1993.

[5] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Designs for Research.* Chicago: Rand McNally and Company, 1963.

[6] G. E. Stark, R. C. Durst and T. M. Pelnik, "An Evaluation of Software Testing Metrics for NASA's Mission Control Center" [Online] MITRE, Available: http://hometown.aol.com/geshome/ibelieve/sqjsubm2.pdf.

[7] *IEEE Guide to Software Requirements Specification,* IEEE Standard 830, 1984.

[8] J. K. Chaar, M. J. Halliday, I. S. Bhandari and R. Chillarge, "In-Process Evaluation of Software Inspection and Test," in *IEEE Transaction on Software Engineering*, Vol 19, No. 11, November, 1993, pp. 1055-1070

[9] J. Nielsen, *Usability Engineering* .San Diego, CA: Academic Press Inc.,1993.

[10] J. T. Huber, "Efficiency and Effectiveness Measures to Help Guide the Business of Software Testing", in *Applications of Software Measurement*, HP

Labs Research Report, 1999, [Online]. Available: http://www.benchmarkqa.com/PDFs/efficiency_measures.pdf.

[11] P. Savage, S. Walters and M. Stephenson, " Automated Test Methodology for Operational Flight Programs," in *Proc. IEEE Aerospace Conference,* vol.4, pp. 293-305, 1997.

[12] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir, "Test Development For Communication Protocols: Towards Automation," in *Computer Networks*, 31, 1999, pp. 1835-1872.

[13] *Test Master User's Guide*, Release 1.9.5, Empirix Inc., New Hampshire, 1999.

Chapter 7 Industrial Applicability of HOTTest and Other Test Generation Tools

Another problem common model based test design techniques face is their limited ability to scale–up to industry scale problems. A case study was designed in order to assess the ability of HOTTest to scale up to industrially large and complex problems. This chapter reports the case study: its design and results.

HOTTest and three other model based test design techniques were used to generate test cases for an industrial application. The three other techniques that were selected for comparison with HOTTest(called HOTTest in this study) are as follows: an Abstract State Machine [6] based test generation technique (called ASML in this study), an UML based test generation technique [2] (called Archetest in this study) and an Extended Finite State Machine [8] based test generation technique(called EFSM in this study). The choice of ASML, Archetest and EFSM was guided by the fact that these are the most commonly used modeling techniques in the industry.

7.1 Description of the Test Design Tools

7.1.1 Archetest

Archetest [2] is a test generation technique that supports test case generation from high level use case and domain models captured using UML [5]. The tool for Archetest is provided as a plug-in to the Rational Rose UML modeling tool [11], enabling the user to specify use cases precisely. The domain model is a class diagram, and serves to indicate the domain classes that can be instantiated by the system. The use cases are formalized to produce a use case specification by adding five key concepts to standard use cases.

1. Preconditions - these state what must be true in the system for the use case to be eligible for execution. Preconditions are always written based on domain model instances.
2. Parameters and partitions - the tool allows the modeler to declare typed parameters that represent input to the use case from an actor. Parameters may have partitions associated with them. Partitions are logical values that testers typically use to think about test data. For example, a password may be valid or invalid, so those values could be used as partitions for a password parameter.
3. Test Data - the partitions are associated with physical values that can be used in actual testing of the system.
4. Results - results are named outcomes of executing the use case. They are guarded by constraints that indicate under what conditions they occur, and also have associated update statements that change the state of the system.
5. Execution template mapping - this defines how the use case is realized in terms of the APIs[9] of the implemented system. This allows executable test scripts to be generated.

[9] Application Programming Interface, it is a set of definitions of the ways in which one piece of computer software communicates with another.

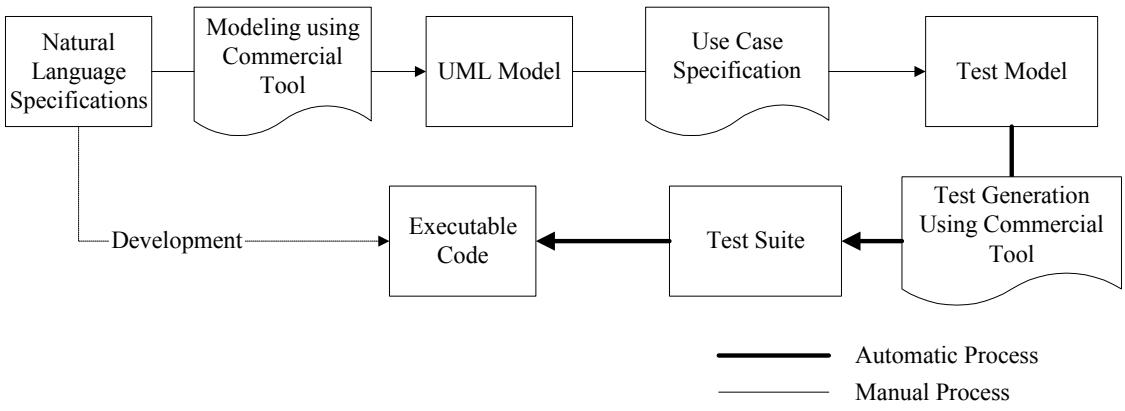

Figure 7.1: The Test Framework using Archetest

Figure 7.1 represents the test generation framework used in Archetest.

Given an Archetest model, several transformations occur leading to the generation of test cases. First, fault modeling techniques are applied to determine interesting test variations to try on a use case by use case basis. Second, integer programming techniques are used to determine efficient ways to flows through the system that consist of multiple use cases. Finally, the test cases are generated using a series of graph traversal techniques. Preliminary studies of the Archetest tool indicate that it can improve both testing quality and the cost associated with testing [2].

7.1.2 ASMLT

The AsmL Test Generator tool, or short *asmlt*, is an integrated test generation environment. It can be used to automatically generate test cases from an AsmL model using various algorithms, and to use such test cases to perform a conformance test against an actual implementation. The test generation process in ASMLT can be divided into following steps:

1) Finding interesting sequences of method calls
2) Finding interesting parameters for each method call
3) Performing a conformance test against an implementation.
4) Sequence Selection

When testing an application, each test sequence consists of a sequence of method calls. The ASMLT's approach towards test-case generation is as follows. First, it generates a finite state machine (FSM) from an AsmL model. The process of generating FSM from ASM is close to that of typical model checking methods. The tool fires all possible transitions and based on an abstraction property, abstracts them into hyper states. The hyperstates are finite in number and thus can form the states of the FSM. Then ASMLT generate test cases using a Chinese postman tour [7] to traverse the states of the FSM.

After the method sequences are generated the next step is to generate test data using the parameter generation feature in ASMLT. There are two ways the parameter generation feature can be used:

1) By itself. In this case the searched-for "parameters" really are values which satisfy certain conditions. Such values can be seen as test cases by themselves.
2) Or, as part of the sequence selection to find parameters for each method call of the test sequence.

Once the test suite is generated and an implementation of the application is available, one can perform a conformance test using ASMLT.

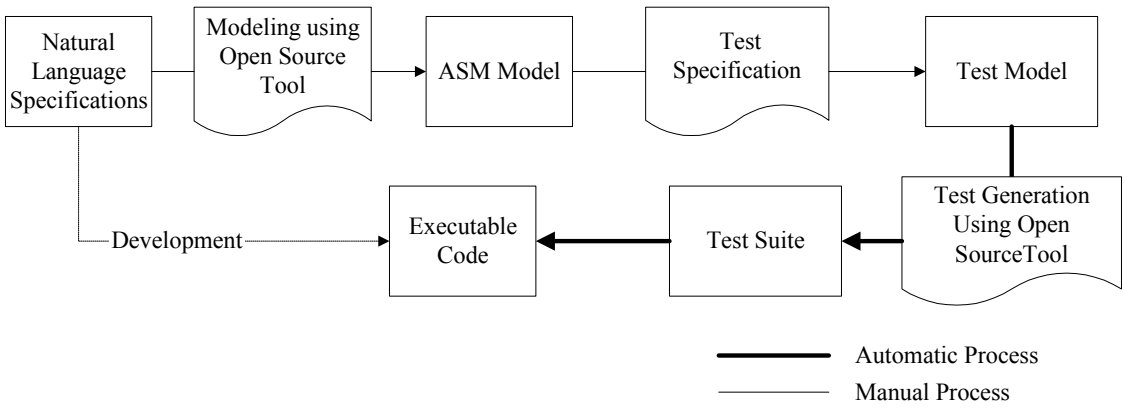

Figure 7.2: The Test Framework using ASMLT

7.1.3 EFSM Based Test Generation

An EFSM consists of hierarchically arranged states and transitions between states. A transition is triggered by an event provided that the enabling condition is satisfied. Tools based on EFSM test generation provide graphical user interfaces for creating EFSM models of the system. The user/tester creates a behavioral model of the system by identifying the states and transitions of the system under test. Each transition has field like actions, predicates, likelihood etc. that help in establishing the context of a test scenario. The states are arranged hierarchically allowing the user/tester to choose an appropriate level of abstraction. Each path (a set of possible transitions from entry to exit states) in the model such constructed signifies a test case for the system under test. These tools can generate test cases following user specified path coverage schemes such as Full Cover, Transition Cover, and Profile Cover etc. Figure 7.3 depicts the test design framework using EFSM based models.

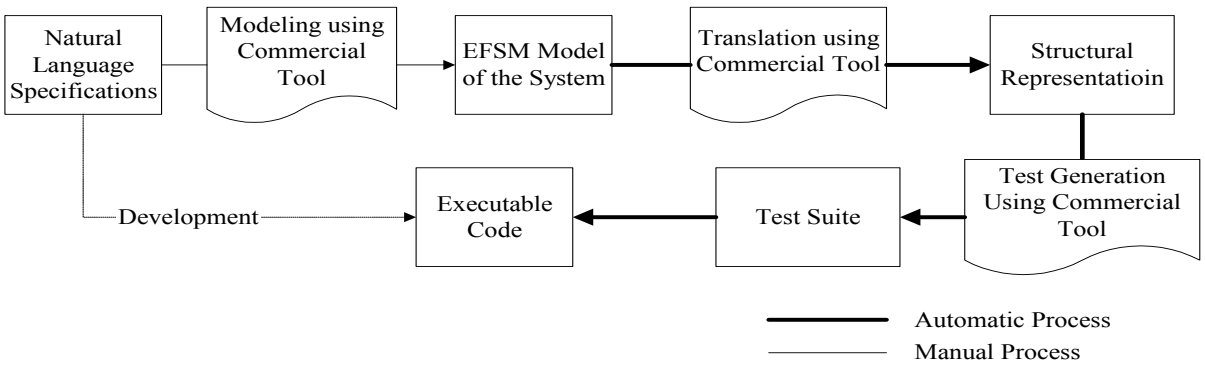

Figure 7.3: The Test Framework using EFSM based modeling

7.2 Design of Case Study

7.2.1 Design of the Measurement Framework

The measurement framework for the case study was designed following the GQM methodology. Chapter 7 GQM defines a measurement model on three levels:

1. Conceptual level (goal): A goal is defined for an object, for a variety of reasons, with respect to various models of quality, from various points of view, and relative to a particular environment.
2. Operational level (question): A set of questions is used to define models of the object of study and then focuses on that object to characterize the assessment or achievement of a specific goal.
3. Quantitative level (metric): A set of metrics, based on the models, is associated with every question in order to answer it in a measurable way.

The purpose of the case study is to characterize HOTTest's test generation abilities against similar existing test generation techniques. Therefore at the conceptual level, the goal is defined as:

Goal: "*Analyze* ***HOTTest*** *to* ***characterize*** *it with respect to its* ***test generation ability*** *from the point of view of the* ***testers (in Industry)****."*

The purpose of the study is carefully chosen as to *characterizing* which implies that the case study will serve to provide the first insight into certain aspects pertinent to test generation ability of HOTTest. The scope of this study limits us to generalize the comparison to all possible test generation tools. In absence of the established benchmarks for test generation, this is the only way to obtain an initial estimate of the necessary parameters.

7.2.1.1 Questions and Metrics

At the operational level we ask the following questions .

1. How **complex** is the tool to use?
2. How easy is it to **learn** the tool?
3. What is the **effectiveness** of the tool?
4. What is the **effort needed for testing** of applications?
5. How does the test generation effort **scale** up with application size?

It is understood that the higher the *complexity* the lower is the usability of the tool. A higher complexity will also mean a higher investment on human resources manifested through higher hiring and training costs. *Ease of learning* is another vital issue that governs applicability. Lower *Ease of learning* implies more difficulty in educating the testers on the test generation technique. *Complexity* and *Ease of Learning* determines directly the productivity in terms of volume and quality of the test cases generated. The next two questions help determine the return-on-investment on installing and using the test generation technique for various application projects.

The *effort needed for testing* and the *effectiveness* of the test technique jointly determine the appropriateness of the technique. The gain in effectiveness should not be at the cost of heavy increase in effort. Also, the testers in the industry need to ascertain that the technique *scales-up* to large applications.

7.2.1.2 The Metrics and the Measurement Models:

At a quantitative level GQM defines metrics. These metrics help answer the questions asked at the operational level. They also provide the necessary comparative basis for the techniques under scrutiny. Following are the metrics that are part of this study:

1. *Complexity*: Complexity of the test generation process is defined as the difficulty in using the tool owing to the number of concepts that one needs to learn in order to produce a correct test model. The concepts include the basic set of operators, functions, and modeling elements that a user needs to learn in order to use the tool. The complexity ($CPLX_{Form}$) is measured using a graph depicting dependencies between semantic concepts. We call such graphs as semantic dependency graphs. A semantic dependency graph relates the various concepts expressing their hierarchical dependencies. The nodes of a semantic graph can consist of any of the following concepts:
    a) Modeling Concepts (n_{mod}): The modeling concepts are the concepts related to test models. For example for a UML based model these are the concepts like Use Case Diagrams, Class Diagrams, and Activity Diagrams etc.
    b) Linguistic Concepts (n_{ling}) : The concepts related to the language of specification constitute the linguistic concepts. These include the

language related constructs, special functions and operators any user needs to learn to be able to use the tool. For example in ASML these are the language constructs supporting abstraction mechanisms.

c) Technique Concepts (n_{tech}) : These are the concepts that relate to the test generation technique. For example in EFSM based test design techniques these include constraint related concepts that define the context of any state.

d) Tool Specific Concepts (n_{tool}) : These are the concepts that should be learnt in order to use the tool supporting the technique. For example for Archetest the concepts related to Tofu combinations are tool specific concepts.

Complexity of any concept is measured on a scale of three (1- Easy, 2- Moderate and 3- Difficult). The difficulty is decided from the perspective of the user of the tool. Complexity of any node in a semantic dependency graph is calculated as follows:

$$CPLX_{node} = \sum_{all\ child\ nodes} CPLX_i + CPLX_{concept} \qquad [7.1]$$

This metric indicates the conceptual complexity of the formalism involved for each of the tools and does not necessarily signify the ease of learning. The ease of learning is dependent on the conceptual complexity and also on the quality and amount of support available for the tool for the purpose of learning. It is possible for tools having similar complexities to differ in their ease of learning.

2. *Ease of learning*: Ease of learning indicates the time a user needs to achieve a specified proficiency-level with the tool. Proficiency is measured by creating a list of concepts and then by measuring the proficiency that the user achieves for each such concept. The proficiency is measured on the basis of users' performance in a pilot project. Proficiency for any individual concept i is given as

$$prof_i = \frac{n_i}{N_i} \ , \qquad [7.2]$$

where n_i is the total number of correct uses of the concept i by the user in the model for the pilot project and N_i is the total number of usage instances of the concept in the model. The proficiency of any user is calculated as

$$PROF = \frac{\sum_{i=1}^{m} prof_i}{m} \ , \qquad [7.3]$$

where m is the total number of concepts used in the model by the user. Ease of learning is measured in accordance to the following equation:

$$EASE = \frac{PROF}{t_{Learn}} \qquad [7.4]$$

The time to learn (t_{Learn}) is the time needed for the user to achieve the recorded proficiency. This includes the time to model and debug the pilot project along with the time spent during the training. The time can be recorded in minutes. Ease of learning contributes positively towards usability. A high ease of learning indicates high usability.

3. *Effectiveness*: The effectiveness of the test technique is defined as the techniques' ability to determine faults. A fault is defined as the inability of an

application to satisfy a requirement. Thus number of faults is equal to the number of requirements that the application fails to satisfy. Thus effectiveness in finding faults can be determined by computing the fraction of the net requirements covered by the test suite generated using the technique. Thus effectiveness for a test suite is defined as:

$$Effectiveness = \frac{Number \text{ of Requirements Covered}}{\text{Total Number of Requirements in the Application}} = \frac{r}{R} \quad [7.5]$$

where r = number of requirements tested by the application and R = net number of requirements to be tested.

4. *Efficiency*: Efficiency is a measure of ease of testing a system after a user's learning is complete. Efficiency is calculated as

$$EFF = \frac{\text{Size of the Testing assignment}(Z)}{\text{Effort in Testing } (T_{MM})} \quad [7.6]$$

The size of the testing assignment can be measured as follows:

$$Size\ of\ the\ Testing\ assignment(Z) = Application\ Size * C_{Coverage} * C_{Complexity} \quad [7.7]$$

where $Application\ Size =$ Size of the system under test in LOC/ FP, $C_{Coverage} =$ Coverage coefficient of the test model, $C_{Complexity} =$ Complexity of the application. The coverage coefficient is measured as in equation 7.5.

Effort in testing can be measured by summing the time for test modeling, time for test generation and the time for test execution. Time for test modeling and time for test generation is a direct measure. Execution time can be computed by summing the time to setup the execution environment,

the time to run the test cases and the time to verify the results. Execution time is an indicator for the effort to execute.

Effort for testing (T_{MM})
= Time to develop the test model(T_M)
+ Time to generate test cases (T_G) [7.8]
+ Time to set up the execution environment(T_S)
+ Time to execute the test cases(T_X)

Efficiency contributes positively towards the usability of the tool. A high efficiency enhances the usability of the test tool.

5. Scalability: Scalability is defined as the tool's ability to scale up to large application. It can be measured as the increase in application size per unit increase in effort. Therefore, we define scalability as :

$$EffortScalability(S_i) = \frac{IncreaseInApplicationSize(\Delta Z)}{IncreaseInEffort(\Delta T_i)} \quad [7.9]$$

ΔT_i is a component of net effort in testing as is defined in equation 7.8. Thus scalability of the tool with regards to effort in modeling is defined as:

$$EffortScalability(S_M) = \frac{IncreaseInApplicationSize(\Delta Z)}{IncreaseInEffort(\Delta T_M)} \quad [7.10]$$

For this case study, ΔZ is the same for all the test generation tools. Therefore, for comparison purposes,

$$EffortScalability(S_i) \sim \frac{1}{IncreaseInEffort(\Delta T_i)} \quad [7.11]$$

7.2.2 Case Study Instruments

The subject of the case study was a summer intern working in Software Testing at the IBM TJ Watson Research Center. The intern had previous formal training in software testing and he compared in experience to a fresh hire of a software firm.

The assignment was test generation for a tool called *JMYSTIQ* (*Java - Managing Your Software To Improve Quality*). JMYSTIQ is a tool for the analysis of defect data from software development and service. It is an essential tool for organizations using the ODC (Orthogonal Defect Classification)[9] methodology for capturing defect information.

JMYSTIQ is implemented in java and is a GUI driven tool that generates and executes SQL queries for a database of defect reports collected during various phases of software development lifecycle. The dataset that results from the queries is analyzed to provide valuable insights on most issues facing a development organization (e.g. product stability, test effectiveness, customer usage, etc.) to drive actions that would take time-consuming specialist task forces to identify.

In addition to JMYSTIQ, two other applications were modeled using the techniques by the subject. One is called SSP and the other is called SearchPUBS.

SSP is a small application for generating queries for the PUBS database. PUBS is a database created in MS Access with information about authors and their publications. The database has three tables named *authors*, *titles* and *titleauthor*. The application generates and executes queries on PUBS. The application asks the user for search options. The user may opt to search by first or last name of the author, by his city, or

by the title of the book written by the author. If successful, the application returns the Name of the author(s) and the corresponding publication(s) or else the application returns a message saying "No such entry". SSP was modeled by the subject while learning the tool. SSP was chosen because it is a relational DB based application, similar to JMYSTIQ.

Search PUBS is also an application based on the database PUBS. It is a VC++ application of size 4.2KSLOC that queries a static relational database called PUBS for author information. The user can form queries by using 12 search criteria provided in a dialog based system.

Following are some artifacts that were part of the case study:

1. **JMYSTIQ Product Description (A1)**: The artifact number **A1** was a set of documents that described the product characteristics of JMYSTIQ. It listed the functionalities that JMYSTIQ needed to satisfy and also presented screenshots and essential directives for the users.
2. **JMYSTIQ Requirements Documents (A2)** : A natural language specification is a description of JMYSTIQ's functional and non functional requirements specified in accordance to the IEEE format [4]. This is called **A2** in this case study and was the sole basis for creating the test models using the tools. **A2** was prepared by the subject of the case study from **A1** under the supervision of an ODC expert.
3. **Parsed Requirement List (A3)**: A parsed requirement list is a list that contains requirements decomposed to their lowest level. These are the requirements that do not subsume or are composed of other requirements. The

parsed requirement list is prepared by an ODC expert on the basis of **A2**. The requirements were further classified as domain specific and generic requirements to help classify the results.

7.2.3 Case Study: Process

The subject of the case study learnt the tools and generated tests for JMYSTIQ. The first step in the case study was study of JMYSTIQ. The subject was trained on ODC and was made to develop the natural language specification (artifact **A2**) for JMYSTIQ from artifact **A1**.

The next step was learning of the test technique ***T0*** by the subject. Learning of the tools was accomplished using SSP. Learning of the tools was accomplished through the technical manuals and the user support documents available for the tool. A log was maintained in order to record the learning time and also to document any special observations during the learning phase. After perusal of the documents, the respective test generation tools were applied on SSP. The results of test generation on the small tool were used to evaluate the proficiency achieved in each tool by the subject. The proficiency of the subject was assessed by an expert tester and who was otherwise not involved directly in the case study. Modeling of JMYSTIQ in ***T0*** immediately followed the learning phase. Similar to the learning phase a log was maintained for recording the time and any special observations made by the subject during the modeling process.

Similar steps were repeated for ***T1***, ***T2*** and ***T3***.

In order to study Scalability, after completion of the modeling process for JMYSTIQ, the subject modeled the application Search Pubs.

Test cases were derived independently for JMYSTIQ and SearchPUBS using each test generation tool. The test suite and results of the test execution were recorded and were later analyzed. The test suites were assessed against the parsed requirements list (**A3**) prepared independently. The execution time and the test generation times for each model were recorded. The test results were also analyzed.

The final step in the case study was that of measurement and analysis. The data recorded during the test modeling phase, the test generation phase and the test execution phase were compiled for the four techniques and were analyzed. Figure 7.4 shows the outline of the case study.

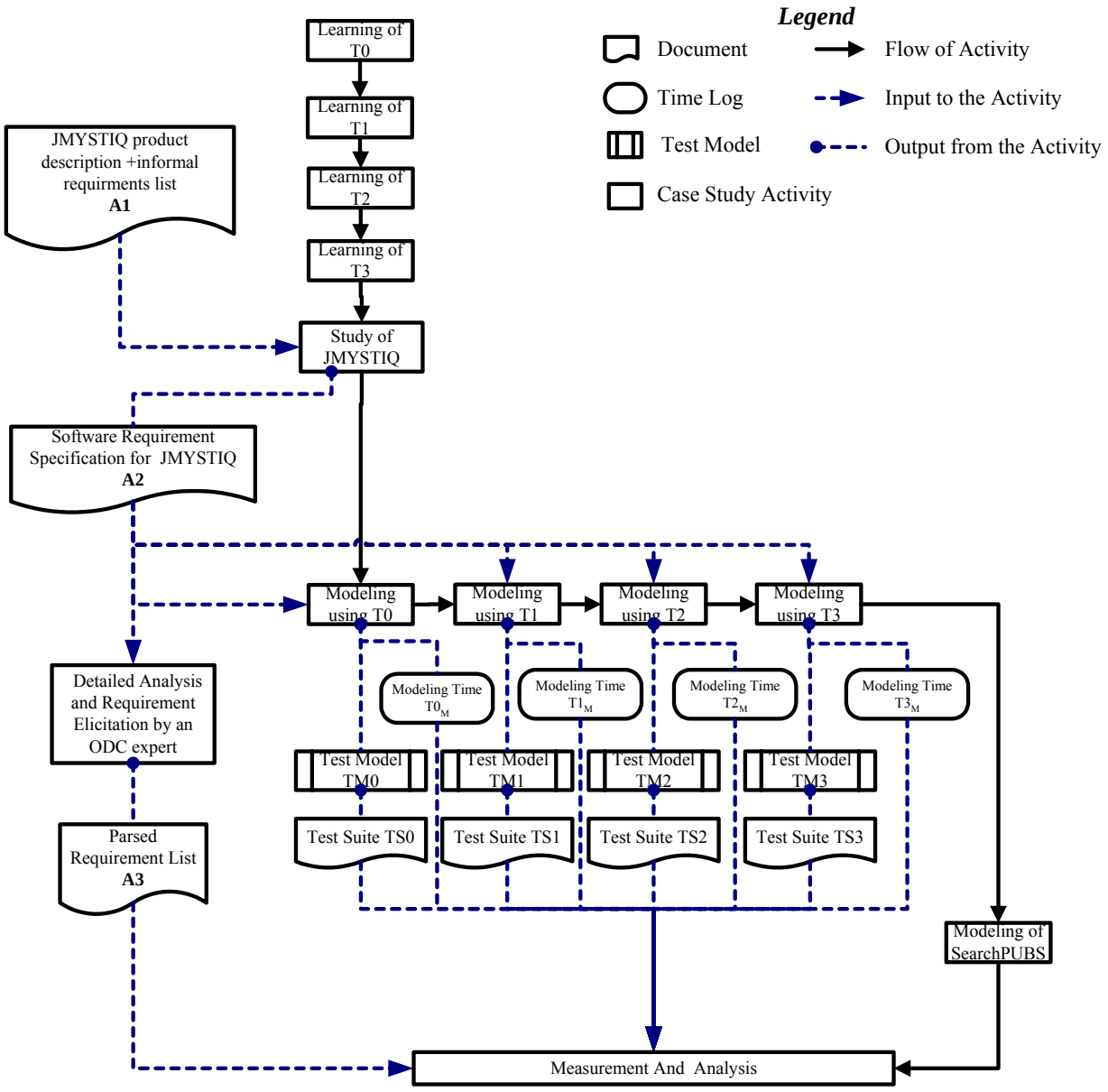

Figure 7.4: Different phases of the case study

7.2.4 Threats to Validity

Internal validity refers specifically to whether an experimental treatment/condition makes a difference or not, and whether there is sufficient

evidence to support the claim. [3] Possible threats to internal validity are identified in [3] as:

- **History**—History refers to the specific events which occur between the first and second measurement and that may affect the outcomes.
- **Maturation**—Maturation indicates the processes within subjects which act as a function of the passage of time.(e.g. hunger, aging, etc)
- **Testing**—Testing refers to the effects of taking a test prior to the experiment on the outcomes of the measurement.
- **Experimental Mortality** – Experimental Mortality refer to the loss of subjects during the experiment.
- **Instrumentation**—Instrumentation refers to the changes in the instrument, observers, or scorers which may produce changes in outcomes.

Since the case study was not conducted in a controlled environment, there was no threat because of *History* and *Maturation*. [3]The effect of *Testing* was reduced by isolating the study of the system from the study of the test design tools. The test of proficiency for each tool, prior to modeling of JMYSTIQ ascertained that the measurement results were not biased because of the continuous learning of the subject. The analysis and measurements were performed after completion of the modeling process and this was to ensure that there is no threat due to *Instrumentation* on the results.

The subject was asked to develop a natural language specification of the requirements of the application to be tested prior to the development of the test model. This ensured that the subject had reached the asymptotic level of the system's

natural learning curve before modeling the application. A single subject ensured minimal variation in the measurement due to personal bias and abilities. A major threat to validity was *Experimental Mortality* [3]. However, the whole time span of the experiment was about four months which is manageably small. Further, the subject was naturally motivated because the study was part of his summer project requirements.

7.3 Case Study Results

7.3.1 Complexity of the Modeling Process

For calculating the complexity of the modeling process the concepts for each tool were enumerated first. Table 7.1 lists the basic concepts needed in each modeling technique. As is seen in the table, the number of concepts in HaskellDB is the least. This is because HaskellDB is a domain specific language and the concepts in HaskellDB are limited and designed to be sufficient for the domain of database application.

HOTTest		**ArcheTest**		**ASMLT**		**EFSM**	
Concepts	*Prof*	*Concepts*	*Prof*	*Concepts*	*Prof.*	*Concepts*	*Prof.*
Function	0.9	prologue	0.9	state variables	1	states	1
polymorphic types	1	epilogue	1	stopping conditions	0.9	models	0.8
user defined types	0.9	consistency	1	update procedures	0.9	randomizations	0.2
basic types	1	activity mirroring	1	partial updates	0.9	type	0.9
Records	1	parameter partitioning	0.9	Methods	1	array i/o	0.2
Sequential flow	1	tofu combinations	1	Values	0.9	scope	0.2
juxtaposition	1	result definitions	1	Constraints	0.9	initialization	0.3
composition	1	context updates	1	Variables	1	IMCF	0.2
Recursion	0.8	execution templates	1	condition loops	1	table models	0.3
pattern matching	0.95	test data	1	Sets	0.9	context	0.3
case constructs	1	inheritance	0.9	Variables	1	events	1
If	1	inclusion	0.9	Constants	1	action	0.5
lists comprehensions	0.9	extension	0.9	Hyperstates	0.8	predicate	0.3
Relations	0.9	actors	1	Abstraction	1	argument	0.5
Attributes	0.8	associations	1	FSM generator	0.9	parameters	0.4
expressions	1	activity diagrams	1	Types	0.8	constraints	0.3
Query	0.9	classes	1	Instantiation	0.8	likelihood	0.5

Restrict	0.9			Sequences	1	path constraints	0.2
Project	0.95			Maps	0.9	"@ constraints"	0.2
set operators	1			non-determinism	1	test info	0.5
logical operators	1			Enumerations	0.9	test file set up	0.3
boolean connectors	1			Classes	0.9	shallow paths	0.5
				parameter generation	1	deep paths	0.5
						coverage scheme	0.5
Total # of Concepts	**22**		**17**		**23**		**24**

Table 7.1: List of Concepts for the tools

As discussed in section 7.2.1.2 the complexity of the test design tools is calculated using semantic dependency graphs. The semantic dependency graphs are constructed by relating the concepts through their hierarchical dependency. For instance in order to learn Archetest one needs to know certain UML concepts and certain concepts related to use case specifications. (Figure 7.7)

For calculating complexities we define two perspectives for each of the tools.

1. *Perspective of a Novice:* A novice is a person who is not familiar with any concepts related to the tool but is familiar with basic principles of test generation.
2. *Perspective of an Expert:* An expert is a person who is well versed with the underlying modeling principles (ASM for ASMLT, UML for Archetest, Haskell for HOTTest, and FSMs for EFSM) and who has past experience in design of test cases.

The complexities of the individual concepts are assessed using these perspectives. The complexity assignments for individual concepts are done during the measurement. The complexities for the nodes are assigned in accordance to the following guidelines:

1. A concept is assigned a complexity value of 3 if it is likely that the user has no prior experience with the concept.
2. A concept is assigned a complexity value of 2 if it is likely that the user has some experience with the concept or with a related concept.
3. A concept is assigned a complexity value of 1 if it is likely that the user has working experience with the concept or with a related concept.

The semantic dependency graphs for all the three tools are depicted in Figures 7.5 -7.12. We depict the dependency graphs along with the respective calculations. The complexities of the nodes have been calculated using equation 7.1. The complexity of the topmost node in the graph is the complexity of the test generation technique.

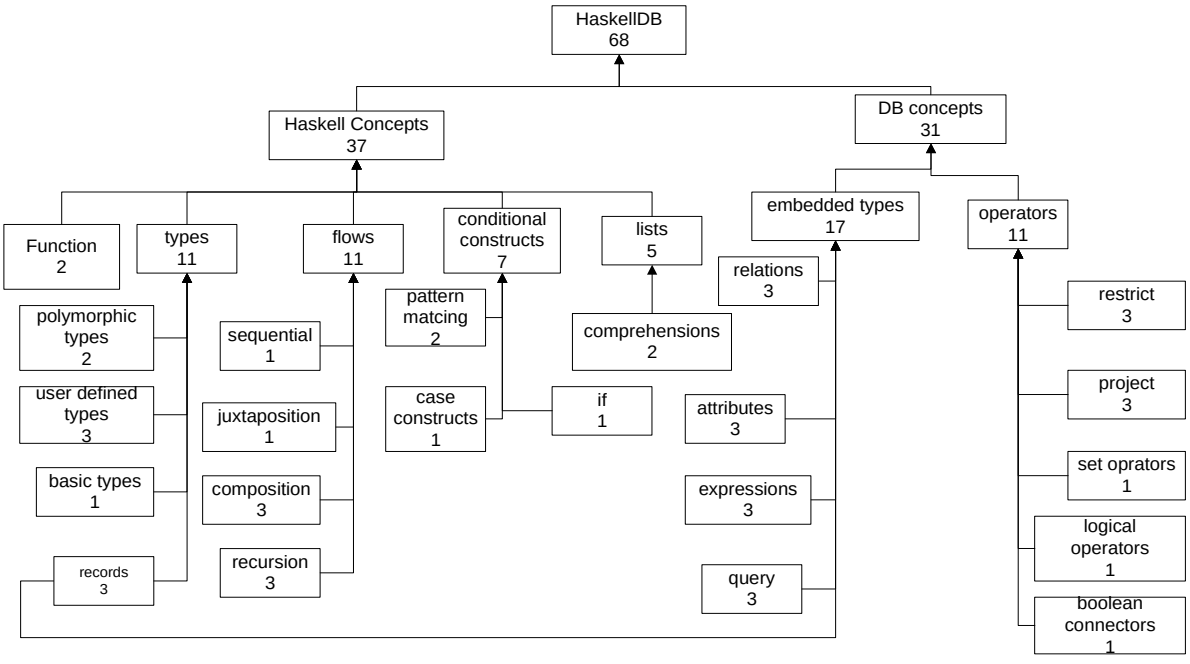

Figure 7.5 Semantic Dependency Graph for HOTTest- Novice's Perspective

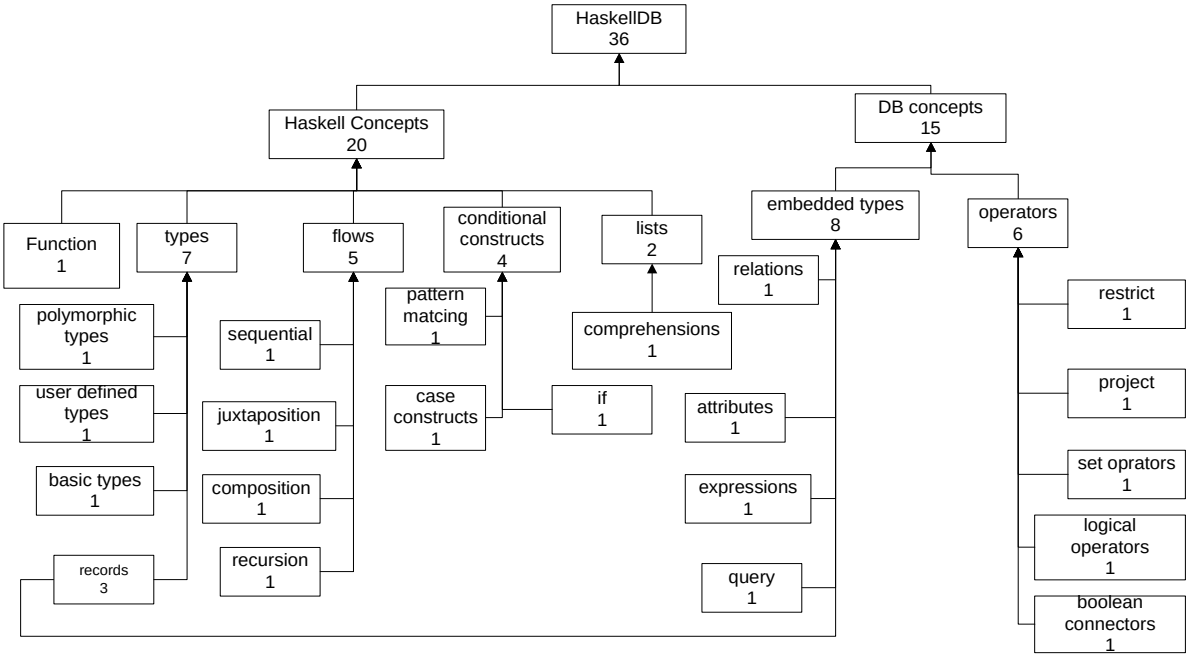

Figure 7.6 Semantic Dependency Graph for HOTTest- Expert's Perspective

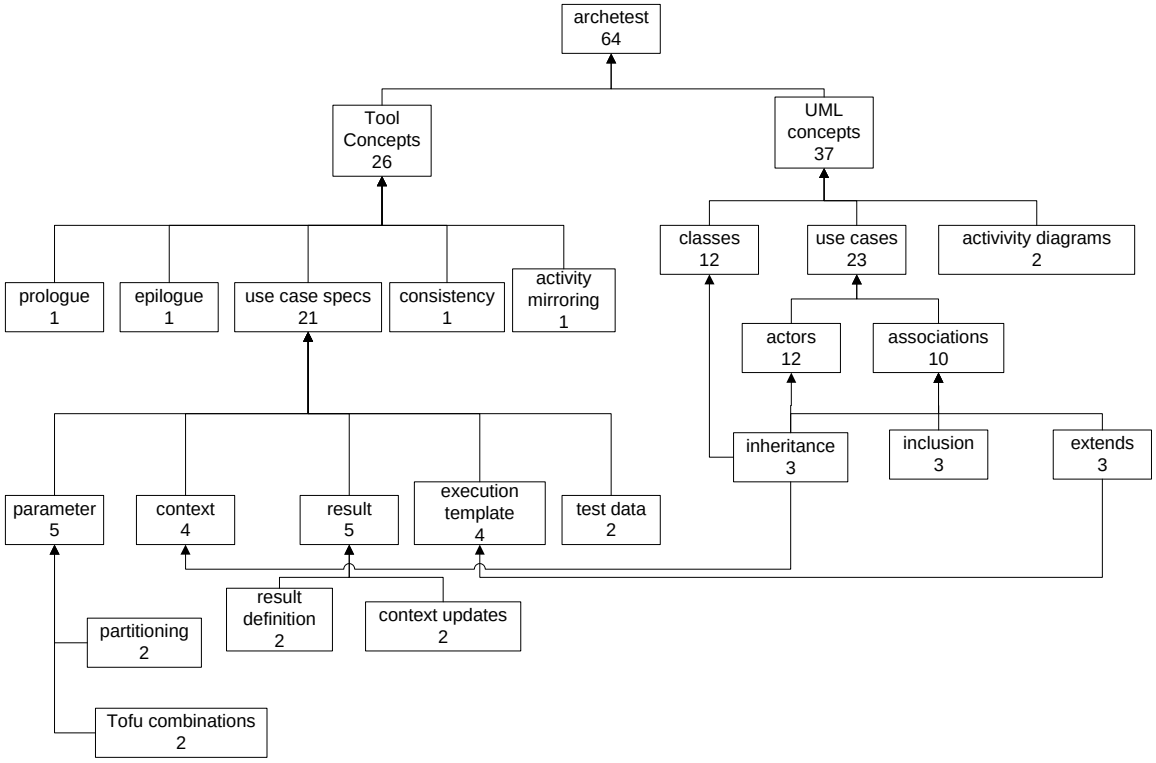

Figure 7.7 Semantic Dependency Graph for Archetest- Novice's Perspective

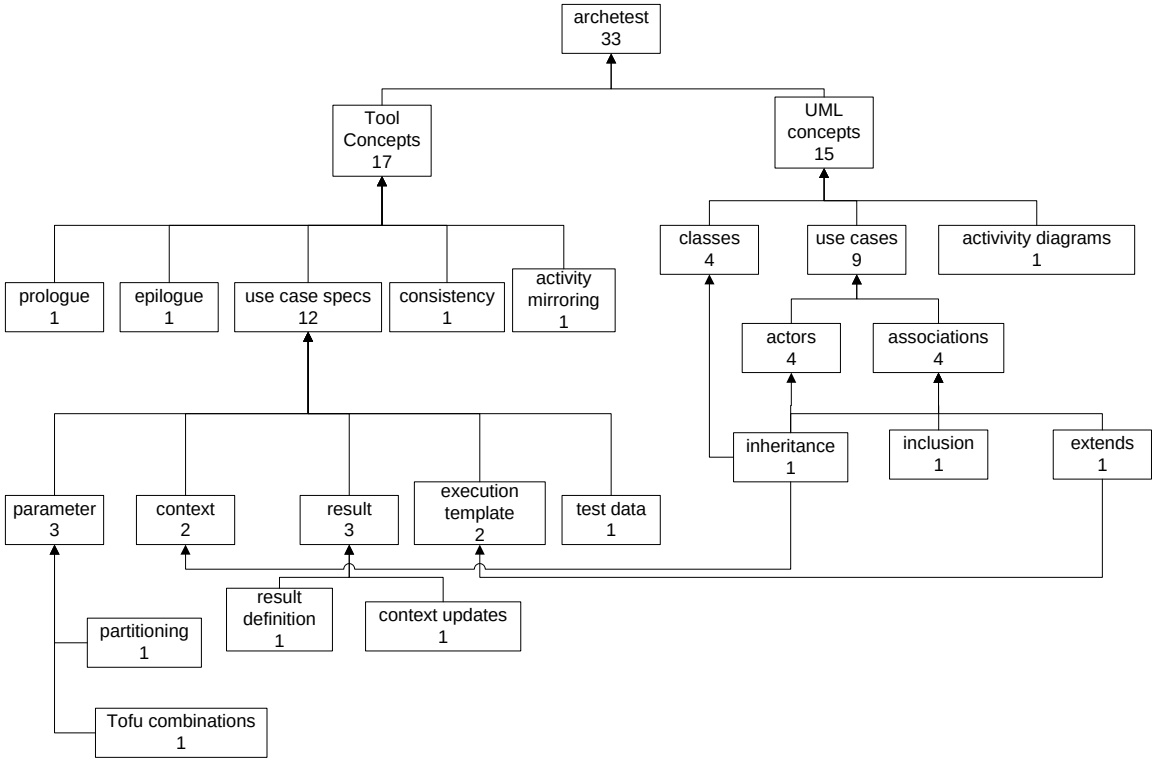

Figure 7.8 Semantic Dependency Graph for Archetest- Expert's Perspective

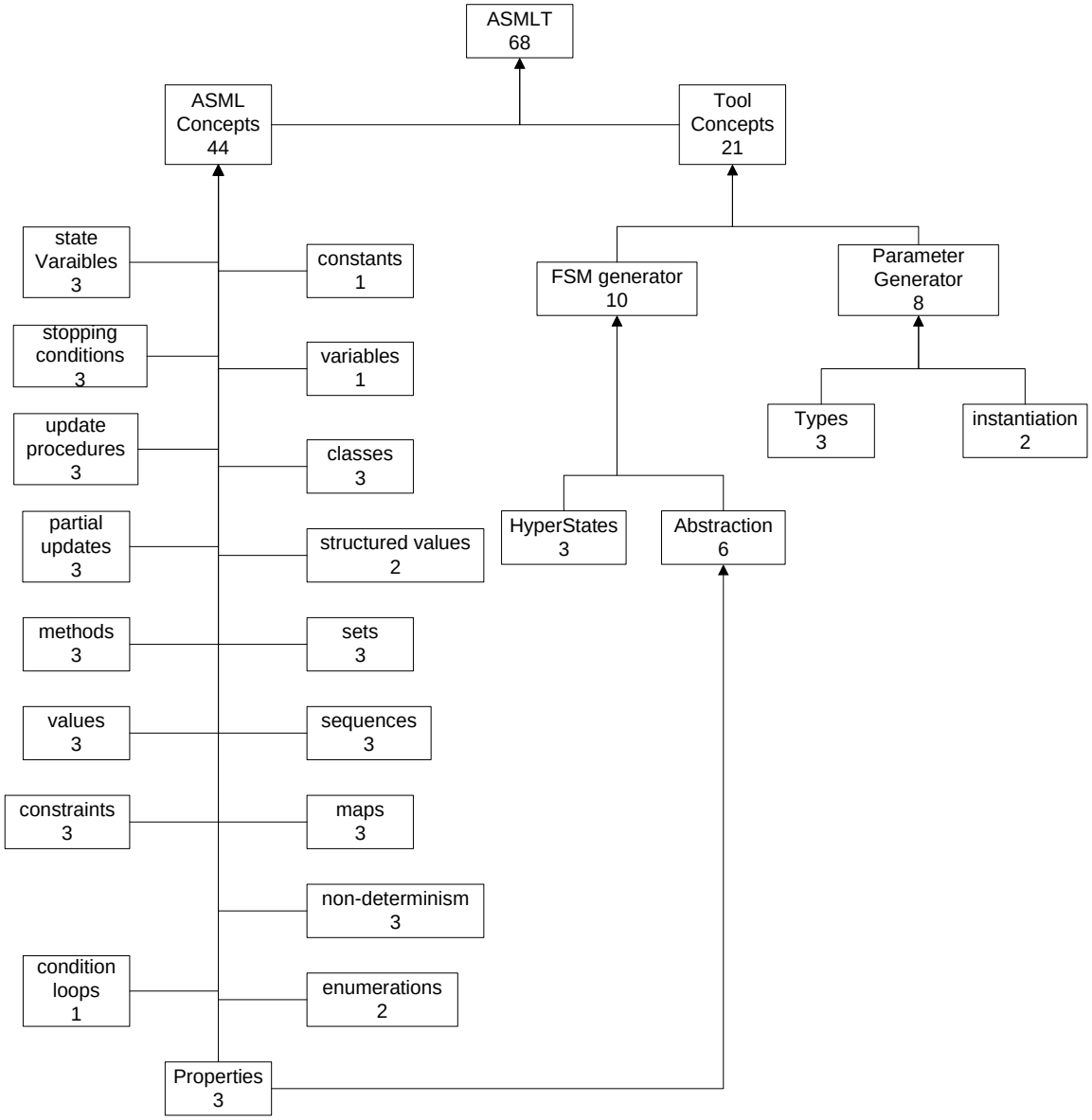

Figure 7.9 Semantic Dependency Graph for ASML- Novice's Perspective

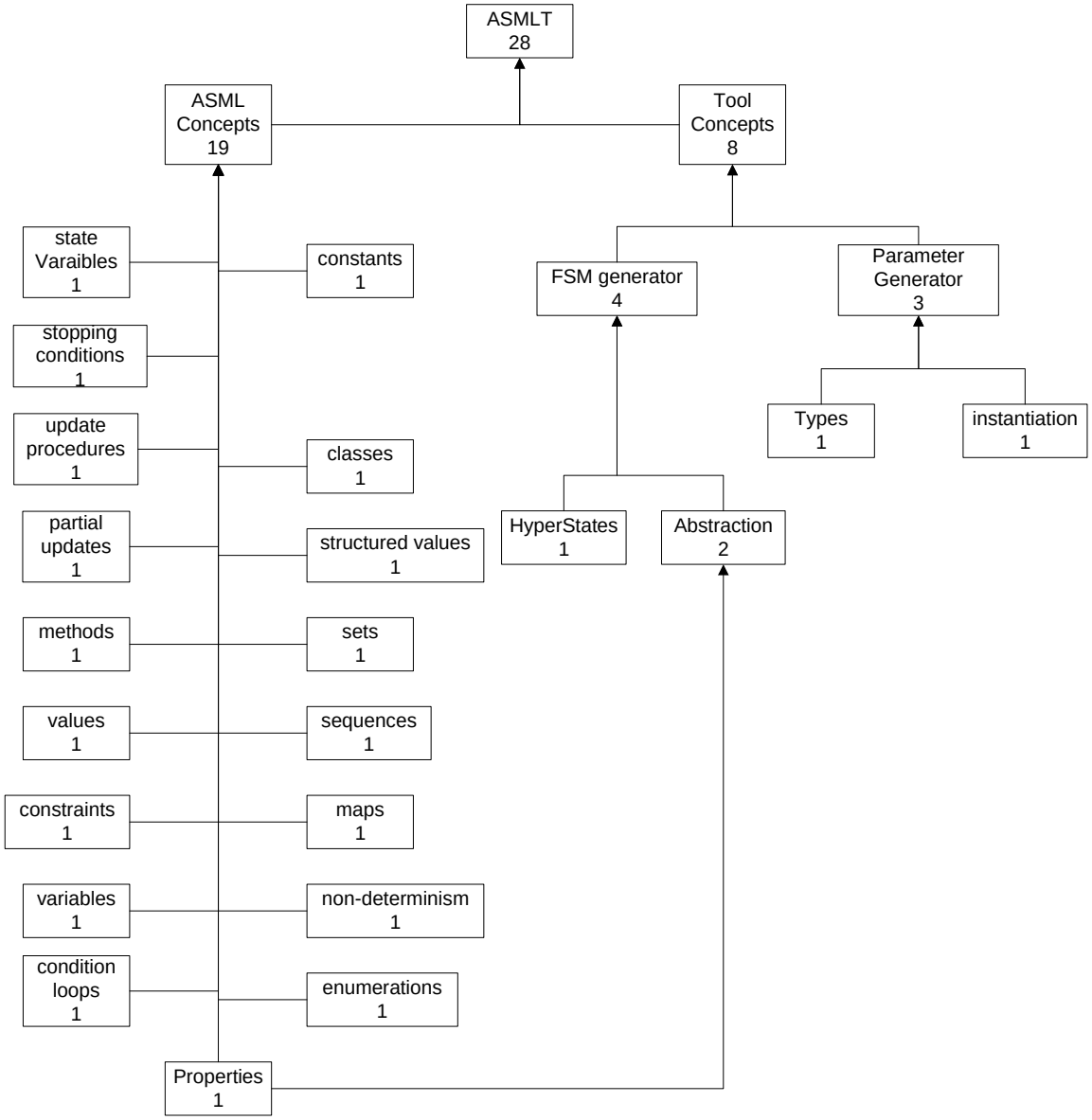

Figure 7.10 Semantic Dependency Graph for ASML- Expert's Perspective

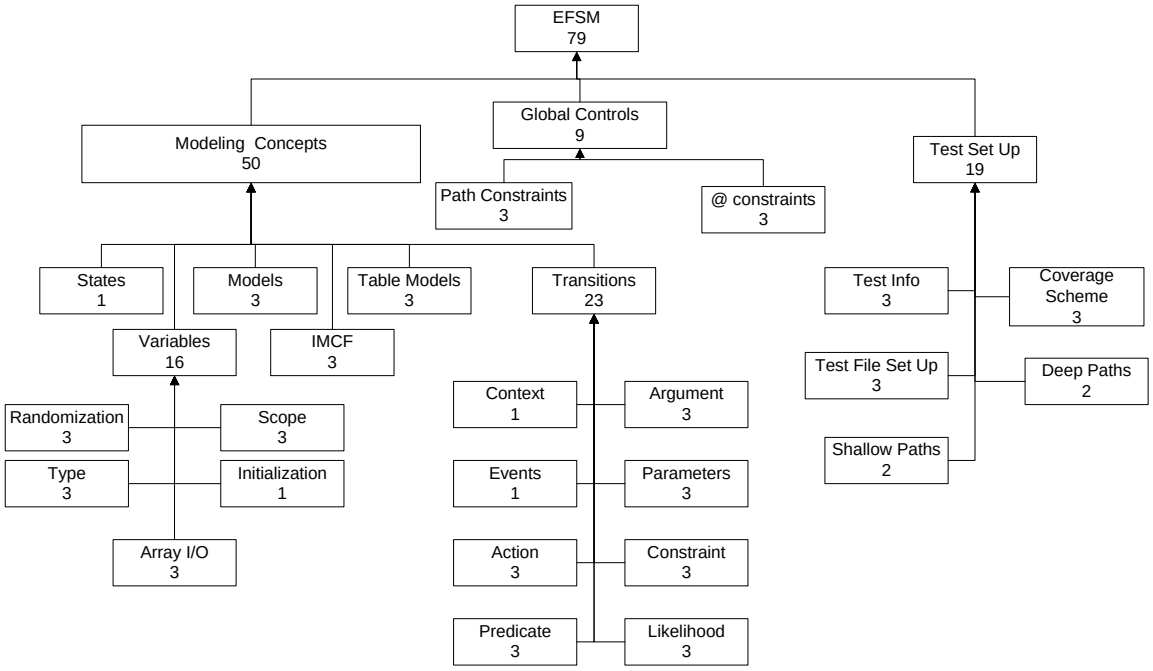

Figure 7.11 Semantic Dependency Graph for EFSM- Novice's Perspective

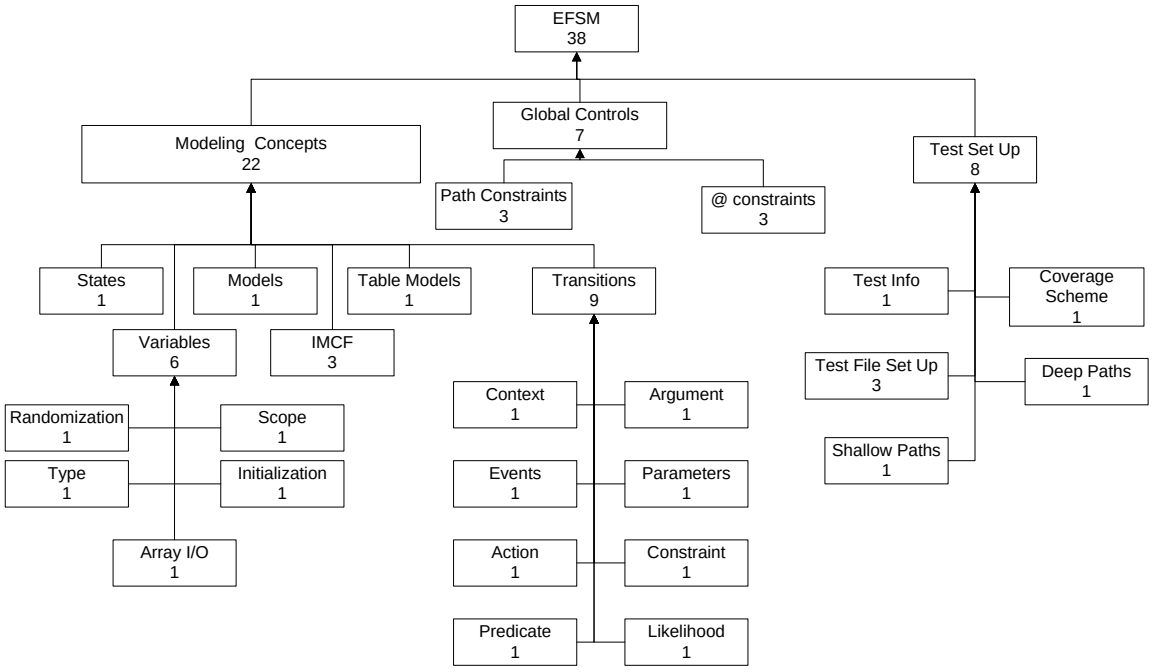

Figure 7.12 Semantic Dependency Graph for EFSM- Expert's Perspective

Figure 7.13 presents a comparison of the complexities of the tools from the two different perspectives.

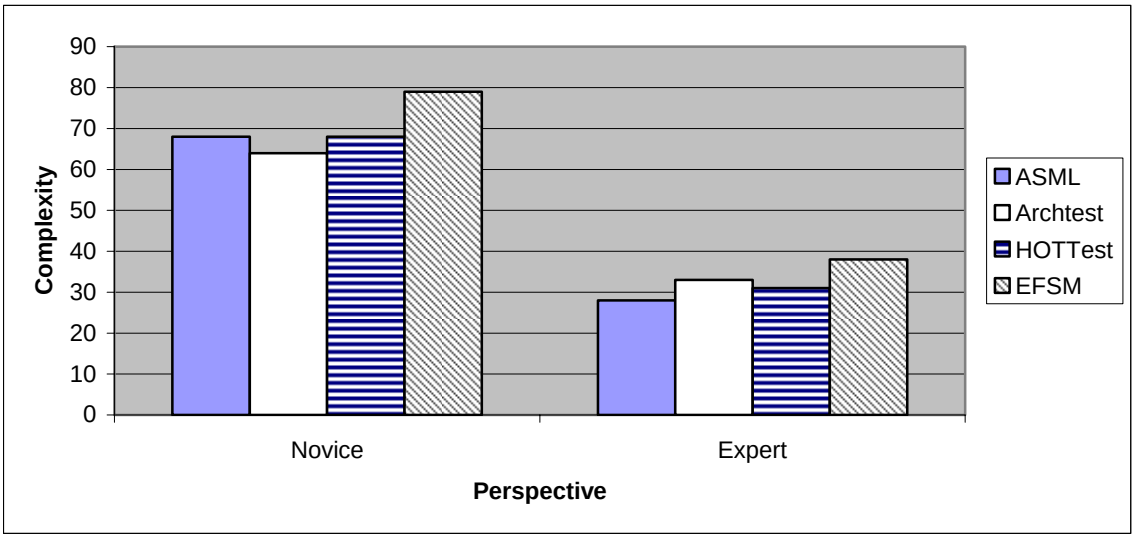

Figure 7.13: Comparative analysis of complexity of the tools

The complexity of the EFSM based tool is the maximum for both the perspectives (novice and expert). But for the other three tools the complexity of formalism are very similar to each other. This observation is interesting because this may imply that there is a threshold value for the complexity which any test generation tool must satisfy. The complexity measure is an indicator of how difficult it is to understand the modeling process for any person and therefore it can be inferred from the observations that if the tools' provide the same quality and quantity of support (for learning), they will need the equal effort for learning for users.

7.3.2 Ease of Learning

The ease of learning is calculated on the basis of the proficiency and the time to learn. The learning data was recorded on SSP. A list of concepts involved in each technique was prepared and proficiency of the user was measured on each such concept. The proficiency was measured using the equation number 7.2 as described in section 7.2.1.2 and the learning time was measured from the data logged during the recording phase. Table 7.1 logs the proficiency achieved by the subject in individual concepts of the test tools. Table 7.2 lists the measurements of the proficiency and the learning time. The measure for learnability is calculated using equation 7.4. Table 7.2 depicts the learnability values for the three techniques normalized to1.

	Total Learning Time	Learning Sessions	Modeling of Pilot	Proficiency	Learnability	Learnability Normalized
Archestest	14:44:00	3:56:00	10:48:00	0.9698413	1.5798319	0.912329844
ASML	22:21:00	13:18:00	9:03:00	0.9318182	1.0006101	0.57783772
HOTTest	13:10:00	5:00:00	8:10:00	0.95	1.7316456	1
EFSM	8:21:00	5:00:00	3:21:00	0.4428571	1.2728828	0.735071211

Table 7.2: Learnability Data for the tools

Figure 7.14 presents a comparison of the proficiencies achieved by the users while using the three test generation tools.

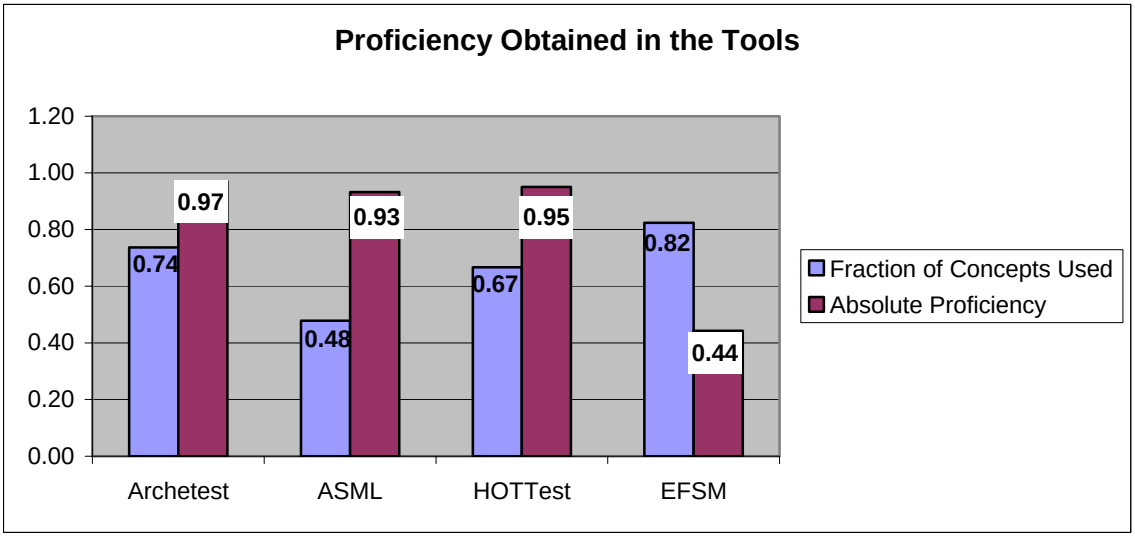

Figure 7.14: Comparative plots of the absolute proficiencies attained by the users

The user could achieve a proficiency of 0.97 with Archetest, 0.95 with HOTTest, 0.93 with ASML and only 0.44 with EFSM. The fraction of the total number of concepts used in the model is also depicted in the graph. This may give an indication of the sample models' ability to judge the proficiency of the user. Since, there was just one application used for studying learnability, the fraction of concepts used in the process is low for all the tools.

The learning time for the user is calculated by summing the time spent during the learning sessions and the time spent in developing the model. Figure 7.15 depicts a comparative plot of the learning times for the test generation tools. It also depicts the break-up of the learning times into times spent in learning sessions and the time spent in modeling the pilot. In Figure 7.16 a comparative plot of the learnability normalized to 1 is presented.

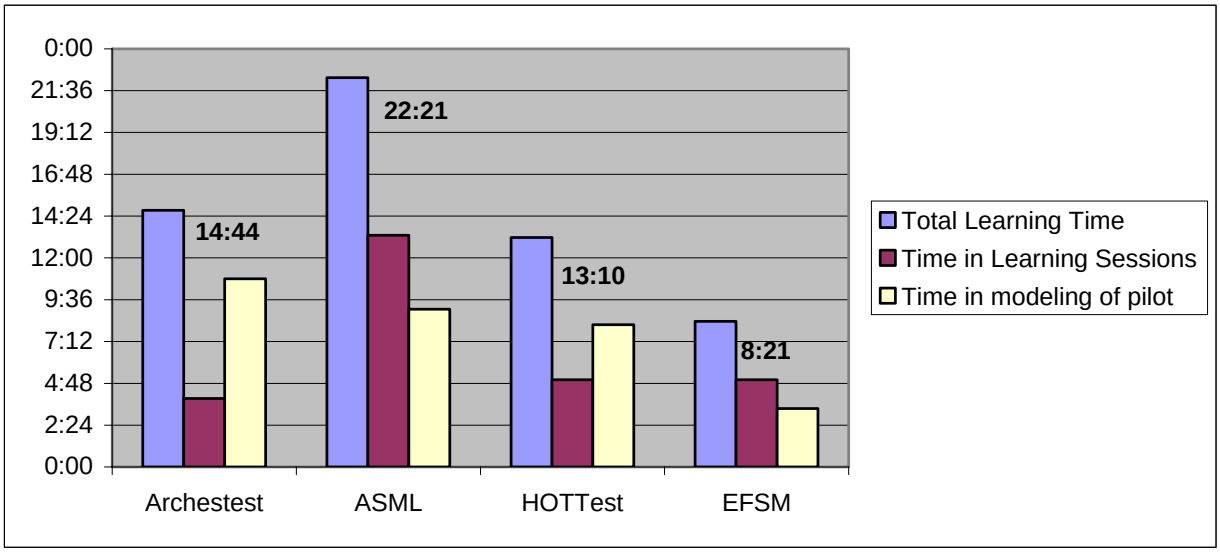

Figure 7.15: A comparative plot of the learning time

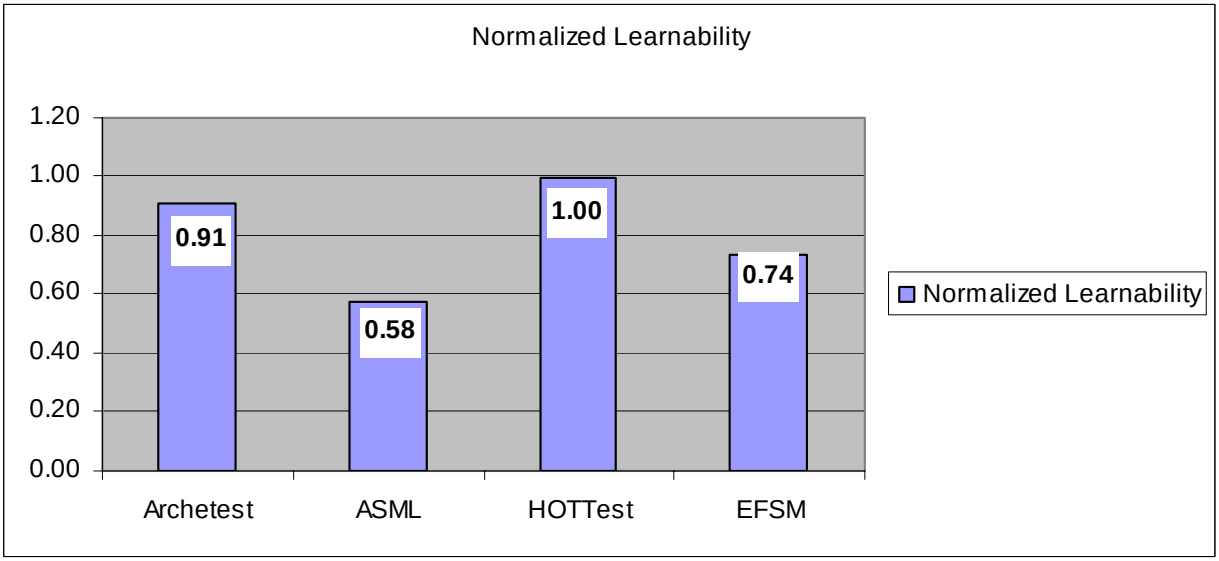

Figure 7.16: A comparative plot of the normalized values of learnability

The time to learn for ASML is 22 hours and 21 minutes whereas that for EFSM models in 8hrs and 21 minutes. The learning time for HOTTest is less than that of ASML but higher than that of EFSM. This is probably because the support materials available for the EFSM based test generator are more developed than the other tools. Also, the modeling of systems using finite states is intuitively more appealing to imperative programmers.

The learnability however, is the highest for HOTTest. This is primarily because the gain in absolute proficiency offsets the loss in time.

7.3.3 Effectiveness

The effectiveness values for the test generation techniques are calculated using equation 7.5. The coverage attained by the test models is measured using the parsed requirement list (artifact **A3**). It is computed by calculating the fraction of requirements in **A3** covered by the test suites. The net number of atomic requirements in **A3** for JMYSTIQ was counted to be *R* (=1260). Total number of requirements covered by the test models is depicted in the column with heading *r*. The number of requirements covered by any test model is calculated by examining the respective test suites generated using the technique (following maximum allowed coverage) and then by identifying the requirements tested by the test suite. A requirement is said to be tested by the test suite, if there are test cases in the test suite that would fail if that particular requirement is not satisfied by the application.

Each requirement in **A3** was later tagged as domain specific or generic requirement by a domain expert. A domain specific requirement was identified when the requirement was a requirement specific to the domain of relational-database-

applications. The total number of requirements in **A3** was 1260 (R) out of which 602 (D) were identified to be specific to the domain of database applications. 210 of the 602 net domain specific requirements were not explicitly mentioned in the original requirements document (**A2**). Some of the 210 implicit requirements were identified by the expert; others were identified while testing with HOTTest. The number of requirements covered using the test techniques (r) and the number of domain specific requirements covered using the technique (d) along with the calculated values of effectiveness of techniques is presented below in Table 7.3

	Measures→					
	R	*r*	*D*	*d*	Effectiveness (%)	Domain Specific Effectiveness (%)
Technique						
HOTTest	1260	1216	602	577	96.51	95.85
ASMLT	1260	965	602	307	76.59	51.00
Archetest	1260	1050	602	392	83.33	65.12
EFSM	1260	843	602	185	66.90	30.73

Table 7.3: Effectiveness and Domain-specific effectiveness calculations

The test model in HOTTest captures the most number of requirements (1216) while that in EFSM captures the least number of requirements (843). HOTTest has an effectiveness of 96.51% and a domain specific effectiveness of 95.85%. The other techniques have a fair value for generic effectiveness but they fail to perform likewise for domain specific effectiveness. Figure 7.17 makes a comparative plot of the effectiveness values for various techniques.

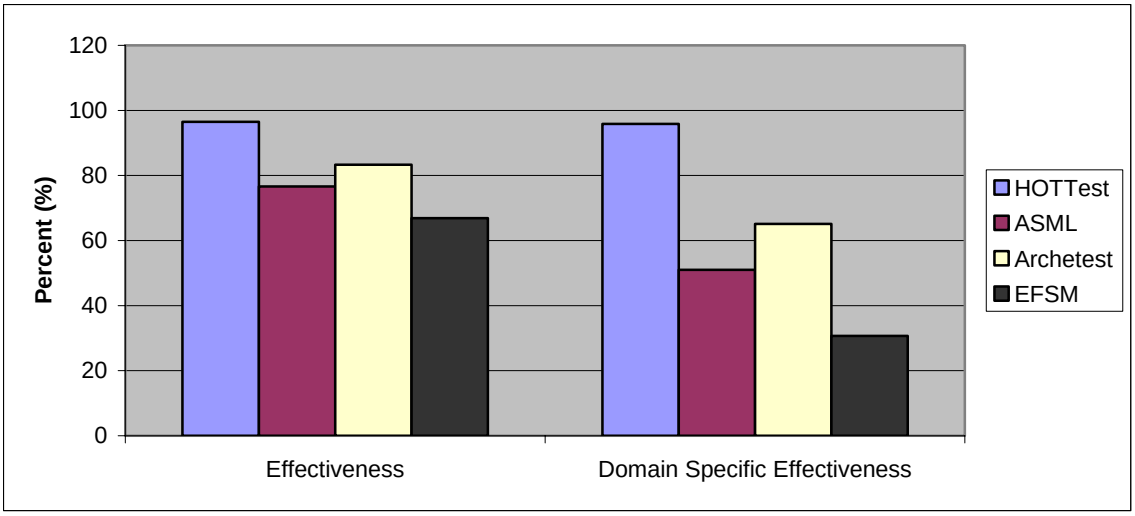

Figure 7.17: Comparative plots of effectiveness values

7.3.4 Efficiency

To calculate efficiency we first need to compute the effort needed in testing using the test generation techniques. The net effort in testing is calculated using equation 7.8 in the measurement model specified in the section 7.2.1.2. Table 7.4 presents the data for effort calculations.

	T_M(mins)	T_G(mins)	T_S(mins)	T_X(mins)	N	T_{MM}(mins)
Archetest[10]	1554.00	--	--	--	--	--
ASML	1356.00	1584.00	1822.00	1309.73	4180	6071.73
EFSM	3030.00	73.00	132.00	1322.89	4222	4557.89
HOTTest	1093.00	83.00	140.00	2052.96	6552	3368.96

Table 7.4: The data for testing effort

The time to model T_M for HOTTest is the least at 1093 minutes and that for EFSM is the most at 3030 minutes. The difference arises mainly due to the fact that modeling in HOTTest requires translation of information from one textual form to another whereas there is a translation into EFSM concepts. The latter demands good

[10] Because of some technical difficulties, access to Archetest was not available for test case generation and execution.

abstraction skills from the modeler. For the same reason the time to model for ASML is comparatively low. However, since the ASM compiler is inefficient, the gain in modeling time is compromised.

The time to generate test cases from the test model T_G is enormously high for ASML. In fact, it was impossible to generate an exhaustive set of test cases from the ASML model without the abstraction rules. This is because the state space for the ASML model is huge and the algorithm for test case generation is still primitive (Chinese Postman Algorithm). T_Gs for the other tools are comparable. It should be less for Archetest that uses innovative, path generation algorithms based on AI planning.

The time for setting up the test environment T_S, includes the time to set up the test harness (Rational XDE tester[10]) and the time to derive the test scripts from the test cases. EFSM and HOTTest have the capabilities of producing test scripts directly from the test model, but the tester needs to map the execution sequence of the application to a sequence of states and transitions. This process is error- prone and contributes heavily towards the test set-up time. The test cases produced by ASML are abstract and the process of translating a test case to a test script is a complete manual process. Therefore, the time to set-up for ASML is the most. Archetest directly produces test script for the test harness and therefore T_S for Archetest should be less than other tools.

N is the number of test cases in the test suite produced from the model following the full coverage scheme. The number of test cases produced with HOTTest is much higher than any of the other test generation tools. This is because

HOTTest generates an extra test case for every domain specific requirement. The time to execute T_X is proportional to the number of test cases generated. Therefore, T_X is highest for HOTTest. This is a compromise one has to do in order to achieve higher test coverage.

The net effort in testing T_{MM} is calculated using equation 7.8. The net effort is highest for ASML primarily due to huge generation and set-up times. The net effort for EFSM is also high because of a huge modeling time. Archetest requires the least effort in testing and is followed by HOTTest. A comparative plot of the effort calculations is presented in Figure 7.18 for the tools.

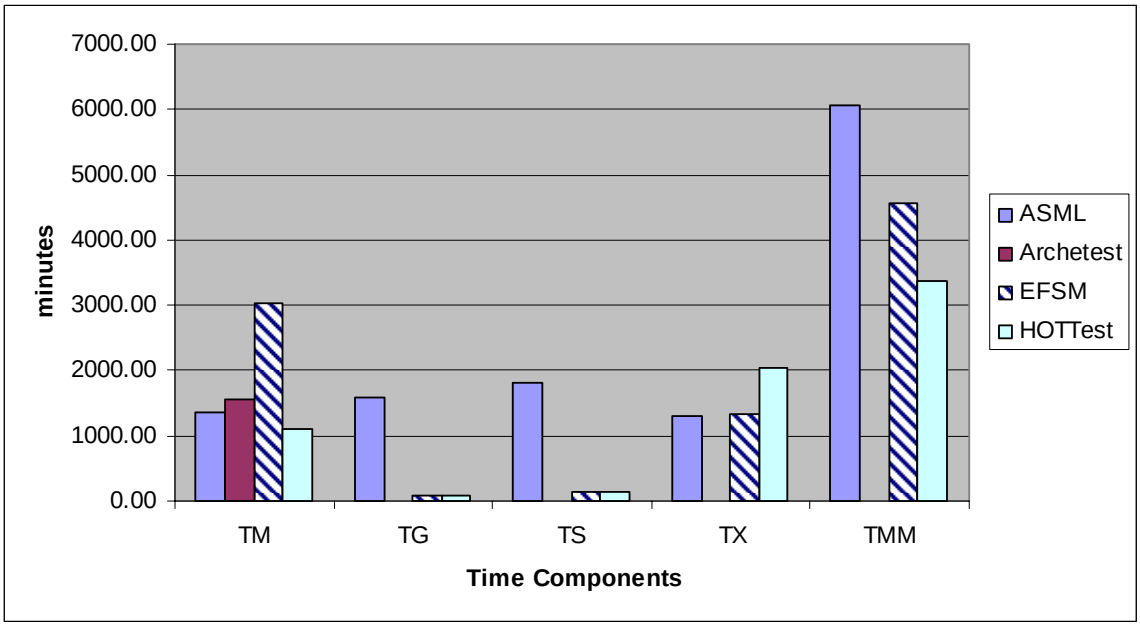

Figure 7.18: A comparative plot of the net effort and its contributors

Efficiency of the tools is calculated using equation 7.6. The coverage attained by the test models is measured using the parsed requirement list (artifact **A3**) and is discussed in section 7.3.3. Since, for each of the test models the application was the

same (that being JMYSTIQ), therefore we don't evaluate the $C_{Complexity}$**Application Size*. This factor eventually cancels out when we do a comparative analysis. Table 7.5 presents the results of the efficiency calculations.

	R	R	T_{MM}	Coverage	Efficiency	Normalized Efficiency
ASML	1260	965	6071.73	76.59	0.013	39.90
Archetest	1260	1050	--	83.33	--	--
EFSM	1260	843	4557.89	66.90	0.015	46.43
HOTTest	1260	1216	3368.96	96.51	0.029	90.61

Table 7.5: Efficiency measurements

Efficiency measurements were normalized to 100 to allow better comparisons. A comparative plot for the normalized efficiency values for the test generation techniques is presented in Figure 7.19.

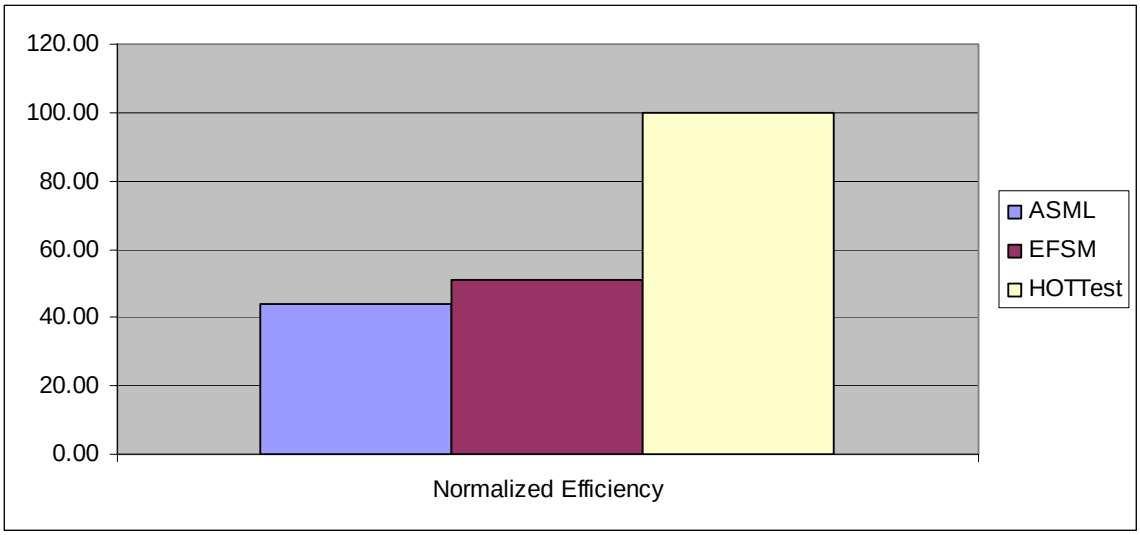

Figure 7.19: A Comparative Plot of Efficiency Calculations

7.3.5 Scalability

As mentioned before, scalability for the tools was measured by measuring effort on two applications, viz. JMYSTIQ and SearchPUBS. The effort data for

JMYSTIQ is presented in section 7.3.4. The measure of effort contributors and the net effort in the modeling of SearchPUBS are presented below in Table 7.6

	T_M	T_G	T_S	T_X	N	T_{MM}
ASML	546	17.00	95.00	23.50	75	681.50
Archetest	518	2.00	15.00	10.03	32	545.03
EFSM	201	0.70	45.00	6.89	22	253.59
HOTTest	490	12.00	50.00	26.63	85	578.63

Table 7.6: The data for testing effort for SearchPUBS

Scalability of effort is calculated for various contributors of the test effort in accordance to equation 7.8. Scalability values were normalized to 100 to allow a better comparison. Table 7.7 presents the scalability measures for the various tools for different effort contributors and the net effort.

	Absolute Scalability					*Scalability normalized to 100*				
	S_M	S_G	S_S	S_X	S_{MM}	S_M	S_G	S_S	S_X	S_{MM}
ASML	1.23E-03	6.38E-04	5.79E-04	7.77E-04	1.86E-04	74.44	4.53	5.04	100.00	51.77
Archetest	9.65E-04	--	--	--	--	58.20	--	--	--	--
EFSM	3.53E-04	1.38E-02	1.15E-02	7.60E-04	2.32E-04	21.31	98.20	100.00	97.74	64.83
HOTTest	1.66E-03	1.41E-02	1.11E-02	4.94E-04	3.58E-04	100.00	100.00	96.67	63.48	100.00

Table 7.7: Absolute and Normalized Scalability Values for the Tools

The scalability for modeling time for HOTTest was the highest followed by that of ASML. The scalability value of modeling for EFSM is the least. This indicates that with increase in size of the system under test, it becomes increasingly easier to describe functionalities in a textual representation rather than using some abstract concept.

Due to some technical difficulties Archetest was not available for most scalability studies. It is expected that Archetest will have good scalability with respect to the

time to generate test cases, the time to set up the test environment and the time to execute the test cases.

The scalability for EFSM and HOTTest with regards to the time to generate test cases is comparably close. However, the scalability for the time to generate is much lower for ASML. This is probably due to the fact that the state space for the finite state machine structure used by ASML for test generation soon becomes unmanageable with increase in size of the system under test.

The scalability values for the test set up time show trends similar to the values for test generation time. The very low scalability for ASML is ascribed to the fact that there is no mechanism to generate test scripts automatically from the test case directives generated using ASML.

The scalability with regards to the execution time is the lowest for HOTTest. This is probably due to the fact that the number of test cases generated in HOTTest increases faster with the size of the system than the other techniques. With increasing number of test cases the execution time suffers.

The scalability for HOTTest with regards to the net effort is the highest followed by EFSM. The weakness in scalability value of ASML is contributed by its weak scalability of the effort to generate test cases and of the effort to set up the test environment. Figure 7.20 presents a comparative plot of the normalized scalability values for the four test generation techniques.

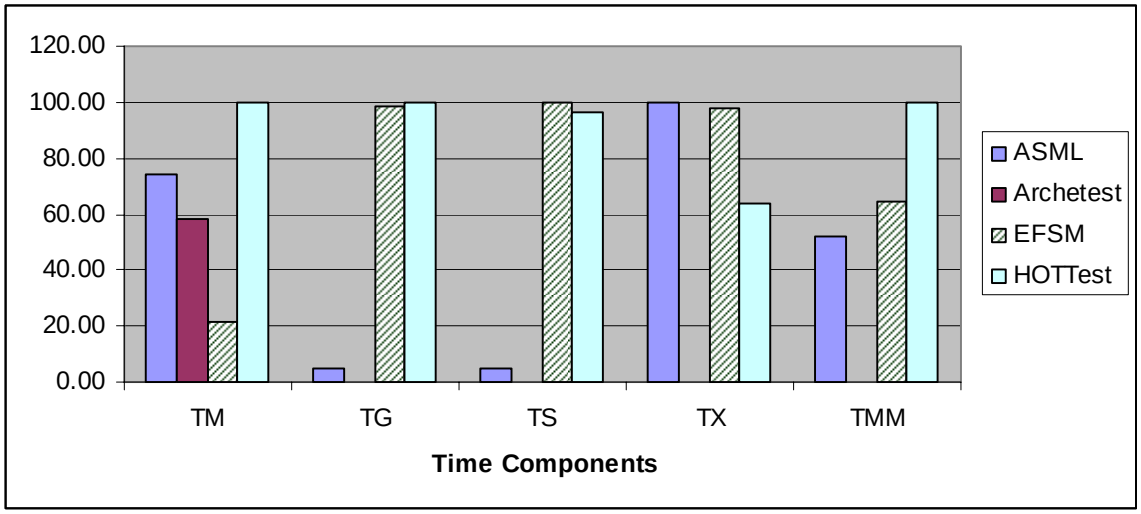

Figure 7.20: Comparative Plots of Normalized Scalability

7.4 Analysis of the Results

A summary of measurements for the metrics selected at the quantitative level of the GQM (see section 7.2.1.2) is presented in Table 7.8.

Metric	Corresponding Question	HOTTest	ASML	Archetest	EFSM
Complexity of Technique (Novice)	**Q1**	68	68	64	79
Complexity of Technique (Expert)	**Q1**	36	28	33	38
Ease of Learning	**Q2**	1.00	0.57	0.91	0.73
Effectiveness	**Q3**	96.51	76.59	83.33	66.90
Efficiency	**Q4**	90.61	39.90	100.00	46.43
Scalability	**Q5**	74.94	38.74	100.00	48.58

Table 7.8: Summary of Measurement Results.

It can be seen from Table 7.8, that except EFSM the complexities of all the test generation tools are comparably close from both the perspectives (Novices and Experts). This implies that the tools are equally hard or equally easy to use, once the learning is complete. The complexity of EFSM is higher than the others because the technique provides too many concepts to master with regards to test generation

related concepts. While, these concepts should make test specifications more effective, they easily can add complexity to the technique.

As for the ease of learning of tools, HOTTest is easier to learn than the other techniques. The subject learnt it better, and committed errors less frequently. The proficiency evaluation was done on SSP. The subject committed most errors in ASML. This is probably because of the poor error feedback provided by ASML. This is also reflected in the highest debugging time for ASML in the modeling of SSP. (401 minutes for ASML compared to 75 minutes for EFSM, 121 minutes for Archetest and 32 minutes for HOTTest). For both Archetest and EFSM the debugging time is lower, because they provide better error feedback and tracking. For HOTTest the static type checker helps commit fewer errors and aids in error tracking.

Effectiveness of HOTTest is highest because it captures the implicit requirements along with the ones explicitly stated in requirements document (**A2**). Also the number of domain specific requirements tested using HOTTest is higher than any other technique. This was expected because HOTTest automatically embeds additional test cases to capture for requirements that were not explicitly mentioned in the original requirements document (Artifact **A2**). None of the other techniques were capable of testing the implicit domain specific requirements. While Archetest was able to capture all 392 of the domain specific requirements explicitly mentioned in **A2**, ASML could capture only 307 of them. (See Table 7.3) ASML could not capture all the explicit domain specific requirements, because of the stringent abstraction properties without which test case generation was impossible. EFSM covered least number of domain specific requirements (185) primarily because the subject could

not embed the requirements in an EFSM whose state space was unmanageably large. The domain specific requirements missed by various test generation techniques were classified further as shown in Table 7.9. The table shows the net number of requirements that are missing and classifies the missed requirements into the requirements related to DbConnection, Missing Fields, Wrong Types, Security and Constraints. (See Chapter 3).

Technique	Missing	Db Connection	Missing Fields	Wrong Types	Security	Constraints
HOTTest	25	0.00	72.00	28.00	0.00	0.00
ASML	295	1.69	59.66	38.64	0.00	0.00
Archetest	210	1.43	57.14	41.43	0.00	0.00
EFSM	417	44.60	32.61	22.78	0.00	0.00

Table 7.9: Classification of the domain specific requirements missed by the techniques

HOTTest, however, could capture only 639 generic requirements out of a total of 658 requirements for the application. This inability was mainly due to the restrained expressive power of HaskellDB which resulted in inability to capture the functionalities related to date and system time. All the other test generation techniques were capable of capturing all 658 generic requirements.

The most efficient tool for test generation purposes was observed to be HOTTest at 90.61. As is discussed during the effort calculations cited in section 7.3.4, HOTTest however, has a very poor execution time. This is due to the huge number of redundant test cases that HOTTest generates. It should be noted that the time to execute and the time to generate the test cases are pure machine times and they don't need any manual effort. If we count only the manual effort to be the effort in testing then we get modified values of efficiency, as is depicted in Table 7.10

	T_M *(mins)*	T_S *(mins)*	*Modified* T_{MM} *(mins)*	Coverage (%)	Efficiency	Normalized Efficiency (%)
ASML	1356.00	1822.00	3178.00	76.59	0.024	30.79
EFSM	3030.00	132.00	3162.00	66.90	0.021	27.03
HOTTest	1093.00	140.00	1233.00	96.51	0.078	100.00

Table 7.10: Efficiency Calculations considering only the Contributions of Manual Effort

Efficiency values clearly depict the advantage HOTTest has over the other tools in terms of the modeling time. This further proves that in large systems it is easier to model in a textual representation than using some other concepts.

In order to set benchmarks for effort calculations, data were collected from past industrial test activities in IBM. Table 7.11 presents test effort data collected from 10 projects within IBM.

KLOC	Person Months
0.5	3
1	4
1	5.5
1.25	4
5	36
7.45	14
11.3	39.5
12.75	32
80	168

Table 7.11: Test effort data from IBM projects

Considering a person month to be equal to 152 hours [1], it can be said that model based testing is definitely rewarding. According to the data, if we do linear interpolation, then for a 50Kloc application, the estimated effort needed in testing is given as 106.8 person months. And when the values of net effort for the test generation tools are compared to the estimated value, every technique has excellent rewards. The highest effort was needed for ASML and it was only 0.66 %(~1%) of the estimate.

Scalability values for individual components of total effort are compared in section 7.3.5. The comparison shows a clear lead for Archetest. Archetest scales up for all effort contributors except for the modeling time. HOTTest performs poorly for execution time but is very good for all the other contributors. Scalability of ASML is very poor in consideration of components like time to generate, time to set up and the time to execute.

7.5 Summary

In summary it can be said that all model based test generation tools have great advantages in terms of effort reduction. Among the tools compared for industrial applicability, HOTTest is 96.51% effective in covering requirements and is most efficient. It has the highest learnability and a complexity comparable to the other techniques. The scalability for HOTTest when considering the modeling time is 100%.

HOTTest has a very high execution time and misses out on some of the requirements. The reason for the high execution time is the huge number of redundant test cases. A dependency analysis of the generated EFSM in HOTTest can reduce the number of test cases and can reduce the execution time. To capture the kind of requirements missed by HOTTest, the underlying domain specific language (in this case HaskellDB) needs to be expanded by embedding types and functions.

7.6 References

[1] B. W. Boehm, C. Abts, A.W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. Reifer, B. Steece, *Software Cost Estimation With COCOMO II.* Prentice Hall, July 2000.

[2] C. E. Williams, "Toward a test-ready meta-model for use cases," *in Proc. Workshop on Practical UML-based Rigorous Development Methods,* Toronto, CA, 2001, 270–287.

[3] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Designs for Research.* Chicago: Rand McNally and Company, 1963.

[4] *IEEE Guide to Software Requirements Specification,* IEEE Standard 830, 1984.

[5] J. Siegel, *Introduction to OMG UML,* OMG Consortium, online http://www.omg.org/gettingstarted/what_is_uml.htm

[6] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, "Model-Based Testing with AsmL.NET," *in Proc. 1st European Conference on Model-Driven Software Engineering ,* December 2003.

[7] M. Kwan. Graphic programming using odd and even points. *Chinese Math.*, 1, pp. 273-277, 1962

[8] P. Savage, S. Walters and M. Stephenson, " Automated Test Methodology for Operational Flight Programs," in *Proc. IEEE Aerospace Conference,* vol.4, pp. 293-305, 1997.

[9] R. Chillarege, I.S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Y. Wong, "Orthogonal Defect Classification – A Concept for In-Process Measurements," in *IEEE Transactions on Software Engineering,* Nov 1992.

[10] *Rational XDE Tester User's Guide,* IBM Corporation., New York, NY, 2004.

[11] *Rational Rose Enterprise Edition*, IBM Corporation, New York, NY 2004.

[12] V. Basili, " Software Quality Assurance and Measurement: A Worldwide perspective", in *Applying the Goal/question/Metric Paradigm in the Experience factory*, Chapter 2, pp 21- 44, International Thomson Computer Press, ITP An International Thomson Publishing Company, 1995.

Chapter 8 Extension to other Domains

In the previous chapters we discovered HOTTest's ability to capture domain specific requirements and generate test cases accordingly. The concept was proven for the domain of database applications. This chapter addresses extension of HOTTest's capabilities to other domains of application. The study presented in this chapter is preliminary and is not a complete study of the subject. This can help in directing the future research.

8.1 Steps for Extension

The basic steps for extending HOTTest to other application domains are as follows:

8.1.1 Domain Analysis

The first step in extending HOTTest to a new domain starts with domain analysis. During domain analysis various applications of the domain are studied and a generic understanding on the domain is achieved. Following this, the domain specific requirements are identified. Identification of domain specific requirements is a two step activity:

- Identify the global requirements of the domain (D_G): These are the requirements that are true for any application of the domain, e.g. the connection specific properties for the domain of database applications like the security properties, the availability properties etc. These properties need to be satisfied for all applications for that domain and is not dependent on some specific functionality of the application.

- Identify the local functional requirements of the domain applications (D_S): These are the requirements specific to the functional features of an application. For instance for the database domain any update operators should satisfy integrity constraints. Such a requirement is valid only for the database applications that perform an update operation.

8.1.2 Design/ Choice of a Domain Specific Language

After a detailed understanding of the domain, a strongly typed domain specific language is designed or chosen. The use of Haskell as the parent language allows seamless integration with the existing tool. The criteria for choice of the DSL include the following features of the DSL:

- *The level of abstraction that the DSL provides*: If the level of abstraction supported by the DSL is too high, it might be very difficult to specify every requirement of the application. A very low level of abstraction on the other hand makes the DSL too complex and again makes the specification process a difficult one.
- *The types, constructs and the functions that the DSL provides:* The DSL types, constructs and the function should be such that they can express the domain requirements properly. Too many functions and types make the DSL difficult to use, but if they are too few then it might be difficult for the user to express all the requirements. Therefore, a balance needs to be struck.
- *The usability of the DSL*: The DSL should not be too complicated to use. The complication in use of a DSL may be affected by one of the factors mentioned above but it may also be affected by other factors like complexity of

formalism, available support, etc. Therefore, independent assessment of usability is important.

- *The extensibility of the DSL to accommodate more domain concepts:* The DSL should be extensible easily in order to accommodate the need to expand and capture more domain requirements.

In absence of an appropriate DSL, a new DSL for test design could be designed by adding required types and library functions in an existing Higher Ordered Typed generic language. The choice of the embedded types and the library functions is a multi-step activity that can be perfected only over years of use. The concept of polymorphic types and type hierarchy supported by Haskell makes it easier to embed types in Haskell.

8.1.3 Associate Requirements to DSL constructs

The next step is to associate the specific functional requirements (D_s) with constructs of the DSL. The global requirements (D_G) are associated with the entire application. They can be directly embedded in the test model every time a domain specific application is specified and after it has been validated for ambiguity, correctness and consistency. For the feature specific requirements, the first step is to identify functions/ operators that realize such features in an implementation. Any such function may act on a set of parameters. The next and the most critical step is to associate the requirements to the function on the basis of the types of the parameters.

Types in a language capture the essential concepts of any domain and a domain specific requirement is essentially a constraint on the application's behavior expressed through the domain concepts. Any domain specific requirement can be

expressed by a tuple, $R =< D, C >$ where, D is the domain concept and C is the associated constraint. For instance consider the domain requirement, "Any probability value should lie between 0 and 1". It can be expressed in a tuple as $R =< probability, \geq 0, \leq 1 >$. Thus if there is a type *probability* in the DSL, we can associate the constraint $\geq 0, \leq 1$ with every instance of the type and whenever there is a call to a function that has a parameter of type *probability* or has an output of type *probability*, we can produce test cases for input or output validation.

A domain specific language should have the types embedded so that each domain concept is addressed and so that the types satisfy associated constraints. For example for the domain of graphical user interface we need to have an embedded type for buttons (since, 'button' is a domain concept). And also it is necessary that elements of type 'button' can be embedded into elements of type 'window' (since, the associated constraint with the concept of button is that they must be associated to some pre-defined window).

A function in a domain specific language, relates the domain specific concepts while satisfying the constraints imposed by the types. Thus if a function f is such that $Y = f(X)$ then we can say that it relates X and Y, where X and Y are lists of domain concepts. Each domain specific concept in the input and the output parameter list is of certain type satisfying the domain constraint. The static type checkers on validating the function f guarantee that the relation expressed by the function f is satisfying the domain constraints. This is what we call a type axiom. Any function validated by the static type checkers satisfies a type axiom and the axiom says that the relation between X and Y is correct. Since, the types of a DSL capture the domain

constraints; we know that X and Y in the specification satisfy the corresponding domain constraints. This information can be embedded in the test oracle to derive additional test cases.

A function may assert multiple type axioms expressed using its parameters. Thus if F is the set of functions provided by the DSL and X_i is the set of parameters for function f_i such that $f_i \in F$, then we express $P_i = \bigcup_j A(x_{ij})$ as the set of type specific properties expressed by f_i when $\bigcup_j x_{ij} = X_i$.

While associating the requirements we need to ensure that $\bigcup_i P_i \supseteq D_s$, $\forall f_i$, where D_s is the set of feature specific domain requirements. This is called the completeness condition. In a case when we cannot satisfy the completeness condition, we may need to embed additional types and functions in the DSL in order to extend its expressive power.

8.2 Extension to the domain of Graphical User Interface

In this section we demonstrate how we can use the principles mentioned in the previous section to extend the principles of domain specific testing to the domain of graphical user interfaces.

8.2.1 Domain Analysis

After an ad-hoc analysis, it was found that following are some of the domain specific requirements that can be associated with GUI applications.

Req. # 1. The items in the list of configuration should be of the right type, e.g. the name of the widget (Basic GUI element like a button, menu etc.) should be of type string, and the coordinates should be a tuple of integers and so on.

Req. # 2. The parent window for which a widget is being created must be predefined. This is necessary because this ensures that all the widgets are associated to certain windows. This is the requirement that says that a 'menu' or a 'button' must be an element of a 'window'.

Req. # 3. Each GUI action initiates a function of the underlying model. This is the requirement that ensures that there are no useless GUI elements and that the GUI is consistent.

Req. # 4. The view of the application is modified in accordance to the change in the underlying model. This requirement necessitates the adherence of the GUI to the model view architecture[1]. This is to ensure that the display reflects the system state.

Req. # 5. The display window should be smaller than the parent window.

The above list is ad-hoc and far from being exhaustive but will suffice to exemplify the steps of extension.

8.2.2 Choice of Domain Specific Language

Many higher ordered typed domain specific languages are available that bind with various GUI frameworks like Window, Gtk, Tcl, Tk, Delphi, AWT etc. The need is for a DSL that has embedded types to address domain concepts like widgets, events, window etc. Following is a list of higher ordered typed domain specific languages derived from Haskell that can be used to specify graphical user interfaces for various applications:

- *wxHaskell* : wxHaskell provides a set of library functions to directly interface with wxWindows - a comprehensive C++ library that is portable across all major GUI frameworks like GTK, Windows, X11, and MacOS X.
- *HToolkit* : HToolkit is a portable Haskell library for writing graphical user interfaces (GUI's). The library is built upon a low-level interface that will be implemented for each different target GUI frameworks. The low-level library is called Port and is currently implemented for GTK and Windows. The middle-level library is named GIO (the Graphical IO library) and is built upon the low-level Port library.
- *Gtk+HS*: Gtk+HS is a Haskell binding for Gtk. This library provides a transcription of the original Gtk API into Haskell. Gtk+HS has a modern, portable library and forms the basis of the Gnome desktop project. The binding, while not complete, covers most of Gtk's core functionality and is ready for use in applications that require a GUI of medium complexity.
- *Htk* : Htk is a library of functions for writing framework independent, graphical user interfaces in Haskell. The library provides a convenient, abstract and high-level way to write window-oriented applications. It also provides a lower level interface to write primitive Tcl/Tk code where helpful. It can be used for UNIX and Windows.
- *TclHaskell* : TclHaskell is a typed, portable encapsulation of Tcl into Haskell. Its distinctive features are the use of Haskell types and type classes for structuring the interface, an abstract notion of event for describing user interaction, and portability across Windows, Unix and Linux.

The above list is again not exhaustive but serves to exemplify the extension process. Among the DSLs listed above, Htk and TclHaskell have embedded types to capture GUI elements. All the remaining DSLs provide libraries to port Haskell functions to the imperative languages commonly used for GUI design and implementation. Since, the objective of our DSL is specification of a system instead of implementation, the selection criteria is governed by the DSL's ability to capture the domain specific properties. Htk and TclHaskell are natural qualifiers as they provide an extensible type library to capture domain concepts that would assist the test case derivation process. The net number of functions in TclHaskell is 239 while that in Htk is 782. So it is assumed that TclHaskell is more usable than Htk. (This may not be the case in reality)

8.2.3 Associating Requirements to DSL constructs

The types embedded in TclHaskell can be categorized into:

1. the types corresponding to GUI elements,
2. the types corresponding to GUI configuration items,
3. the types corresponding to GUI events,
4. the types corresponding to display configurations,
5. and the basic types in Haskell, viz., String, Num etc.

All functions in TclHaskell specifications have parameters and return values corresponding to any of these types. Upon type validation we know that the functions in the specification are type correct. Thus, any function from the validated specification is a source of oracle information. For instance whenever the operator "**`button`**" is used in TclHaskell, the type system guarantees that the button is

configured using objects of right type. The type of "**button**" is defined in Haskell notation as:

```
button::WPath→[Conf But]→GUI Button
```

which says that the operator **button** takes in two arguments:

1. First a variable of type **WPath**
2. and second, a list of variables of type **Conf But**.

While the type **WPath** forces the first argument to be only a valid "window" description, the type **[Conf But]** forces the second argument to be a set of valid configuration commands applicable to buttons. In other words, the type **Conf But** ensures that a *button* can be configured only through a set of allowed library functions whose return types are **Conf But**, eg., text. The function text defines the label of a button and is designed to have the following type:

```
text :: String → Conf But
```

Thus for the library function **button** the input data set is strictly characterized. This also means for the oracle that before the call to function button, the window must exist (Req #2) and the configuration list is appropriate (Req#1). Therefore, if we associate test cases for these two requirements with every call to the button function, then we can assert if Req #1 and Req #2 are satisfied by the application.

Typically a GUI specification in TclHaskell will contain the following four categories of functions:

1. Widget Creation Functions: These functions create various windows components like the buttons, menus and textboxes. These are the functions that relate the configuration items with GUI elements while creating them. Thus, these are the functions ideal for relating the requirement numbers 1 and

2. Any functionality creating a widget should be aware of the requirements 1 and 2. Therefore, any time there is a call to a widget creation function, we should have domain specific test cases testing for requirements 1 and 2.

2. Support Functions: These functions enhance the GUI display by naming the buttons, providing titles for windows etc. These are the functions that configure GUI items. Hence, they are also the candidates for requirement # 1. However, since there is no creation of a widget involved here, requirement #2 does not hold.
3. Event Binding Functions: These functions bind action to various GUI events like clicking of a button, clicking on a menu, typing of the keys, etc. Naturally they qualify for the requirement # 3 and 4.
4. Display Functions: These functions display a widget on the screen. They determine the display location, size, refresh rate etc. Therefore they can be bound with requirement # 5 on display

After relating these requirements to the functions, states can be embedded in the EFSM based structural representation as explained in chapter 4. Table 8.1 lists the various groups of functions and the associated requirements.

Type	Description	Axioms	Requirement
Widget Creation Functions:	These functions create various windows components like the buttons, menus and textboxes.	• The Configuration Items have the right type. • The parent window exists. • The widget has the right number of configuration parameters	Req # 1. Req # 2.
Support Functions:	These functions enhance the GUI display by naming the buttons, providing titles for windows etc.	• The name is a string. • The GUI element exists	Req # 1

Event Binding Functions:	These functions bind action to various GUI events like clicking of a button, clicking on a menu, typing of the keys, etc.	• The action corresponds to the type of GUI element. • Each action initiates one function	Req # 3. Req # 4.
Display Functions:	These functions display a widget on the screen. They determine the display location, size, refresh rate etc.	• The display configuration have right number of parameters • The display configuration has right type	Req # 5

Table 8.1: The function categories of TclHaskell and the associated requirements

A complete list of TclHaskell functions and their categories is provided in Appendix C.

8.3 Summary

In summary it can be said that to extend the test generation technique to different domains we need to first identify a set of domain requirements. These domain requirements can be classified further as requirements that are true for all domain specific applications and the requirements that are true for certain domain specific applications. The latter are called feature specific domain requirement. For each feature specific domain requirement we need identify a function in the DSL that implements the feature. The feature specific requirement is associated with that function and its parameters. In a case when the entire set of domain specific requirements cannot be associated with the embedded functions, then there is a need to extend the DSL. The DSL can be extended by embedding extra types and functions.

8.4 References

[1] Krasner and Pope, *A Cookbook Approach to Using MVC*, JOOP, 1(3): 26-49.

Chapter 9 Conclusion and Future Work

In this chapter we conclude by identifying the advantages of HOTTest over other model based test design techniques. We also identify the shortcomings and limitations of HOTTest. Avenues for future research are subsequently identified.

9.1 Advantages of HOTTest

HOTTest has the following advantages over other model based test design techniques:

1. Higher Effectiveness: In HOTTest test models are created from a natural language specification of a system using a domain specific language. This means that the tester needs to translate from one textual representation to another. The identification of the system states is automatic. This relieves the tester from the burden of abstracting the system and identifying the state transitions manually. Thus the modeling process has increased level of automation and is therefore, less error-prone. This also ensures a higher requirement coverage and consequently higher effectiveness.

   Further, HOTTest identifies some domain specific implicit requirements, for which test cases are automatically embedded in the test suite. This leads to an increase in overall and domain specific effectiveness. HOTTest is capable of identifying greater number of domain specific and generic defects.

   HOTTest's abilities have been experimentally validated against ASMLT, Archetest and EFSM based testing. (see chapter 7) The requirement coverage obtained by using HOTTest is on an average twice that of the other test generation techniques.

2. Efficient Test Generation: With increase in size of the application, the complexity of the modeling in model based testing increases. The extent of increase in the modeling complexity for HOTTest is minimal as compared to Archetest and EFSM based testing. This is again because of the text based representation, adopted by HOTTest. Subsequently the modeling time for large applications is the least when compared to other model based test generation techniques. Similar reduction in modeling time is observed in another text based modeling technique ASML and therefore, confirms our assumption. Reduction in modeling directly manifests into reduction in net effort for testing because in model based testing the primary contributor to the effort is the modeling effort. Thus for larger applications we need comparatively less effort in HOTTest. This implies a direct gain in efficiency.
3. Comparable Usability: For a person who is usually conversant with procedural languages, the learning of a functional language is not easy. This implies a slow learning process, but once the learning is complete, modeling in HOTTest becomes easier. The modeling process is less error prone and is more efficient. Being declarative in nature the modeling language is easy to use and lends itself easily for writing specifications. Usability measured through Learnability, Efficiency, Error Proneness, Satisfaction and Ease is hence comparable to other test generation tools. (See chapter 6)
4. High Scalability: As is discussed in Chapter 7, scalability of HOTTest to industrial scale problems is very high. Using HOTTest, it was possible to model an industrial application (>50kloc) and to generate test cases for it. It

scales up easily on accounts of test modeling time, test generation time and test set up time. It does not scale up similarly while considering the execution time.

5. Lends itself to model checking: The structural representation used in HOTTest is a finite state machine based representation with embedded semantics in terms of actions and constraints. This kind of representation lends itself easily to finite state machine based model checking tools for validation of systems. The system can be checked for safety and live-ness properties and vulnerable configurations could be detected.

9.2 Limitations of HOTTest

HOTTest at its present state has the following limitations:

1. Separate DSL for every domain: HOTTest depends on type axioms offered by domain specific languages in order to capture implicit domain specific requirements. Although explicit requirements for applications can be tested by HOTTest for any application domain, the implicit requirements can only be addressed if an appropriate DSL is available. This effectively means that in order to test for domain specific requirements, we need separate domain specific language for each of the domains. Also, before use, domain experts need to define potential domain specific requirements and such requirements need to be expressed through domain specific constructs. This is an overhead that reduces the extendibility of HOTTest.
2. Training in every DSL: Also, since we have a separate DSL for every application domain, therefore before a tester can start availing domain specific

test generation capabilities of HOTTest he needs to learn the corresponding DSL. This process however, becomes easier with each new DSL that is learnt. This is because the DSLs used in HOTTest are all embedded in Haskell, and they share the same grammar and therefore have similar constructs.

3. Inability to capture every requirement: HOTTest cannot ensure that the implicit domain specific requirements covered by the test suite form the exhaustive set of such requirements. The effectiveness of HOTTest is largely dependent on the prior domain research conducted and the number of domain specific requirements identified during such research. If the research is incomplete the test generation effectiveness will suffer accordingly.
4. Assumption of correct requirements: Like any other model based testing, HOTTest assumes that the requirement specification for the application is correct. Some of the requirement faults can be uncovered through static analysis of the test model but there is no effective way to ensure that the system specification is thorough, consistent and safe. Model checking tools, however, can validate a system for such properties starting from the EFSM that is generated as part of the structural representation of the system in HOTTest.
5. Redundant Test Cases: HOTTest at its present state generates a number of redundant test cases. Since the axiom embedding is done in a non-conditional manner, the test model verifies for the same property every time a similar functionality is called. For example if a certain table is missing in the database, it is understood that the fields of the table will also be missing from

the database. At this point, it is not possible in HOTTest to identify such dependencies. As a result, we have test cases for checking failures which are in reality dependent on each other. This results in an excessive number of redundant test cases. These test cases increase the net execution time for HOTTest and add up to the net testing effort.

9.3 Future Research

Following are some possible avenues for future research:

1. State Space Reduction: The state space of the HOTTest test model can be reduced by eliminating dependent failure states [] or by identifying spatial or semantic symmetry in the system []. Such a reduction will reduce the number of test cases produced by HOTTest and will thus increase the efficiency of HOTTest. This will also make HOTTest more scalable and applicable to large scale industrial applications.
2. Generalized Axiom Embedding Process: At present the domain specific requirements are associated manually by identifying the type axioms offered by the functions. This is a process that accounts for a major overhead for HOTTest for extending it to other domains of interest. It might be possible to develop a generalized theory for type embedding and extension to other domains. Such a theory will enhance HOTTest's applicability to other domains.
3. Development of an integrated testing environment: The prototype tool for HOTTest is made up of three independent components. The HaskellDB modeler, the EFSM generator and a test case generator. One of these

components is a commercial tool and another is an open source tool. Through future research an integrated test generation environment can be developed that will provide the testers with an environment where they can specify the system, generate the test cases and execute them without having to depend on multiple applications.

Appendix A: HaskellDB

HaskellDB [5] focuses mainly on the type-safe construction of SQL database queries [2, 6]. This appendix introduces the types and operations that HaskellDB provides, and attempts to give an informal semantics for HaskellDB queries. For a detailed understanding on HaskellDB the reader is referred to [5]

HaskellDB Types

Types play a crucial role in HaskellDB. The relational database concepts are expressed through Haskell records available through TREX extensions. Additional types are embedded to define attributes, relations and tables of the database. This section introduces the main HaskellDB specific types, and explains what they're modeling.

`Rel`: the Relation Type

A relation groups together a collection of attributed values. It is represented by the abstract type `Rel`:

```
data Rel r -- abstract
```

It is parameterized over a type that precisely captures the relation — that is, what attribute-value pairs the relation brings together. For instance, if the relation is intended to describe a phone book, it will at the very least contain the names and phone numbers of the subscribers. Its `Rel` type would then be:

```
phBook :: Rel (name :: Expr String,number :: Expr String)
```

The type is stating that for each record/row in the database, it will contain an entry for the `name` and `number` — and those two only. Associated with both the `name` and `number` attributes are String values.

In other words, the type argument to `Rel` precisely captures the attributes of the relation and the type of the attribute values. Notice that the `Rel` type makes use of TREX records, an extension to the Haskell type system. [3, 4][11]

The benefits of using types to precisely describe a relation will become clearer once we present the combinators for working with relations.

`Table`: Database Tables

Relational databases represent relations via tables, and HaskellDB defines a table type that is parameterized over the type of the relation:

```
data Table rel -- abstract
```

The type argument is the same as the type used for `Rel`. In the case of the phone book the table would have the type

```
phBookTable :: Table (name :: Expr String, number :: Expr String)
```

`Attr`: Attributes

A relation associates a collection of attributed values. In HaskellDB, attributes are first-class values, all of which have the `Attr` type:

```
data Attr rel value -- abstract
```

It is parameterized over two type arguments, the second of which is the type of the value the attribute contains or denotes (its "domain", to use relational database terminology). The first type parameter constrains (or specifies, depending upon how one looks at it) what kinds of *relations* the attribute can be associated with. Consider the phone book example. It had two attributes, which will have the following types:

```
attrName :: (r\name) => Attr (name :: Expr String | r) String
attrNumber :: (r\number) => Attr (number :: Expr String | r) String
```

[11] Unfortunately, only the Hugs interpreter has an implementation of TREX. This in turn means that HaskellDB is confined to Hugs for the near future.

These types use TREX also. The type for `attrNumber` is saying:

Given a relation that has an attribute with name `number` *and an associated value of type* `String` *(and no other attributes named* `number`*), then* `attrNumber` *is an attribute of this relation.*

`phBook` does indeed satisfy those constraints, so `attrNumber` does represent its `number` attribute. Similarly for `attrName`.

A word about names and name spaces in HaskellDB: In the above example, `name` and `number` were used as field labels in the TREX types. These field labels have a separate name space from that of other Haskell values, so it is legal to also use the field labels as the names for the attributes, *i.e.*,

```
name :: (r\name) => Attr (name :: Expr String | r) String
number :: (r\number) => Attr (number :: Expr String | r) String
```

We will do this from now on, but chose not to initially to avoid confusing the name of an attribute in a relation from that of the name which represents that attribute in HaskellDB.

`Expr`: Typed SQL Expressions

To build up interesting queries with HaskellDB, we want to construct expressions that can be converted into the query language of the dB system we're talking to. In the case of HaskellDB, the query language is SQL, so we want to have a way of constructing well-formed SQL expressions. To cut a long story short, HaskellDB uses the following type to represent SQL expressions:

```
data Expr t -- abstract
```

`Expr` is essentially an *abstract syntax tree* representation of possible SQL expressions; it is a datatype whose values correspond directly to SQL expressions.

The role of the type parameter is analogous to that played by types in most programming languages: it prevents us from constructing `Expr` values that correspond to ill-formed SQL expressions. See [5] for details as to exactly why this representation is the preferred one, and more explanation on how the type parameter to `Expr` guarantees type-correctness of the corresponding SQL expression.

Aside: *The original paper [5] and the implementation differ in one aspect:* `Attrs` *use* `Expr t` *for their value types in the implementation, as opposed to just* `t` *in the paper. The reason for this mismatch is not clear, but it does mean that one may only store valid SQL expression values in HaskellDB attributes, a constraint that makes good design and programming sense.*

Summary

The world of relational databases uses different terms for the same object, depending upon how that object is being used or viewed. The following table lists the most common synonyms:

Concept	**Synonyms**
Database	table
Relations	rows
Fields	attributes, columns

Table A1.1: Synonyms of Database Concepts

The first column lists the more abstract concepts, while the second column lists common ways these concepts are implemented. Attributes fall somewhere in between. One way to relate the HaskellDB types is to the following:

- a database or table, of type `Table r`, consists of a collection of

- rows or relations, of type `Rel r`, where each
- column or attribute or field, of type `Attr (Expr t)`, is named and contains a value, of type `Expr t`.

Some confusion also arises due to the use of *relation* to occasionally mean a set or list of rows.

Representing Queries

HaskellDB provides the user with a monad to build up a query or relational expression: the `Query` monad. It provides the following basic operations:

```
data Query a -- abstract
returnQ :: a -> Query a
bindQ :: Query a -> (a -> Query b) -> Query b
table :: Table r -> Query (Rel r)
restrict :: Expr Bool -> Query ()
project :: r -> Query (Rel r)
```

By using a monad, HaskellDB code can then be phrased using Haskell's overloaded notation for monads (the "do" notation). Here's an example query:

```
-- project out all the names from the phone book.
names = do
   ph <- table phBookTable
   project ( ph ! name )
```

HaskellDB combinators/operators

With the relevant types all introduced, let's have a look at the HaskellDB combinators and operators one can use to construct queries. A query works over rows or relations, so we first need to be able to select the tables to draw the rows from.

Throughout, we will use the following databases as examples. The first two are simple author/publication databases, called *home* and *away*, respectively:

home:

Name	Author ID	Title
Carol	1	“Integrating software into PRA”
Ming	2	“Reliability Prediction Systems for Software”
Avik	3	“Stateful Transformations in Functional Language”

away:

Name	Author ID	Title
Carol	1	“Integrating software into PRA”
Sush	2	“Cooking Sushi?”
Anand	3	“Visiting places on demand!”
Sachin	4	“Reliability and Availability”
Avik	5	“Learning HaskellDB: My Journey to Hell and Back”

The next two examples tables are a list of customers and their orders (called `orders`), and a separate database of customers and their phone numbers (called `phonebook`):

orders:

Name	Cost
Carol	10
Carol	500
Sush	10
Avik	3000
Avik	30

Sush	200

phonebook:

Name	**Number**
Anand	555-1020
Avik	555-9943
Sush	555-2134

Row Selection

You select a row using the `table` operator:

```
table :: Table r -> Query (Rel r)
```

This says that given a database or table consisting of rows of type `r`, this will return a query whose result is a row of type `r`.

For example, the following code fragment will select the Home database:

```
h <- table home
```

Now we can access any of the fields of the database, using `!`, as we shall see below, by referring to the database with the variable `h`. In fact, any operation we wish to perform on the database `home` will use the variable `h` to refer to the database. More accurately, `h` refers to any row of the database `home`.

Attribute/Column Selection

Given a row, you can select a particular column / attribute from it using the `(!)` operator:

```
(!) :: Rel r -> Attr r a -> Expr a
```

Notice how the `r` type variable ensures that we're only able to select attributes/columns that the relation actually contains.

For example,

```
h ! title
```

will extract the value of the field named `title` in the row referred to by `h`. However, `h ! cost`

will generate an error like:

```
ERROR HaskellDBExample.hs:113 - Type error in application
*** Expression : h ! cost
*** Term : h
*** Type : Rel (name :: Expr String, authorId :: Expr Int, title ::
Expr String)
*** Does not match : Rel (cost :: Expr Int | b)
*** Because : field mismatch
```

This means that `h` does not a field called `cost`, and is how record field mismatch errors appear in TREX. In this way, HaskellDB guarantees that any field accesses are valid (that is, the database being queried really does have the corresponding field).

Projection

At the query-level, you can project a new relation using the `project` operator:

```
project :: r -> Query (Rel r)
```

But, project from what? From an expression constructed using the `(!)` operator, most likely. To make this clearer, here is a query which selects all the names and titles in the `Away` database:

```
names :: Query (Rel (name :: Expr String,title :: Expr String))
names = do
   entry <- table away
   project ( who = entry ! name, what = entry ! title )
```

This can be read as:

For each row in the table `Home`*, construct a new row that contains a field named* `who`*, and one named* `what`*.*

The database so constructed is:

Who	**What**
Avik	"Learning HaskellDB: My Journey to Hell and Back"
Sush	"Cooking Sushi?"
Anand	"Visiting places on demand!"
Sachin	"Reliability and Availability"
Yuan	"Two Thumbs Up!"

Restriction

What if you wanted to constrain the row names returned to only those written by Anand?

You'd use the `restrict` operator:

```
restrict :: Expr Bool -> Query ()
```

For example,

```
ming :: Query (Rel (name :: Expr String, authorId :: Expr Int, title
:: Expr String)))
ming = do
      entry <- table away
restrict ( entry ! name .==. constant "Anand" )
```

This results in the following table:

Name	**Author ID**	**Title**

Anand	3	“Visiting places on demand!”

Join

What about the join operator, how do we express it in HaskellDB? Through `restriction` and the `(!)` operator:

```
names :: Query (Rel (name :: Expr String, phone :: Expr String))
names = do
        customer <- table orders
        phBook <- table phonebook
        restrict ( phBook ! name .==. customer ! name)
        project ( who = phBook ! name, what = phBook ! number )
```

In other words, one relates two relations with `restrict` expressions that compare columns from different tables (which correspond to the relations).

The above query yields this table:

who	**what**
Avik	555-9943
Sush	555-2134

Carol doesn’t appear, since his phone number is not in the `phonebook` table; Anand doesn’t appear since he has no orders in the `orders` table.

Basic Set Operations

HaskellDB provides you with the basic operations of an relational algebra:

```
union :: Query (Rel r) -> Query (Rel r) -> Query (Rel r)
minus :: Query (Rel r) -> Query (Rel r) -> Query (Rel r)
intersect :: Query (Rel r) -> Query (Rel r) -> Query (Rel r)
```

The Cartesian product of two relations is provided via join; see Section 3.5. Each of the above set operations may only be applied to relations that have precisely the same set of attributes. Note that this means that the types of the values stored in the

attributes must be the same; simply having the same field name is not sufficient. This condition is guaranteed by the types of the operators.

For example, the following code:

```
table home `union` table away
```

yields this table:

Name	**Author ID**	**Title**
Carol	1	"Integrating software into PRA"
Sush	2	"Cooking Sushi?"
Anand	3	"Visiting places on demand!"
Sachin	4	"Reliability and Availability"
Avik	5	"Learning HaskellDB: My Journey to Hell and Back"
Ming	2	"Reliability Prediction Systems for Software"
Avik	3	"Stateful Transformations in Functional Language"

Note that Avik now has two different `authorIds`, and that the duplicate entry for Carol has been removed.

But the following code:

```
table home `union` table orders
```

will result in a "field mismatch" error, since the two tables have different fields. This is sometimes known as failing *union-compatibility*, but there is a similar constraint on the other set operators, so we will use the term *field-compatibility*.

Set operator `minus` performs the set difference operation on the given tables (returning only those rows from the first argument which do not occur in the second), provided they are field-compatible:

```
table away `minus` table home
```

yields:

Name	Author ID	Title
Sush	2	"Cooking Sushi?"
Anand	3	"Visiting places on demand!"
Sachin	4	"Reliability and Availability"
Avik	5	"Learning HaskellDB: My Journey to Hell and Back"

Finally, `intersect` performs set intersection between the tables in question, provided they are field-compatible:

```
table home `intersect` table away
```

yields this table:

Name	Author ID	Title
Carol	1	"Integrating software into PRA"

Building Expressions

Using the parameterized `Expr` type and its operators, you can construct more interesting `restrict` expressions—here's a complete list of the SQL operators supported by HaskellDB.

Comparison operators

We've already seen some examples of the use of one comparison operator, equality, written `.==.`, in the `restrict` examples above. Here's the complete list:

```
(.==.) :: Eq a => Expr a -> Expr a -> Expr Bool -- Equality
(.<>.) :: Eq a => Expr a -> Expr a -> Expr Bool -- Inequality
(.<.) :: Ord a => Expr a -> Expr a -> Expr Bool -- Less than
(.<=.) :: Ord a => Expr a -> Expr a -> Expr Bool -- Less than or
equal
```

```
(.>.) :: Ord a => Expr a -> Expr a -> Expr Bool -- Greater than
(.>=.) :: Ord a => Expr a -> Expr a -> Expr Bool -- Greater than or
equal
```

These, like all of the boolean expression forms in HaskellDB, are really only useful to the `restrict` operator, and tend to be used in conjunction with the boolean operators below. Note the naming convention: take the normal Haskell operator name and put periods on either end. This convention is used for most of the primitive operators. It indicates that the operator in question operates upon `Expr` types (*i.e.*, Haskell expressions that *represent* SQL expressions).

Boolean operators

The following boolean operators may be used to construct more complex restriction expressions:

```
_not :: Expr Bool -> Expr Bool -> Expr Bool -- Negation
(.&&.) :: Expr Bool -> Expr Bool -> Expr Bool -- Conjunction
(.||.) :: Expr Bool -> Expr Bool -> Expr Bool -- Disjunction
```

Here's a (quite contrived) example of their use:

```
x <- table away
restrict (x ! name .==. constant "Sush" .||._not (x ! authorId .==.
3))
```

This yields the following table:

Name	**Author ID**	**Title**
Carol	1	"Integrating software into PRA"
Sush	2	"Cooking Sushi?"
Sachin	4	"Reliability and Availability"
Avik	5	"Learning HaskellDB: My Journey to Hell and Back"

Arithmetic operators

When building new tables with `project`, we often want to perform some calculation:

```
(.+.) :: Num a => Expr a -> Expr a -> Expr a -- Addition
(.-.) :: Num a => Expr a -> Expr a -> Expr a -- Subtraction
(.*.) :: Num a => Expr a -> Expr a -> Expr a -- Multiplication
(./.) :: Num a => Expr a -> Expr a -> Expr a -- Division
(.%.) :: Num a => Expr a -> Expr a -> Expr a -- Mod
```

These operators are also used in conjunction with so-called *aggregate* operators. Here's an example which adds the various `authorId`s from tables `home` and `away`:

```
x <- table home
y <- table away
project (name = "Author id sum",sum = _sum x authorId .+._sum y authorId)
```

This results in the following table:

name	**Sum**
Author id sum	21

We'll see more examples below.

String operators

The following string operations are available:

```
(.++.) :: Expr String -> Expr String -> Expr String -- Concatenate
cat :: Expr String -> Expr String -> Expr String -- Concatenate
constant :: ShowConstant a => a -> Expr (Maybe a) -- Values into strings
```

Imagine a database called `names`, with two fields: `firstName` and `surname`. Then we

can build a table of `fullNames` thus:

```
x <- table names
project (fullName = x ! firstName .++.x ! surName)
```

Suppose `names` looked like this:

firstName	**surName**
George	Bush
Clay	Williams
Marv	Zelkowitz

Then the above results in the following table:

fullName
George Bush
Clay Williams
Marv Zelkowitz

`constant` may be used (along with `.++.`) to build display or description strings, or strings for pattern matching (see Section 3.7.5). Any normal Haskell datatype that has a `Show` instance (*i.e.*, any type that can be printed) may be used.

Pattern Matching

One can use `like` to perform simple pattern matching on string-value fields:

```
like :: Expr String -> Expr String -> Expr Bool
```

There are two special wildcard characters: "%" and " ". The first will match any string, while the second will match any character. These characters may be escaped, using the escape character "\".[12]

So to search for titles in containing a given string, one would use:

```
z <- table titles
restrict ( like (constant ("%" ++ t ++ "%")) z ! title )
```

Aggregate operators

An aggregate operator acts upon all of the values of a given field of a table at once. Here is a list of the supported aggregate operators:

```
count :: Rel r -> Attr r a -> Expr Int -- number of values
_sum :: Num a => Rel r -> Attr r a -> Expr a -- sum of all values
_max :: Num a => Rel r -> Attr r a -> Expr a -- maximum
_min :: Num a => Rel r -> Attr r a -> Expr a -- minimum
avg :: Num a => Rel r -> Attr r a -> Expr a -- average
stddev :: Num a => Rel r -> Attr r a -> Expr a -- std deviation
(sampled)
stddevP :: Num a => Rel r -> Attr r a -> Expr a -- std deviation
variance :: Num a => Rel r -> Attr r a -> Expr a -- variance
(sampled)
varianceP :: Num a => Rel r -> Attr r a -> Expr a -- variance
```

Consider the `Orders` table, and suppose we want to calculate the number of orders over $50, and average them. This code will suffice:

```
x <- table Orders
restrict ( x ! Cost .>=. 50 )
project ( average = _sum x Cost ./. count x Name )
```

However, we could also just write:

[12]The precise characters used varies with the particular SQL server that your HaskellDB program is interacting with. These values work with the Microsoft SQL servers.

```
x <- table Orders
restrict ( x ! Cost .>=. 50 )
project ( average = avg x Cost )
```

The difference between `stddev` and `stddevP` is that the former calculates the standard deviation from a *sample* of the values of the given field, where as the latter bases its calculation upon *all* values of the given field. (The difference between `variance` and `varianceP` is analogous.) The sample size chosen is dependent upon the SQL server.

Sorting responses

To present the rows satisfying a query in a given order, one can use the `order` function:

```
asc :: Rel r -> Attr r a -> Expr Order
desc :: Rel r -> Attr r a -> Expr Order
order :: [Expr Order] -> Query ()
```

`order` takes a list of so-called `Order` expressions. `asc` and `desc` are used to influence the "direction" of the sort. Each takes a row and a field upon which to sort. `Order` sorts in a lexicographic fashion (first by the first element of its argument list, then by the second if necessary, *etc.*). The following code sorts the orders order of their cost (largest first), with orders of the same cost being sorted alphabetically:

```
x <- table orders
order [ desc x cost, asc x name ]
return x
```

This results in the following table:

Name	**Cost**
Avik	3000
Carol	500
Sush	200
Avik	30

Carol	10
Sush	10

In SQL, one doesn't need to specify whether the sort should be ascending or descending; if no order is specified, then the default is ascending. However, HaskellDB requires the specification of the sorting direction.

Filtering responses

It is often useful to cut down on the number of responses returned. For this, one can use `top` or `topPercent`:

```
top :: Integer -> Query ()
topPercent :: Integer -> Query ()
```

Both take an integer argument. `top n` will discard all but the first `n` responses; `topPercent n` will discard all but the first `n%` responses. `top` and `topPercent` are often used in conjunction with `order`.

The following takes the query from the above example, but will only return the first two results:

```
x <- table orders
restrict ( x ! cost .>=. 500 )
order [ desc x cost, asc x cost ]
top 2
return x
```

`NULL` – Missing Data in HaskellDB

SQL's `NULL` is a complex beast. SQL most commonly uses `NULL` for unknown values, for values to be filled in later, for optional values, and to represent the result

of nonsensical calculations (*e.g.*, division by zero), but this list is by no means complete.[13] 3

When dealing with fields whose values may be `NULL`, then the following operations will be essential:

```
isNull :: Expr a -> Expr Bool -- True if NULL
notNull :: Expr a -> Expr Bool -- True if not NULL
nullable :: ShowConstant a => a -> Expr (Maybe a) -- Similar to constant
```

`nullable` is essentially the same as `constant`, except that `NULL` values should be rendered as the string "`NULL`".

Comments

There are many SQL operators not supported by HaskellDB (such as more complex aggregate boolean operators). This is due to the fact that it arose out of a research project, whose main aim was investigation of embedded domain-specific languages, rather than constructing a fully-functional interface to SQL databases.

Another shortcoming is that there is no facility for easily adding user-defined operations to HaskellDB. This is impossible to get around, since HaskellDB doesn't perform any expression evaluation itself; the SQL database server is responsible for that. If HaskellDB were to pass on to it some user-defined operation, then the SQL server would not know how to proceed. The only alternative is to implement the operation in terms of those SQL operations provided by HaskellDB, augmented with Haskell expressions.

[13] Chapter 6 of [1] has a deeper discussion on the use and abuse of `NULL` in SQL databases.

References

[1] J. Celko. *Joe Celko's SQL for Smarties: Advanced SQL Programming*. Morgan Kaufman, second edition, 2000.

[2] C. J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, 1993.

[3] B. R. Gaster. Polymorphic extensible records for Haskell. In *Haskell Workshop*, Amsterdam, June 1997. ACM.

[4] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical report NOTTCS-TR-96-3, Languages and Programming Group, Department of Computer Science, University of Nottingham, Nottingham NG7 2RD, UK, Nov. 1996.

[5] D. Leijen and E. Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain-Specific Languages (DSL)*, Austin, Texas, Oct. 1999. Available for download at http://www.cs.ruu.nl/people/daan/papers/dsec.ps.

[6] Microsoft. *DevGuru Jet SQL Developer Index*. http://www.devguru.com/ Technologies/jetsql/quickref/jet sql list.html.

Appendix B: HaskellDB Axioms

Operator	Type	Usage	Axioms
-- query Operators			
(!)	:: Record r => Rel r -> Attr r b -> Expr b	table ! field	Field Exists in the table
project	:: Record r => r -> Query (Rel r)	project(column = attribute,...)	No two fields in the relation are same
restrict	:: Expr Bool -> Query ()	restrict(Boolean Expression)	The boolean expression is valid
table	:: Table r -> Query (Rel r)	table TableName	The table exists in the database.
-- binary query operators			
union	:: Query (Rel a) -> Query (Rel a) -> Query (Rel a)	union(query1 query2)	The two relations have the same fields(type and names)
intersect	:: Query (Rel a) -> Query (Rel a) -> Query (Rel a)	intersect(query1 query2)	The two relations have the same fields(type and names)
divide	:: Query (Rel a) -> Query (Rel a) -> Query (Rel a)	divide(query1 query2)	The two relations have the same fields(type and names)
minus	:: Query (Rel a) -> Query (Rel a) -> Query (Rel a)	minus(query1 query2)	The two relations have the same fields(type and names)
-- operators over expressions embedded inside queries			
(.==.)	:: Eq a => Expr a -> Expr a -> Expr Bool	attribute1 .==. attribute2	The expression on either sides of the operator attributes are of same type and are equalable.
(.<>.)	:: Eq a => Expr a -> Expr a -> Expr Bool	attribute1 .<>. attribute3	The expression on either sides of the operator attributes are of same type and are equalable.
(.<.)	:: Ord a => Expr a -> Expr a -> Expr Bool	attribute1 .<. attribute4	The expression on either sides of the operator are of same type and are ordinal.
(.<=.)	:: Ord a => Expr a -> Expr a -> Expr Bool	attribute1 .<=. attribute5	The expression on either sides of the operator are of same type and are ordinal.
(.>=.)	:: Ord a => Expr a -> Expr a -> Expr Bool	attribute1 .>=. attribute6	The expression on either sides of the operator are of same type and are ordinal.
(.>.)	:: Ord a => Expr a -> Expr a -> Expr Bool	attribute1 .>. attribute7	The expression on either sides of the operator are of same type and are ordinal.
(.&&.)	:: Expr Bool -> Expr Bool -> Expr Bool	boolean expr1 .&&. boolean expr2	The expressions on either side of the operator must be a boolean expression
(.||.)	:: Expr Bool -> Expr Bool -> Expr Bool	boolean expr1 .||. boolean expr3	The expressions on either side of the operator must be a boolean expression
(.*.)	:: Num a => Expr a -> Expr a -> Expr a	attribute1 .*. attribute2	The expressions on either side of the operator must be of the same type and are of the Number class.
(./.)	:: Num a => Expr a -> Expr a -> Expr a	attribute1 ./. attribute3	The expressions on either side of the operator must be of the same type and are of the Number class.
(.%.)	:: Num a => Expr a -> Expr a -> Expr a	attribute1 .%. attribute4	The expressions on either side of the operator must be of the same type and are of the Number class.
(.+.)	:: Num a => Expr a -> Expr a ->	attribute1 .+. attribute5	The expressions on either side of the operator must be of the same type and are of the Number class.

	Expr a		
(.-.)	:: Num a => Expr a -> Expr a -> Expr a	attribute1 .-. attribute6	The expressions on either side of the operator must be of the same type and are of the Number class.
(.++.)	:: Num a => Expr a -> Expr a -> Expr a	attribute1 .++. attribute7	The expressions on either side of the operator must be of the same type and are of the Number class.
_not	:: Expr Bool -> Expr Bool	_not boolean expr1	The expression on which the operator acts must be boolean.
like	:: Expr String -> Expr String -> Expr Bool	like(attribute1 attribute2)	The expressions on which the operator acts must be of type String.
cat	:: Expr String -> Expr String -> Expr String	cat (attribute1 attribute2)	The expressions on which the operator acts must be of type String.
isNull	:: Expr a -> Expr Bool	isNull(attribute1)	NONE
notNull	:: Expr a -> Expr Bool	notNull(attribute1)	NONE
constant	:: ShowConstant a => a -> Expr a	constant X	NONE
count	:: Rel r -> Attr r a -> Expr Int	count(table, attribute)	The attribute must be an element of the field.
_sum	:: (Num a) => Rel r -> Attr r a -> Expr a	_sum(table, attribute)	The attribute must be an element of the field and be a number.
_max	:: (Num a) => Rel r -> Attr r a -> Expr a	_max(table, attribute)	The attribute must be an element of the field and be a number.
_min	:: (Num a) => Rel r -> Attr r a -> Expr a	_min(table, attribute)	The attribute must be an element of the field and be a number.
avg	:: (Num a) => Rel r -> Attr r a -> Expr a	avg(table, attribute)	The attribute must be an element of the field and be a number.
stddev	:: (Num a) => Rel r -> Attr r a -> Expr a	stddev(table, attribute)	The attribute must be an element of the field and be a number.
stddevP	:: (Num a) => Rel r -> Attr r a -> Expr a	stddevP(table, attribute)	The attribute must be an element of the field and be a number.
variance	:: (Num a) => Rel r -> Attr r a -> Expr a	variance(table, attribute)	The attribute must be an element of the field and be a number.
varianceP	:: (Num a) => Rel r -> Attr r a -> Expr a	varianceP(table, attribute)	The attribute must be an element of the field and be a number.
-- ascending, descending			
asc	:: Rel r -> Attr r a -> Expr Order	asc(table, attribute)	The attribute must be an element of the field.
desc	:: Rel r -> Attr r a -> Expr Order	desc(table, attribute)	The attribute must be an element of the field.
order	:: [Expr Order] -> Query ()	order(asc/desc table attribute)	The expression must express order.
top	:: Integer -> Query ()	top(integer)	The expression must have integer as an input.
topPercent	:: Integer -> Query ()	topPercent(integer)	The expression must have integer as an input.
DbOptions(dbUserID :: String, dbPassword ::String, dbHost::String, dbName::String)			
dbOptions	:: DbOptions (the default/empty value).	dbOptions	The dbUserID,dbPassword,dbHost and dbName must be String.
(!.)	:: Row a b => a c	row!.attribute	The row must exist and it must contain the attribute.

	-> Attr c b -> b		
query	:: Database db row -> Table r-> Query (Rel r) -> IO ()	query(db table query1)	The db must exist, table must exist, the query must be valid.
lazyQuery	:: Database db row -> Table r-> Query (Rel r) -> IO ()	lazyQuery(db table query)	The db must exist, table must exist, the query must be valid.
strictQuery	:: Database db row -> Table r-> Query (Rel r) -> IO ()	strictQuery(db table query)	The db must exist, table must exist, the query must be valid.
insert	:: Database db row -> Table r-> Query (Rel r) -> IO ()	insert(db table query)	The db must exist, table must exist, the query must be valid.
delete	:: Database db row -> Table r-> (Rel r -> Expr Bool) -> IO ()	delete(table (function))	The db must exist, table must exist, the boolean expression must be valid.
update	:: (Record r, Record s) => Database db row -> Table r-> (Rel r -> Expr Bool)-> (Rel r -> s)-> IO ()	update(db table function1 function2)	The db must exist, the table must exist,the condition must be a valid boolean condition,The record must be updated accordingly.
insertNew	:: (Record r) => Database db row -> Table r -> r -> IO ()	insertNew(db table row)	The db must exist, the table must exist, the record must be a valid one.
showQ	:: Query (Rel a) -> Doc	showQ(query)	The query must be valid
showOpt	:: Query (Rel a) -> Doc	showOpt(query)	The query must be valid
showSql	:: Query (Rel a) -> Doc	showSql(query)	The query must be valid

Appendix C: TclHaskell Axioms

Type	Description	Axioms			Category
Widget Creation Functions:	These functions create various windows components like the buttons, menus and textboxes.	TheConfiguration Items have the right type	The parent window exist	The widget has the right number of cofiguration parameters	**A**
Support Functions:	These functions enhance the GUI display by naming the buttons, providing titles for windows etc.	The name is a string	the GUI element exists		**B**
Event Binding Functions:	These functions bind action to various GUI events like clicking of a button, clicking on a menu, typing of the keys, etc.	action correspond to the type of GUI element	each action intiates one function		**C**
Display Functions:	These functions display a widget on the screen. They determine the display location, size, refresh rate etc.	diplay configuration have right number of parameters	display configurationn has right type		**D**

Function	Type	Category
Button	`button :: Window -> [Conf But] -> GUI Button`	A
Window	`window :: [Conf Top] -> GUI Window`	A
Frame	`frame :: Window -> [Conf Fra] -> GUI Frame`	A
mkChildOf	`mkChildOf :: Widget c w -> GUI WPath`	A
mkSibling	`mkSibling :: Widget c w -> GUI WPath`	A
addFinaliserW	`addFinaliserW :: Widget a b -> GUI () -> GUI ()`	A
rootWin	`rootWin :: GUI Window`	A
genWindow	`genWindow :: [Conf Win] -> GUI Window`	A
menu'	`menu' :: WPath -> [Conf Men] -> GUI Menu`	A
Menu	`menu :: Has_use_menu w => Widget a w -> [Conf Men] -> GUI Menu`	A
Popup	`popup :: Menu -> (Int,Int) -> GUI ()`	A
Tearoff	`tearoff :: Bool -> Conf Men`	A
Mbutton	`mbutton :: Menu -> [Conf MBut] -> GUI MButton`	A
mbutton'	`mbutton' :: Menu -> [Conf MBut] -> GUI MButton`	A
mradioButton	`mradiobutton :: Menu -> [Conf MRB] ->`	A

	`GUI MRadiobutton`	
mradioButton'	`mradiobutton :: Menu -> [Conf MRB] -> Int -> GUI MRadiobutton`	A
mcheckButton	`mcheckbutton :: Menu -> [Conf MChe] -> Int -> GUI MCheckbutton`	A
Separator	`separator :: Menu -> GUI Separator`	A
separator'	`separator' :: Menu -> Int -> GUI Separator`	A
frame'	`frame' :: WPath -> [Conf Fra] -> GUI Frame`	A
Frame	`frame :: Window -> [Conf Fra] -> GUI Frame`	A
inFrame	`inFrame :: Frame -> PackInfo`	A
inWindow	`inWindow :: Frame -> PackInfo`	A
label'	`label' :: WPath -> [Conf Lab] -> GUI Label`	A
Label	`label :: Window -> [Conf Lab] -> GUI Label`	A
button'	`button' :: WPath -> [Conf But] -> GUI Button`	A
radioButton'	`radiobutton' :: WPath -> [Conf RB] -> GUI Radiobutton`	A
radioButton	`radiobutton :: Window -> [Conf RB] -> GUI Radiobutton`	A
Radio	`radio :: [Radiobutton] -> GUI Radio`	A
Mradio	`mradio :: [MRadiobutton] -> GUI Radio`	A
Coval	`coval' :: Canvas -> CCoord -> CCoord -> [Conf COva] -> GUI COval`	A
coval'	`coval :: Canvas -> CCoord -> Coord -> [Conf COva] -> GUI COval`	A
cline'	`cline' :: Canvas -> [CCoord] -> [Conf CLin] -> GUI CLine`	A
Cline	`cline :: Canvas -> [Coord] -> [Conf CLin] -> GUI CLine`	A
crectangle'	`crectangle' :: Canvas -> CCoord -> CCoord -> [Conf CRec] -> GUI CRectangle`	A
Crectangle	`crectangle :: Canvas -> Coord -> Coord -> [Conf CRec] -> GUI CRectangle`	A
carc'	`carc' :: Canvas -> CCoord -> CCoord -> [Conf CAr] -> GUI CArc`	A
Carc	`carc :: Canvas -> Coord -> Coord -> [Conf CAr] -> GUI CArc`	A
cpoly'	`cpoly' :: Canvas -> [CCoord] -> [Conf CPol] -> GUI CPoly`	A
Cpoly	`cpoly :: Canvas -> [Coord] -> [Conf CPol] -> GUI CPoly`	A
ctext'	`ctext' :: Canvas -> CCoord -> [Conf CTex] -> GUI CText`	A
Ctext	`ctext :: Canvas -> Coord -> [Conf CTex] -> GUI CText`	A
cbitmap'	`cbitmap' :: Canvas -> CCoord -> [Conf CBit] -> GUI CBitmap`	A
Cbitmap	`cbitmap :: Canvas -> Coord -> [Conf CBit] -> GUI CBitmap`	A
cimage'	`cimage' :: Canvas -> CCoord -> [Conf CBit] -> GUI CBitmap`	A
Cimage	`cimage :: Canvas -> Coord -> [Conf CBit] -> GUI CBitmap`	A
cwindow'	`cwindow' :: Canvas -> CCoord -> PWidget w -> [Conf CWin] -> GUI CWindow`	A
Cwindow	`cwindow :: Canvas -> Coord -> PWidget w -> [Conf CWin] -> GUI CWindow`	A
Scrollbar	`scrollbar :: WPath -> [Conf Scr] -> GUI Scrollbar`	A
vscroll'	`vscroll' :: ScrollableY w => WPath -> PWidget w -> [Conf Scr] -> GUI Scrollbar`	A

hscroll'	hscroll’ :: ScrollableX w => WPath -> PWidget w -> [Conf Scr] -> GUI Scrollbar	A
Vscroll	vscroll :: ScrollableY w => PWidget w -> [Conf Scr] -> GUI Scrollbar	A
Hscroll	hscroll :: ScrollableX w => PWidget w -> [Conf Scr] -> GUI Scrollbar	A
entry'	entry’ :: WPath -> [Conf Ent] -> GUI Entry	A
Entry	entry :: Window -> [Conf Ent] -> GUI Entry	A
getEntry	getEntry :: Entry -> GUI String	A
setEntry	setEntry :: Entry -> String -> GUI ()	A
vscale'	vscale’ :: WPath -> [Conf Sca] -> GUI Scale	A
hscale'	hscale’ :: WPath -> [Conf Sca] -> GUI Scale	A
Vsacle	vscale :: Window -> [Conf Sca] -> GUI Scale	A
Hscale	hscale :: Window -> [Conf Sca] -> GUI Scale	A
insertListbox	insertListbox :: Listbox -> LIndex -> [String] -> GUI ()	A
edit'	edit’ :: WPath -> [Conf Edi] -> GUI Edit	A
Edit	edit :: Window -> [Conf Edi] -> GUI Edit	A
setMarkGravity	setMarkGravity :: Mark -> Gravity -> GUI ()	A
getMarkGravity	getMarkGravity :: Mark -> GUI Gravity	A
tagId	tagId :: Tag -> TagId	A
tag'	tag’ :: Edit -> TagId -> [TIndex] -> [Conf Tg] -> GUI Tag	A
Tag	tag :: Edit -> [TIndex] -> [Conf Tg] -> GUI Tag	A
selectionTag	selectionTag :: Edit -> GUI Tag	A
Embedded	embedded' :: Edit -> WPath -> TIndex -> [Conf Ew] -> GUI Embedded	A
embedded'	embedded :: Edit -> PWidget a' -> TIndex -> [Conf Ew] -> GUI Embedded	A
getAllEmbedded	getAllEmbedded :: Edit -> GUI [WPath]	A
newState	newState :: a -> GUI (GUIRef a)	A
newGUIArray	newGUIArray :: Int -> a -> GUI (GUIArray a)	A
mkDialog	mkDialog :: a -> GUIRef (Maybe a) -> Window -> GUI a	A
parentWpath	parentWPath :: Widget a b -> WPath	B
Wtag	wtag :: Widget a b -> String	B
Wpath	wpath :: Widget c w -> WPath	B
Title	title :: Window -> String -> GUI ()	B
Destroy	destroy :: Window -> GUI ()	B
Text	text :: String -> Conf w	B
Command	command :: GUI () -> Conf w	B
Start	start :: GUI () > IO ()	B
Quit	quit :: GUI ()	B
Proc	proc :: IO a > GUI a	B
failGUI	failGUI :: IOError -> GUI a	B
tryGUI	tryGUI :: GUI a -> GUI (Either IOError a)	B
catchGUI	catchGUI :: GUI a -> (IOError -> GUI a) -> GUI a	B
Tcl	tcl :: [String] -> GUI String	B

tcl_	tcl_ :: [String] -> GUI ()	B
tcl_append	tcl_append :: WPath -> String -> WPath	B
Cascade	cascade :: Menu -> Menu -> [Conf CB] -> GUI Cascade	B
cascade'	cascade' :: Menu -> Menu -> [Conf CB] -> Int -> GUI Cascade	B
gridAdd	gridAdd :: PWidget w -> Coord -> [GridInfo] -> GUI ()	B
gridForget	gridForget :: PWidget w -> GUI ()	B
gAnchor	gAnchor :: Anchor -> GridInfo	B
ginFrame	ginFrame :: Frame -> GridInfo	B
ginWindow	ginWindow :: Window -> GridInfo	B
menuButton	menubutton' :: WPath -> Maybe WPath -> [Conf MB] -> GUI Menubutton	B
menuButton'	menubutton :: Window -> [Conf MB] -> GUI Menubutton	B
getCoords	getCoords :: CWidget w -> GUI [(Int,Int)]	B
setCoords	setCoords :: CWidget w -> [Coord] -> GUI ()	B
insertEntry	insertEntry :: Entry -> EIndex -> String -> GUI ()	B
deleteEntry	deleteEntry :: Entry -> EIndex -> EIndex -> GUI ()	B
setICursor	setICursor :: Entry -> EIndex -> GUI ()	B
clearEntrySelection	clearEntrySelection :: Entry -> GUI ()	B
setEntrySelectionAnchor	setEntrySelectionAnchor :: Entry -> EIndex -> GUI ()	B
listbox'	listbox' :: WPath -> [Conf Lis] -> GUI Listbox	B
Listbox	listbox :: Window -> [Conf Lis] -> GUI Listbox	B
deleteListbox	deleteListbox :: Listbox -> LIndex -> LIndex -> GUI ()	B
resetListbox	resetListbox :: Listbox -> [String] -> GUI ()	B
clearListboxSelection	clearListboxSelection :: Listbox -> LIndex -> LIndex -> GUI ()	B
setListboxSelection	setListboxSelectionAnchor :: Listbox -> LIndex -> GUI ()	B
getEdit	getEdit :: Edit -> GUI String	B
getFromTo	getFromTo :: Edit -> TIndex -> TIndex -> GUI String	B
loadEdit	loadEdit :: Edit -> FilePath -> GUI ()	B
saveEdit	saveEdit :: Edit -> FilePath -> GUI ()	B
reserEdit	resetEdit :: Edit -> String -> GUI ()	B
deleteEdit	deleteEdit :: Edit -> TIndex -> TIndex -> GUI ()	B
insertEdit	insertEdit :: Edit -> String -> GUI ()	B
insertEditTagged	insertEditTagged :: :: Edit -> TIndex -> String -> [TagId] -> GUI ()	B
eqTIndex	eqTIndex :: Edit -> TIndex -> TIndex -> GUI Bool	B
ltTindex	ltTIndex :: Edit -> TIndex -> TIndex -> GUI Bool	B
gtTIndex	gtTIndex :: Edit -> TIndex -> TIndex -> GUI Bool	B
cmpTIndex	cmpTIndex :: Edit -> TIndex -> TIndex -> GUI Ordering	B

cutClipboard	`cutClipboard :: Edit -> GUI ()`	B
copyClipboard	`copyClipboard :: Edit -> GUI ()`	B
pasteClipboard	`pasteClipboard :: Edit -> GUI ()`	B
mark'	`mark' :: Edit -> MarkId -> TIndex -> GUI Mark`	B
Mark	`mark :: Edit -> TIndex -> GUI Mark`	B
getMarkPos	`getMarkPos :: Mark -> GUI (Int,Int)`	B
removeMark	`removeMark :: Mark -> GUI ()`	B
currentMark	`currentMark :: Edit -> Mark`	B
tagEdit	`tagEdit :: Tag -> GUI Edit`	B
readState	`readState :: GUIRef a -> GUI a`	B
writeState	`writeState :: GUIRef a -> a -> GUI ()`	B
modState	`modState :: GUIRef a -> (a->a) -> GUI ()`	B
readGUIArray	`readGUIArray :: GUIArray a -> Int -> GUI a`	B
getOpenFileName	`getOpenFileName :: GUI (Maybe String)`	B
getSaveFileName	`getSaveFileName :: GUI (Maybe String)`	B
tcl_eventUntil	`tcl_eventUntil :: GUIRef a -> (a -> Bool) -> GUI ()`	B
After	`after :: Int -> GUI () -> GUI Remover`	B
parseInt	`parseInt :: String -> Int`	B
Rgb	`rgb :: (Int,Int,Int) -> String`	B
tcl_callback	`tcl_callback :: String -> ([String]->GUI ()) -> GUI (String,GUI ())`	B
trapDeleteWindow	`trapDeleteWindow :: Window -> GUI () -> GUI ()`	B
getTclTime	`getTclTime :: GUI Double -- time in seconds since program started`	B
tcl_debug	`tcl_debug :: Bool -> GUI () – print debugging info or not`	B
bind	`bind :: Widget c w -> TkEvent -> GUI () -> GUI Remover`	C
Bindxy	`bindxy :: Widget c w -> TkEvent -> ((Int,Int)->GUI ()) -> GUI Remover`	C
bindXY	`bindXY :: Widget c w -> TkEvent -> ((Int,Int)->GUI ()) -> GUI Remover`	C
bindArgs	`bindArgs :: Widget c w -> (Bool,TkEvent,String) -> ([String]->GUI ()) -> GUI Remover`	C
Click	`click :: [String] -> GUI ()`	C
getMCheck	`getMCheck :: MCheckbutton -> GUI Bool`	C
setMCheck	`setMCheck :: MCheckbutton -> Bool -> GUI ()`	C
varMCheck	`varMCheck :: MCheckbutton -> String`	C
setRadio	`setRadio :: Radio -> Int -> GUI ()`	C
getRadio	`getRadio :: Radio -> GUI Int`	C
varRadio	`varRadio :: Radio -> String`	C
getRadio'	`getRadio’ :: Radio -> GUI WTag`	C
setRadio'	`setRadio’ :: Radio -> WTag -> GUI ()`	C
appendMRadio	`appendMRadio :: Radio -> MRadiobutton -> GUI ()`	C
removeMRadio	`removeMRadio :: Radio -> MRadiobutton -> GUI ()`	C
appendRadio	`appendRadio :: Radio -> Radiobutton -> GUI ()`	C
removeRadio	`removeRadio :: Radio -> Radiobutton -> GUI ()`	C
getCheck	`getCheck :: Checkbutton -> GUI Bool`	C

setCheck	`setCheck :: Checkbutton -> Bool -> GUI ()`	C
varCheck	`varCheck :: Checkbutton -> String`	C
canvas'	`canvas' :: WPath -> [Conf Can] -> GUI Canvas`	C
Canvas	`canvas :: Window -> [Conf Can] -> GUI Canvas`	C
citem_canvas	`citem_canvas :: CWidget a -> GUI Canvas`	C
citem_number	`citem_number :: CWidget a -> Int`	C
isEntrySelected	`isEntrySelected :: Entry -> GUI Bool`	C
setEntrySelection	`setEntrySelection :: Entry -> EIndex -> EIndex -> GUI ()`	C
adjustEntrySelection	`adjustEntrySelection :: Entry -> EIndex -> GUI ()`	C
setToEntrySelection	`setToEntrySelection :: Entry -> EIndex -> GUI ()`	C
getScale	`getScale :: Scale -> GUI Int`	C
setScale	`setScale :: Scale -> Int -> GUI ()`	C
getListboxEntris	`getListboxEntries :: Listbox -> LIndex -> LIndex -> GUI [String]`	C
getListboxSize	`getListboxSize :: Listbox -> GUI Int`	C
getListboxSelection	`getListboxSelection :: Listbox -> GUI [Int]`	C
setMark	`setMark :: Mark -> TIndex -> GUI ()`	C
getAllMarks	`getAllMarks :: Edit -> GUI [Mark]`	C
getMark	`getMark :: Edit -> MarkId -> Mark`	C
previousMark	`previousMark :: Edit -> TIndex -> GUI Mark`	C
nextMark	`nextMark :: Edit -> TIndex -> GUI Mark`	C
insertionMark	`insertionMark :: Edit -> Mark`	C
getAllTags	`getAllTags :: Edit -> GUI [TagId]`	C
getTagsAt	`getTagsAt :: Edit -> TIndex -> GUI [TagId]`	C
tagRemove	`tagRemove :: Tag -> [TIndex] -> GUI ()`	C
tagRanges	`tagRanges :: Tag -> GUI [((Int,Int),(Int,Int))]`	C
tagNextRange	`tagNextRange :: Tag -> TIndex -> TIndex -> GUI (Maybe ((Int,Int),(Int,Int)))`	C
tagPrevRange	`tagPrevRange :: Tag -> TIndex -> TIndex -> GUI (Maybe ((Int,Int),(Int,Int)))`	C
tagText	`tagText :: Tag -> GUI [String]`	C
writeGUIArray	`writeGUIArray :: GUIArray a -> Int -> a -> GUI ()`	C
modGUIArray	`modGUIArray :: GUIArray a -> Int -> (a->a) -> GUI ()`	C
Packadd	`packAdd :: PWidget w -> [PackInfo] -> GUI ()`	D
Geometry	`geometry :: Window -> Geometry -> GUI ()`	D
showWindow	`showWindow :: Window -> GUI ()`	D
hideWindow	`hideWindow :: Window -> GUI ()`	D
menuSize	`menuSize :: Menu -> GUI Int`	D
Raise	`raise :: PWidget w -> Maybe WPath -> GUI ()`	D
Lower	`lower :: PWidget w -> Maybe WPath -> GUI ()`	D
packForget	`packForget :: PWidget w -> GUI ()`	D

Expand	`expand :: Bool -> PackInfo`	D
packAnchor	`packAnchor :: Anchor -> PackInfo`	D
packPos	`packPos :: PlacePos WPath -> PackInfo`	D
checkButton'	`checkbutton' :: WPath -> [Conf Che] -> GUI Checkbutton`	D
checkButton	`checkbutton :: Window -> [Conf Che] -> GUI Checkbutton`	D
moveObject	`moveObject :: CWidget w -> Coord -> GUI ()`	D
removeObject	`removeObject :: CWidget w -> Maybe WTag -> GUI ()`	D
lowerObject	`lowerObject :: CWidget w -> Maybe WTag -> GUI ()`	D
raiseObject	`raiseObject :: CWidget w -> GUI ()`	D
bboxObjects	`bboxObjects :: [CWidget a] -> GUI (Int,Int,Int,Int)`	D
Xview	`xview :: ScrollableX b => Widget a b -> GUI (Double,Double)`	D
xMoveTo	`xMoveTo :: ScrollableX b => Widget a b -> Double -> GUI ()`	D
xScroll	`xScroll :: ScrollableX b => Widget a b -> ScrollUnit -> GUI ()`	D
Yview	`yview :: ScrollableY b => Widget a b -> GUI (Double,Double)`	D
yMoveTo	`yMoveTo :: ScrollableY b => Widget a b -> Double -> GUI ()`	D
yScroll	`yScroll :: ScrollableY b => Widget a b -> ScrollUnit -> GUI ()`	D
scanMark	`scanMark :: Scan w => Widget a w -> Int -> Int -> GUI ()`	D
scanDrag	`scanDrag :: Scan w => Widget a w -> Int -> Int -> GUI ()`	D
listboxMoveToSee	`listboxMoveToSee :: Listbox -> LIndex -> GUI ()`	D
addListboxSelection	`addListboxSelection :: Listbox -> LIndex -> LIndex -> GUI ()`	D
putPosTag	`putPosTag :: Edit -> TIndex -> String -> [Conf Tg] -> GUI Tag`	D
setWithTags	`setWithTags :: Edit -> TagId -> [TIndex] -> GUI ()`	D
lowerTag	`lowerTag :: Tag -> Maybe TagId -> GUI ()`	D
raiseTag	`raiseTag :: Tag -> Maybe TagId -> GUI ()`	D
Stretch	`stretch :: Bool -> Conf Ew`	D
Align	`align :: Align -> Conf Ew`	D

Appendix D: Experiment Data for Usability Experiment

The following steps were followed to test the hypotheses and evaluate the effect size of the findings

1. The data was first scanned for outliers. Outliers were found using Box-Whisker plots and were not considered for analysis.
2. The data were then analyzed for normality. *Kolmogorov Smirnov (K-S) tests* were performed on each data set. If the significance value of *K-S tests* exceeded the threshold value of 0.05 ($p > 0.05$), then the data was inferred to be normal.
3. If the data was normal, *dependent t-tests* were performed or else *Wilcoxon Signed Ranked (W-S) Tests* were performed to test the null hypotheses. If the significance value for the tests exceeded the threshold value of 0.05 ($p > 0.05$) the null hypothesis was accepted, otherwise the alternative hypothesis was accepted.
4. The effect size was calculated by calculating the correlation coefficient from the test statistic. The threshold values of small, medium and large effect sizes are $r = 0.10$, $r = 0.30$ and $r = 0.50$ respectively.

Table D.1 presents the details of the statistical analysis.

Variable	Test Technique	Mean	Normality Tests K-S tests			t- test					Verdict on Hypothesis	W-S tests		Effect Size	
			K-S Z	K-S p	Verdict	t	DF	sig (2 – tailed)	Mean Difference	Std Error Difference	Null Hypothesis?	W-S Test Z	W-S test sig	Effect Size	Effect Size verdict
Learn	T1	0.003	1.292	0.349	Normal	6.056	18	0.000	0.0015	0.00025	Rejected	N/A	N/A	0.61	Large
	T2	0.001	0.993	0.071	Normal										
Eff	T1	0.25	0.836	0.487	Normal	5.913	21	.000	0.1632	.0276	Rejected	N/A	N/A	0.75	Large
	T2	0.084	0.554	0.918	Normal										
E I	T1	2.82	1.607	0.011	Not Normal	N/A	N/A	N/A	N/A	N/A	Rejected	-2.885	0.004	-0.39	Medium
	T2	6.10	0.873	0.431	Normal										
Sat	T1	2.833	1.288	0.073	Normal	0.368	37	0.715	0.11	0.302	Accepted	N/A	N/A	0.06	Low
	T2	2.944	1.050	0.220	Normal										
Ease	T1	2.7381	1.093	0.183	Normal	2.592	37	0.014	0.062	0.240	Rejected	N/A	N/A	0.39	Medium
	T2	3.3611	1.212	0.106	Normal										
EffectP	T1	0.8668	1.163	0.133	Normal	N/A	N/A	N/A	N/A	N/A	Rejected	5.647	0.000	0.87	Large
	T2	0.1197	1.886	0.002	Not Normal										
$EffectP_{geric}$	T1	0.884	1.599	0.012	Not Normal	N/A	N/A	N/A	N/A	N/A	Rejected	5.579	0.000	0.86	Large
	T2	0.491	0.614	0.846	Normal										
EffP	T1	0.0018	0.839	0.482	Normal	5.227	21	0.000	0.0011	0.00021	Rejected	N/A	N/A	0.75	Large
	T2	0.0007	0.554	0.918	Normal										

Table D.1: Results of Statistical Analysis of Data

Bibliography

[1] A. A. Reyes and D. J. Richardson, "Siddhartha:A Method for Generating Domain-Specific Test Driver Generators," in *Proc. 14th IEEE International Conference on Automated Software Engineering*, Cocoa Beach, FL, 1999, pp. 81-90.

[2] A. Deursen and P. Klint, "Little languages: Little maintenance?" *Journal of Software Maintenance*, 10:75-92, 1998.

[3] A. Deursen, P. Klint and J. Visser, "Domain-specific Languages: An annotated bibliography," in *ACM SIGPLAN Notices*, vol.35, no. 6, pp. 26-36, 2000 http://www.cwi.nl/projects/dsl

[4] A. Field and G. Hole, *How to Design and Report Experiments* .London, UK: SAGE Publications, 2003.

[5] A. Hughes and D. Grawoig, *Statistics: A Foundation for Analysis* .Reading, MA: Addition Wesley Publishing Company, 1971.

[6] A. M. Memon, M. E. Pollack and M. L. Soffa, "Hierarchical GUI test case generation using automated planning," in *IEEE Transactions on Software Engineering*, Volume: 27 Issue: 2, Feb. 2001 pp: 144 –155.

[7] A. Sinha , H. Nejad, C. Smidts, A. Moran, J. Widamier , "Concurrent Modeling for High Assurance Software Test and Development", Fast Abstract, *Proceedings of DSN 2002*, pp. B-45-B-46.

[8] A. Sinha, C. Smidts and A. Moran, "Enhanced Testing of Domain Specific Applications by Automatic Extraction of Axioms from Functional

Specifications," in *Proc. 14th International Symposium on Software Reliability Engineering*, Denver, 2003, pp. 181-190.

[9] B. Beizer, *Software Testing Technique* .Boston, USA: International Thomson Computer Press, 1990.

[10] B. E. John, "Evaluating Usability Evaluation Techniques," in *ACM Computing Surveys*, [Online] 28 (4es). Available: http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a139-john/.

[11] B. Vaysburg, L. H. Tahat and B. Korel, "Dependence Analysis in Reduction of Requirement Based Test Suites," in *Proc .International Symposium on Software Testing and Analysis*, Roma, Italy, pp. 107-111, 2002.

[12] B. W. Boehm, C. Abts, A.W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. Reifer, B. Steece, *Software Cost Estimation With COCOMO II.* Prentice Hall, July 2000.

[13] C. E. Williams, "Toward a test-ready meta-model for use cases," *in Proc. Workshop on Practical UML-based Rigorous Development Methods,* Toronto, CA, 2001, 270–287.

[14] C. J. Wang and M. T. Liu, "Generating Test Cases for EFSM with Given Fault Models," in *Proc. IEEE Infocom*, 2, pp. 774-781, 1993.

[15] C. L. Heitmeyer, B. G. Labaw, "Consistency checks for SCR-style requirements specifications." Technical Report 9586, NRL, Washington DC, 1978.

[16] D. Bruce, " What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation".

[17] D. Chays, S. Dan, P.G. Frankl, F.I. Vokolos and E.J. Weyuker, "A Framework for Testing Database Applications," in *Proc. ISSTA 2000*, Portland, 2000, pp. 147-157.

[18] D. K. Peter and D. L. Parnas, "Using Test Oracles Generated from Program Documentation," in *IEEE Transactions on Software Engineering*, Vol 24, No. 3, pp. 161-173,1998.

[19] D. Leijen and E. Meijer, "Domain Specific Embedded Compilers," in *Proc.2nd USENIX Conference on DSL*, Austin, 1999, pp. 109-122.

[20] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Designs for Research.* Chicago: Rand McNally and Company, 1963.

[21] E. M. Clarke , J. M. Wing , et al., "Formal Methods: State of the Art and Future Directions", in *ACM Computing Surveys,* Vol. 28, No. 4, December, 1996, pp. 626-643.

[22] G. Arango, "Domain analysis: From art form to engineering discipline," in *Proc.5th International Workshop on Software Specification and Design*, Pittsburgh, 1989, pp. 152-159.

[23] G. E. Stark, R. C. Durst and T. M. Pelnik, "An Evaluation of Software Testing Metrics for NASA's Mission Control Center" [Online] MITRE, Available: http://hometown.aol.com/geshome/ibelieve/sqjsubm2.pdf.

[24] G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing," in *Planning Report 02-3*. Prepared by RTI for the National Institute of Standards and Technology (NIST), May 2002: see http://www.nist.gov/director/prog-ofc/report02-3.pdf

[25] *HaskellDB Manual*, Galois Connections Inc., 2002.

[26] *IEEE Guide to Software Requirements Specification,* IEEE Standard 830, 1984.

[27] J. C. Knight, C. L. DeJong, M. S. Gibble and L.G. Nakano, "Why Are Formal Methods Not Used More Widely?" in Proc. 4th NASA Formal Methods Workshop, Hampton, VA, 1997.

[28] J. Cohen, "A Power Primer," in *Psychological Bulletin*, Vol. 112, no. 1, pp. 155-159.

[29] J. Dick and A. Faivre, "Automating the Generation and Sequencing of Test Cases from Model-Based Specifications," in *Proc. First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, Odense, 1993, pp.268-284.

[30] J. Hughes, "Why Functional Programming Matters?," in *The Computer Journal*, vol.32, pp 98-107, April 1989.

[31] J. K. Chaar, M. J. Halliday, I. S. Bhandari and R. Chillarge, "In-Process Evaluation of Software Inspection and Test," in *IEEE Transaction on Software Engineering*, Vol 19, No. 11, November, 1993, pp. 1055-1070.

[32] J. Nielsen, *Usability Engineering* .San Diego, CA: Academic Press Inc.,1993.

[33] J. Siegel, *Introduction to OMG UML,* OMG Consortium, online http://www.omg.org/gettingstarted/what_is_uml.htm

[34] J. T. Huber, "Efficiency and Effectiveness Measures to Help Guide the Business of Software Testing", in *Applications of Software Measurement*, HP

Labs Research Report, 1999, [Online]. Available: http://www.benchmarkqa.com/PDFs/efficiency_measures.pdf.

[35] J. Tretmans and A. Belinfante, "Automatic testing with formal methods," in *Proc. EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November pp. 8-12, 1999. EuroStar Conferences, Galway, Ireland.

[36] James McDonald and John Anton. "SPECWARE - Producing Software Correct by Construction." *Kestrel Institute Technical Report* KES.U.01.3., March 2001.

[37] K. Finney, "Mathematical notation in formal specification: Too difficult for the masses?" in IEEE Transactions on Software Engineering, Volume 22, No 2, pp.158-159, 1996

[38] Krasner and Pope, *A Cookbook Approach to Using MVC*, JOOP, 1(3): 26-49.

[39] L. J. Jagadeesan, A. Porter, C. Puchol, C. J. Ramming and L. G. Votta, "Specification-based testing of reactive software: tools and experiments," in *Proc.19th International Conference on Software Engineering*, Boston, 1997, pp. 525–535.

[40] Luqi and J. Goguen, "Formal Methods: Problems and Promises," in IEEE Software, Volume 14, No 1, pp 73-85, 1997.

[41] M. Antoniotti and A. Göllü, "SHIFT and SMART-AHS: A language for hybrid system engineering modeling and simulation." Ramming J. C., editor, *Proceedings of the USENIX Conference on Domain-Specific Languages*, Berkeley, CA, October 15-17 1997, pages 171-182.

[42] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, "Model-Based Testing with AsmL.NET," *in Proc. 1st European Conference on Model-Driven Software Engineering ,* December 2003.

[43] M. Sage, "TclHaskell-User Manual", August 1999

[44] M. Van den Brand, A. Van Deursen, P. Klint, S. Klusener, and E. Van der Meulen, "Industrial applications of ASF+SDF", Wirsing M. and Nivat M., editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 9-18. Springer-Verlag, 1996.

[45] M. Kwan. Graphic programming using odd and even points. *Chinese Math.*, 1, pp. 273-277, 1962

[46] P. A. Stocks, P. A., D. A. Carrington, "Deriving Software Test Cases from Formal Specifications," in *Proc.6th Australian Software Engineering Conference*, Sydney 1991, pp. 327-340.

[47] P. Hudak , "The Haskell School of Expression", Cambridge University Press, NY, 2000.

[48] P. Hudak, , "Conception, Evolution and Application of Functional Languages", *ACM Computing Surveys*, Vol.21, No.3, September 1991

[49] P. Savage, S. Walters and M. Stephenson, " Automated Test Methodology for Operational Flight Programs," in *Proc. IEEE Aerospace Conference,* vol.4, pp. 293-305, 1997.

[50] P.A.V. Hall, "Relations between Specifications and Testing," in *Information and Software Technology*, vol.33, no. 1, pp. 47-52, 1991.

[51] R. Chillarege, I.S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Y. Wong, "Orthogonal Defect Classification – A Concept for In-Process Measurements," in *IEEE Transactions on Software Engineering,* Nov 1992.

[52] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir, "Test Development For Communication Protocols: Towards Automation," in *Computer Networks*, 31, 1999, pp. 1835-1872.

[53] *Rational Rose Enterprise Edition*, IBM Corporation, New York, NY 2004.

[54] *Rational XDE Tester User's Guide,* IBM Corporation., New York, NY, 2004

[55] S. Chandra, B. Richards, and J. R. Larus, "Teapot: A domain-specific language for writing cache coherence protocols", *IEEE Transactions on Software Engineering*, 25(3), May/June 1999, pages 317-333.

[56] S. Thompson, *Haskell: The Craft of Functional Programming*, Essex: Addison-Wesley, 1999.

[57] *Silktest User's Guide*, Version 6.5, Segue Software Inc., Lexington, MA, 2002.

[58] *Test Master User's Guide*, Release 1.9.5, Empirix Inc., New Hampshire, 1999.

[59] *Testing Tool Information*, Grove Consultants, 2002.

[60] University of Massachusetts, CS 530 Lecture Slides, 2001.

[61] V. Basili, " Software Quality Assurance and Measurement: A Worldwide perspective", in *Applying the Goal/question/Metric Paradigm in the*

Experience factory, Chapter 2, pp 21- 44, International Thomson Computer Press, ITP An International Thomson Publishing Company, 1995.

[62] *WinRunner User's Guide,* Version 7.01, Mercury Interactive Inc., Sunnyvale, CA, 2001.