CRANFIELD UNIVERSITY

Ambreen Hussain

Use of Domain Specific Languages in Test Automation

School of Engineering
Software Test Automation

MSc by Research
Academic Year: 2012 - 2013

Supervisor: Dr. Stuart Barnes
April 2013

CRANFIELD UNIVERSITY

School of Engineering
Software Test Automation

MSc by Research

Academic Year 2012 - 2013

Ambreen Hussain

Use of Domain-Specific Language in Test Automation

Supervisor: Dr. Stuart Barnes
April 2013

This thesis is submitted in partial fulfilment of the requirements for the degree of MSc
(NB. This section can be removed if the award of the degree is based solely on examination of the thesis)

© Cranfield University 2013. All rights reserved. No part of this publication may be reproduced without the written permission of the copyright owner.

ABSTRACT

The primary aim of this research project was to investigate techniques to replace the complicated process of testing embedded systems in automotive domain. The multi-component domain was composed of different hardware to be used in testing procedure which increased the level of difficulty in testing for an operator. As a result, an existing semi-automated testing procedure was replaced by more simpler and efficient framework (ViBATA). A key step taken in this scenario was the replacement of manual GUI interface with the scriptable one to enhance the automation. This was achieved by building a Domain-specific language which allowed test definition in the form of human readable scripts which could be stored for later use.

A DSL is a scripting language defined for a particular domain with compact expressiveness. In this case the domain is testing embedded systems in general and automotive systems in particular. The final product was a test case specification document in the form of XML as an output of generated code from this DSL which will be input to ViBATA to make test specification component automated.

In this research a comparative analysis of existing DSLs for alternative domains and investigation of their applicability to the presented domain was also performed. The technologies used in this project are Xtext to define the DSL grammar, Xtend to generate code in Java and Simple framework to generate output in XML. The stages involved in DSL development and how these stages were implemented is covered in this thesis.

The developed DSL for this domain is tested for automotive and calculator systems in this thesis which proved that this is more general and flexible. The DSL is consistent, efficient and automated test specification component of testing framework in embedded systems.

Keywords:

Xtext, Xtend, Eclipse, Xbase, System Testing, Automotive Systems

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Stuart Barnes to confide in me and giving this opportunity first and secondly for mentoring me from start till end of this research by arranging meetings and resolving issues I had. I want to thank my husband Mr. Mohsin as well for making nice curries for me. His support and encouragement was very important for me to achieve these results. Last but not least I am very much grateful to my parents, family and friends for their prayers and wishes.

TABLE OF CONTENTS

ABSTRACT i
ACKNOWLEDGEMENTS ii
LIST OF ABBREVIATIONS vi
Glossary viii
1. INTRODUCTION 1
  1.1 Kinds of DSL 2
    1.1.1 External DSL 2
    1.1.2 Internal DSLs 3
  1.2 Benefits of building a DSL 5
    1.2.1 Increase development productivity 5
    1.2.2 Better communication with people in Domain 5
    1.2.3 Change in Execution context 6
    1.2.4 Alternative Computational Model 6
  1.3 Problems with DSLs 7
    1.3.1 Difficulty in learning languages 7
    1.3.2 Building Cost 7
    1.3.3 Densely populated Language 8
    1.3.4 Blinkered Abstraction 8
  1.4 Motivation 8
  1.5 Aims and Objectives 9
2 Literature Review 12
  2.1 Test Automation Techniques 12
    2.1.1 Testing Framework/Workbench 12
    2.1.2 Record/Playback Testing (R/P) 14
    2.1.3 Model-Based Test Automation 15
  2.2 Two approaches to perform Domain Analysis 16
  2.3 External or Internal DSL 19
  2.4 Textual or Graphical DSL 19
  2.5 Tool comparison 21
    2.5.1 Comparison of MSDSL tools and Eclipse modelling plug-ins Framework 21
    2.5.2 A Comparison of Tool Support for Textual Domain-Specific Languages 22
  2.6 Model Based Testing in Automotive Systems 23
  2.7 Example Implementations of DSL based Systems 24
    2.7.1 A DSL for Simulation Composition 24
    2.7.2 CAST: Automated Software Tests for Embedded Systems 25
    2.7.3 Habitation: A DSL for Home Automation 27
    2.7.4 A Domain-Specific Language for Ubiquitous Healthcare 27
    2.7.5 Domain Specific language for Cellular Interactions 28

2.7.6 A DSL in Embedded Systems 29
2.7.7 MobDSL 29
2.7.8 SLCO 30
3 Theory and Technologies 32
3.1 Technologies used in the Project 32
3.1.1 Eclipse Xtext 32
3.1.2 Xtend 33
3.1.3 Simple Framework 35
3.2 Development Stages of DSL 35
3.2.1 Domain Analysis and DSL Behaviour 35
3.2.2 Define Concrete Syntax and Rules (Grammar) 36
3.2.3 Development of Language Artefacts 38
3.2.4 Model Constraint 38
3.2.5 Integrating DSL with target Platform 39
3.2.6 DSL to platform Transformation 40
3.3 Software Testing 42
4 Methodology 44
4.1 1st generation testing procedures - ControkDesk & Python scripts 44
4.2 2nd generation testing procedures - ViBATA 46
4.3 Overview of DSL 55
4.4 Implementation of Development stages of DSL 56
4.4.1 Domain Analysis 56
4.4.2 Using Domain Elements to Create Grammar Rules 57
4.4.3 Writing Code Generator in Xtend 62
4.4.4 Model Validation 72
4.4.5 Model Scoping 74
4.4.6 Content Assist 76
4.5 DSL to Platform Transference 77
4.5.1 Program in DSL 78
4.5.2 Generated Code 80
4.5.3 Output of the Code 82
4.6 XML Plugin for DSL output in JLR Project 82
5 Results, Analysis and Discussion 85
5.1 Use Cases 85
5.2 Validation of Use Cases 85
5.2.1 Define Environment and Test Case 85
5.2.2 Define Test Setup 95
5.2.3 Defining XML File and Location 96
5.3 Research Questions 99
5.4 Implications 100
5.4.1 Can this DSL work with other embedded system? 100
5.4.2 What will happen if Device Changes 105

5.4.3 What will happen if DSL program variable changes.................. 106
5.4.4 What will happen if user selects an XML file having different elements.. 106
6 Conclusion and Future work... 108
6.1 Future Work 109
REFERENCES... 111
APPENDICES 118
Appendix A Modifications done in Software 118

LIST OF ABBREVIATIONS

JLR	Jaguar Land Rover
ABB	Asea Brown Boveri
IPC	Instrument Panel Cluster
APT	Automatically Programmed Tool
BNF	Backus-Naur Form
DSL	Domain-Specific Language
GPL	General Purpose Language
ViBATA	Visual Based Test Automation
AST	Abstract Syntax Tree
EBNF	Extended Backus-Naur Form
ECU	Electronic Control Unit
SUT	System Under Test
TAF	Test Automation Framework
ACT	Air Traffic Controller
GMF	Graphical Modelling Framework
OMG	Object Management Group
TPT	Time Partitioning Testing
HiL	Hardware in the Loop
MiL	Model in the Loop
SiL	Software in the Loop
MDE	Model Driven Engineering
oAW	OpenArchitectureWare
API	Application Programming Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
JDT	Java Development Tools
OCL	Object Constraint Language
TSM	Test Specification Manager
TCM	Test Configuration Manager
TEM	Test Execution Manager
HD	Hardware Driver
TD	Test Driver
TAC	Test Automation Core
MIC	Model-Integrated Computing
CAST	Computer-Aided Specification and Testing
TESLA	Test Specification Language

Glossary

MiL: *Model-in-the-Loop testing refers to the kind of testing done to verify the accuracy / acceptability of a plant model or a control algorithm. [75]*

HiL: *Hardware-in-the-Loop refers to a process in which an embedded system (e.g. real electronic control unit or real mechatronic component) via its inputs and outputs to a matched counterpart, which generally HiL Simulator is known and serves as a replica of the real environment of the system is connected. [76]*

SiL: *In the method of software in the loop (SiL) as opposed to HiL no special hardware is used. The created model is the software only converted to the code understood by the target hardware (for example, a MATLAB / Simulink model to C-code). This code is executed on the development computer, together with the simulated model, instead of running as hardware in the loop on the target hardware*

1. INTRODUCTION

A Domain-Specific Language (DSL) is a small computer programming language that focuses on particular domain with limited expressiveness [1]. It is not a new technology as the concept has been there since the 1950s. Examples include Automatically Programmed Tool (APT), a DSL for numerically controlled machine tools programming, developed in 1957-1958 and Backus-Naur Form (BNF) is the well-known syntax specification formalism developed in 1959 [30]. The opposite approach to DSL is a GPL (General Purpose Languages) such as Java and C#. Although a GPL can be used to solve any kind of computing problem, it might not always give the best solution. A key difference between a GPL and a DSL is scope of DSL is limited to a specific problem domain while GPL's scope is much wider. GPL follows an imperative computation model which tells the computer what should happen in what sequence and how it should happen by using conditional statements, variables and loops in program. The program in GPL does not display the intent of the program instead a sequence of steps. While DSL uses declarative programming model which concentrates more on what should happen instead of how it should happen. Code written using DSL shows the intent of the program [1]. A DSL can also adopt imperative computational model mostly technical DSLs does it but it will still hide a lot of information about the code [2]. DSL improves developers' productivity and communication with domain user. Examples of DSL include, Regular expressions for text processing, Logo for pencil like drawing, Hyper Text Markup Language (HTML) [45] and Ruby on Rails for building web applications, Cascading Style Sheets (CSS) [46] for defining style of elements on web page and Structured Query Language (SQL) [47] for relational databases. [44]

1.1 Kinds of DSL

There are two main kinds of DSL: External DSL and Internal or embedded DSL. In this section details of these styles of creating DSL are given

1.1.1 External DSL

An external DSL is a language with custom or borrowed (XML) syntax, separate from the main language of the application it works with. This custom syntax is formed by defining the grammar for DSL using notation like BNF, or Extended Backus-Naur Form (EBNF) based Xtext. The grammar is a collection of rules defined to make the syntax of language. A tool such as Another Tool for Language Recognition (ANTLR) [48] or GNU Bison [49] generates a parser by running over the code and produces an abstract syntax tree (AST). Program written in an external DSL can be interpreted directly or can generate code in a GPL to execute in target platform. The most common examples are SQL, CSS, Regular expressions and XML configuration files. For example consider text processing to validate an Israeli phone number 03-9876543. A code in GPL to do this is shown in Figure 1.1

```
public bool ValidatePhoneNumber(string input)
{
    if (input.Length != 10)
        return false;
    for (int i = 0; i < input.Length; i++)
    {
        if (i == 2 && input[i] != '-')
            return false;
        else if (char.IsDigit(input[i]) == false)
            return false;
    }
    return true;
}
```

Figure 1.1 Code in GPL to validate phone number [2]

The code in figure 1.1 is concentrating more on how to check an input string and validate if it is a phone number. This code is difficult to understand for a non-programmer who will need an effort to comprehend it. Now consider using a tool which is dedicated to text processing known as Regular Expressions.

Same task using this tool is confined to one line code only shown in Figure 1.2 [2]

```
public bool ValidatePhoneNumber(string input)
{
    return Regex.IsMatch(input, @"^\d{2}-\d{7}$");
}
```

Figure 1.2 code in Regular Expression to Validate Phone Number [2]

The code in Figure 1.2 is completing intent of DSL with less code without stating how it is done in quite clearer way but to understand this line of code one will need to understand syntax of the DSL in this case it is regular expressions.

1.1.2 Internal DSLs

Also known as embedded DSLs, an internal DSL is a particular way of using an existing GPL. An internal DSL uses a subset of the host language's features in a particular style to handle one small aspect of the system. Examples of internal DSL are Lisp, Ruby etc. Consider the following example which defines the difference between the two kinds of DSL. In this example we are having a problem of designing set of shapes and want to design a graphical modelling tool. A grammar with some rules for this DSL is shown in figure 1.3 [16]

```
Definitions ::= Definition*
      Definition ::= Define Id Shape
      Width Eq Number
      Height Eq Number

      FillColor Eq Color
      OutlineColor Eq Color
      Decorator*
End Id

Shape ::= Rectangle | RoundedRectangle | Ellipse

Eq ::= "="

Decorator ::= Decorator Id
      Position Eq Position
End Id

Position ::= Center|
             TopLeft |
             TopRight |
             BottomLeft |
             BottomRight
```

Figure 1.3*: Grammar for the DSL [16]*

In this grammar it is assumed the definitions for Id, Number and Color are defined in grammar defining language. The rule Definitions is having arbitrary number of rule Definition which is having some keywords such as Define, Width and Height. 'Eq' points to another rule which defines equal (=) sign. Rule 'Decorator' defines position which is another rule. The following snippet of code is defining a function for Rectangle shape using this grammar Figure 1.4

```
Define AnnotationShape Rectangle
      Width=1.5
      Height=0.3
      FillColor=khaki
      OutlineColor=brown
      Decorator Comment
            Position="Center"
      End Comment
End AnnotationShape
```

Figure 1.4 ***an example of code in external DSL [16]***

This form of making DSL in which grammar is made first and parsed by a parser generator known as external DSL but the same objective can be easily achieved by the following script in C# (Figure 1.5) by using libraries and structures previously defined for the shapes and drawings.

```
Shape AnnotationShape = new Shape(ShapeKind.Rectangle,
                                  1.5,
                                  0.3,
                                  Color.Khaki,
                                  Color.Brown);
Decorator Comment = new Decorator(Position.Center);
AnnotationShape.AddDecorator(Comment);
```

Figure 1.5 an internal DSL example [16]

Both of above techniques of creating DSL achieve the same goal. The intent of the code is clear, expressive and complete. The focus is limited and approach is declarative.

1.2 Benefits of building a DSL

DSLs are tools with limited focus and are not like object-oriented or agile processes of developing software. DSL is a thin coating over a model where the model can be a library or framework. The benefits of DSL should be kept separate from those provided by model. DSLs have certain benefits which are defined in this section. When anyone considers creating a DSL he should keep these benefits in mind and decide which is applicable to his circumstances. [1].

1.2.1 Increase development productivity

Main advantage of a DSL is that it delivers the objective of system in more clear and concise way. There is less probability of defects in code due to limited expressiveness. The clarity of code makes it easier to write the code and easier to find the defects. Defects in the system impact productivity because it takes time to debug, find and fix these. The model alone provides quite substantial improvement in productivity. It avoids duplication by gathering common code; it also provides abstraction which makes easier to understand the problem. DSL enhances benefits by providing more expressiveness to read and manipulate the abstraction, thus increase development productivity. It can help people to learn how to use an API and how different methods in API should be combined together [1].

1.2.2 Better communication with people in Domain

The main reason of any software's failure is lack of communication between its user and developer. DSL can improve this communication by providing a language focused on a particular domain. This benefit does not fit for every type of DSL such as for regular expression. Because regular expressions exhibit complex structure to solve the problem in text processing. The user needs to learn each symbol to define an expression for processing text such as text to validate email address as shown in section 1.1.1. It is a common argument that with DSLs there will be no need of programmers anymore but that is not true. Domain experts will not compose DSLs but only read, understand and write programs using the language [1]. In this way they can find faults easily.

Involving domain experts can help to perform ‘Domain Analysis’ to build domain models which will be described in section 2.2.

1.2.3 Change in Execution context

The reason that generated code can run in different environment is main driver of using DSL. This usage brings limitations in case of internal DSL because it uses host language to process. A model can be executed directly or code can be generated from it. DSL allows execution of same behaviour in different language environments using code generation. One can create business rules to generate code in C# and Java or validations can be defined which can run in C# on the server and JavaScript on the client [1].

1.2.4 Alternative Computational Model

A GPL uses an imperative computation model which means instructing the computer to do things in a specific sequence, use conditional statements to handle control flow, loops and variables. A software can be developed with imperative logic but after a while developers think it could be done better with Dependency Network e.g. to run a test compilation always need to be updated. So the languages such as Ant which are designed to describe builds use dependencies between tasks as primary structuring mechanism. This kind of non-imperative programming also known as declarative programming because it allows declaring what should happen instead of describing how should happen. The behaviour of alternative computation model comes from Semantic model. DSL makes it much easier for people to manipulate declarative programs because it populates the semantic model [1]

The reasons because of which someone would be interested in making DSLs are described in similar way by [2]

1. To make a technical task simpler for domain expert because of limited expressiveness.
2. To express actions and rules using terms related to a particular domain which are familiar to people in domain
3. To replace manual system by automating task and actions.

1.3 Problems with DSLs

One should not decide to create a DSL if the benefits given above are not applicable to his problem or the benefits are not worth the cost of building the DSL. Many problems with DSLs are related to any particular style of building DSL. Even if a DSL is worth applying different problems arise that are overstated because usually people are not familiar with how to develop a DSL. This section defines problems with DSL mentioned by [1]

1.3.1 Difficulty in learning languages

One problem people report is difficulty in learning different languages if a project has more than one DSL. They underestimate how hard to learn a GPL. Every project has some abstractions in codebase which needs an effort to learn. If a project is using a GPL, it will be using different libraries to capture those abstractions. A DSL is much simpler to learn than a GPL but the question is how hard it is to learn a model underlying a DSL on its own. A DSL makes it easier to understand and manipulate that model which reduces the learning cost.

1.3.2 Building Cost

As there is code to write and maintain a DSL that requires a small building cost. A DSL should not be developed if the benefit is limited. Every library cannot be benefited by having DSL wrapper over it such as if command-query API is working fine then there is no need to build another API on the top of it. Maintaining a DSL is quite crucial, a simple internal DSL can be problematic if most of members of development team find it hard to understand and with parsers external DSL is intimidating for them. One thing which increases the cost of DSL development is that people are not used to building it and there are new techniques to learn. Although these costs should not be ignored they can lessen with time. The cost of building DSL is the cost over the cost of building a model. Every complicated area has some mechanism to overcome its

complexity, if it is complicated to build a DSL then it is complicated enough to benefit from a model. A DSL can help to think about a model (library or framework) and reduce its building cost. It can make it easier to deal with bad library by wrapping it up.

1.3.3 Densely populated Language

If a company builds its systems using an in-house built language, it becomes difficult to hire new staff and keep up with technology change. A DSL should not have too much functionality that it accidently becomes a GPL. A focus on its limited expressiveness should not be ignored. If it needs more functionality it is better to consider creating more than one language and combine them instead of making one DSL too big. Secondly for a particular problem if there is already a DSL available and it is open source, it is better to use that instead of making one from scratch.

1.3.4 Blinkered Abstraction

A DSL always has some abstraction which enables to think about a subject area and allows expressing the behaviour of the domain in easier way. Blinkered abstraction is something that puts blinkers on one's thinking and does not fit in the abstraction. It takes a lot of time and effort to fit it in instead of changing abstraction to absorb the new behaviour. With any abstraction, a DSL should be looked like something evolving not finished [1].

1.4 Motivation

The need of Domain-Specific Language for test automation in this project came from software needed by testing team at JLR (Jaguar Land Rover). This software named as ViBATA (Visual Based Test Automation) is built by Cranfield University and is currently working at JLR. The purpose of this software is to replace a tightly coupled semi-automated testing system. The previous manual testing involved reading test case from Excel sheets; sending signals to

Instrument Panel Cluster (IPC) with the help of graphical component of ControlDesk which is experiment software for seamless Electronic Control Unit (ECU) development; and getting output on IPC. This manual testing was replaced by semi-automated testing system which introduced the use of camera to capture output as an image which could be recorded by software named Insight. The details of this previous implementation of testing procedure are given in section 4.1 of this thesis.

Transference of test cases from Excel sheet to ViBATA is done by efficient functionality which allows copying a test case and paste it on the software but is still manual. On executing a test, input lines of the test case can send signals to the IPC and output lines compare the result obtained from camera with the expected outcome. The detailed overview of the software is described in section 4.2.

Now the problem is test case transference in ViBATA is manual. User need to copy each test case and paste it onto the software. This transference can be made automated by introducing even more efficient programming code which could read excel sheet and recognise test case and enter into the system in their respective categories and IPCs. But would this functionality be consistent with every release of test case specification excel sheet and enter test cases without any mistake. This question gave the idea of using Domain-Specific language because of its limited expressiveness, clarity and descriptive nature. With DSL test case transference can be made automated and it can bring a lot of flexibility.

1.5 Aims and Objectives

To automate the test case transference DSL will be the best choice because of its declarative nature and limited expressiveness. Test cases could be defined by using a interface but that would not be that efficient as DSL could be. Different versions of SUT will have same test case specifications with little detail

changed which can be made easily by using DSL and test cases can be created for each version of SUT in no time. Scripting language is always a good choice to specify test cases in any testing system. The main objectives of this research in this regard are

1. Build a domain-specific language to provide domain user with a facility to define test cases and information about device used. He can define test setup. He should also be able to update and delete the test cases
2. Language should be easy to understand and learn for domain user
3. The output of the code generated by program written in DSL should be consistent and readable for ViBATA
4. The code generation from DSL should be flexible which will bring the novelty in testing embedded systems
5. Language should be able to detect errors
6. Language should facilitate user with code completion

The approach of using DSL in domain of testing embedded system is also used by Wahler [9] at ABB [50]. The testing framework is decoupled and language is external type of DSL with custom syntax explained in detail in section 2.7.2. The novelty brought by current study is the introduction of flexible code generation. Wahler used Scala interpreter to execute the language instructions while this approach will use code generator to produce code in developer's choice GPL. Two research questions are also observed during the development of this DSL, first is what are the characteristics of the DSL for testing embedded systems and second is what we need to extend it to specific environment i.e. automotive. Both of these research questions are answered in detail in section 5.3.

In this chapter an introduction to DSL, its kinds, benefits and problems with DSLs are given. The problem in automation of testing for embedded system is also mentioned which became the motivation to build a DSL. Also aims and objectives of this thesis are described in this chapter.

2 Literature Review

This chapter gives first introduction to work observed by people in the field of test automation, comparison between tools and types of DSL, and then current state of the art is discussed.

2.1 Test Automation Techniques

In this section different forms of test automation are discussed such as Testing framework [5], Record/Playback [18] and model-based test automation [20].

2.1.1 Testing Framework/Workbench

Testing framework like JUnit is one of test automation approaches used for regression testing. JUnit is set of Java classes that user can extend to build an automated testing framework. Individual test is an object which is executed by the test runner. The tests should be written in a way that shows whether the tested system has behaved as expected. A software testing workbench consists of tools is used to perform testing. Apart from the ability that facilitate automated test execution testing workbench may also provide functionality to simulate other parts of the system and to generate test data [5].

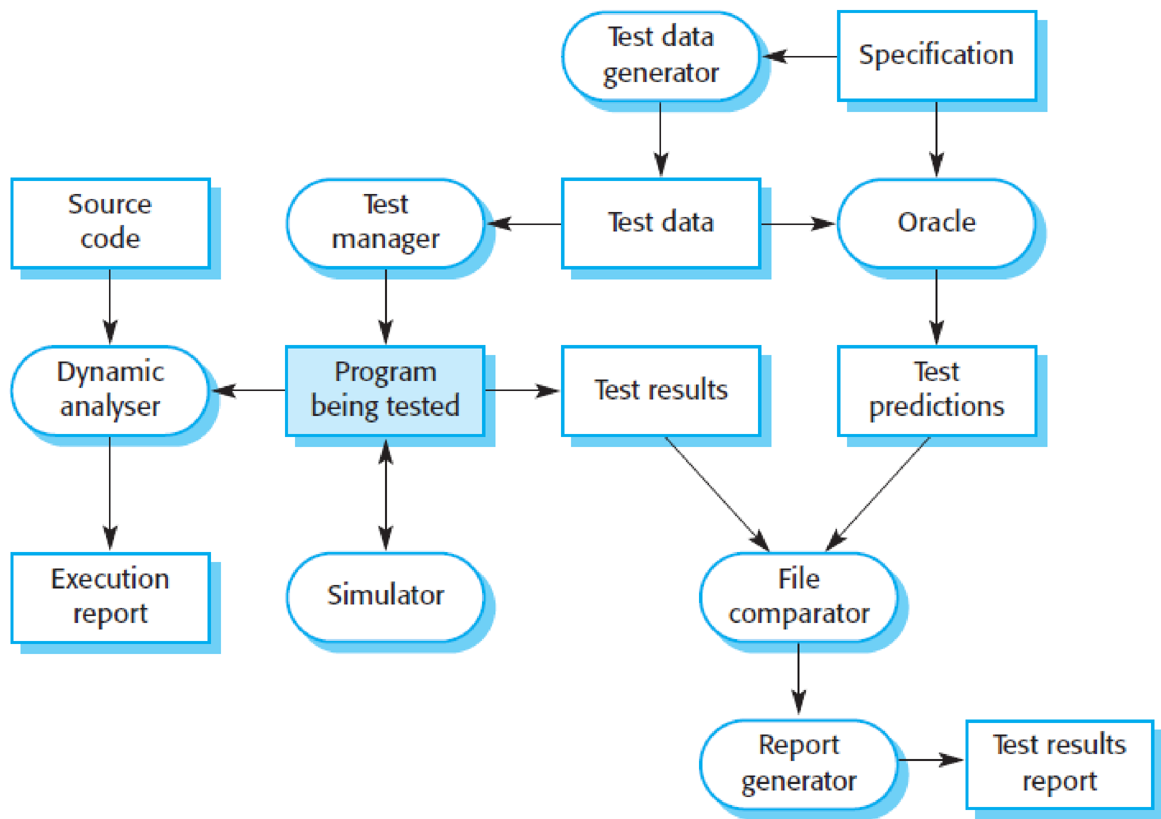

Figure 2.1 ***A Testing Workbench [5]***

A testing workbench is shown in Figure 2.1 which might have tools illustrated below

A **Test Manager** manages the whole system of running tests. It keeps track of test data, expected results and program facility tested. Example is JUnit

A **Test Generator** generates data for the program to be tested. Data can be fetched from database or by using patterns to generate random data

Oracle provides the predictions of expected test results. An oracle can be either previous version of the program or prototype systems. Back-to-back testing is running the oracle and program under test in parallel and differences in their outputs are noted.

A **File Comparator** compares the test results with the previous results and reports the differences. Comparators are usually used in regression testing where test results of different program versions need to be compared.

A **Report Generator** provides report definition and generation facilities for the test results

A **Dynamic Analyzer** analyses the number of times each statement in the program is executed and generates execution profile

Simulator: Target simulator simulates the machine where program will run. "*User Interface simulators are script-driven programs that simulates multiple simultaneous user interactions*" [5]

There are many advantages of using automated test tools. It is easy to execute regression tests automatically with a press of single button without any attendee overnight or on weekend. It provides the ability to rerun all automated test cases or selected subset of test cases against new build or release and a confidence that modifications in the system have not impacted adversely on existing functionality [4].

2.1.2 Record/Playback Testing (R/P)

In record and playback type of test automation, user performs actions on UI of System under Test (SUT) which are recorded in the form of test tool's language script when it is in the record mode. These scripts can be replayed back into UI thus executing test automatically. Most commercial record/playback test tools are WinRunner [51], QARun [52], QuickTest Pro [53], and IBM Rational Robot [54] etc. In R/P testing each test run for once per release and on every release new test needs to be created because change in the system fails old recorded test so maintenance of testing scripts is very crucial [18].

There are certain limitations of Record/Playback testing which include: It is difficult to maintain scripts because of long list of user actions and re-running of tests sometimes interrupted because of synchronization problems. Data used for such recorded tests is hardcoded which is from software development point of view is not a good practice. Tests cannot handle unexpected error. Same kind of limitations are given by [19] like behaviour, interface, data and context sensitivity; if any of these changes the test fails making bad reputation of record/playback test automation.

There are ways suggested by [19] to make record/playback a successful mean of test automation which include making the system context insensitive by configuring it with a known starting point in terms of data and date. Whenever functionality changes a new test should be recorded but when UI changes there should be other tests which can check if it is changed so the tests for the business logic should not get failed.

Record/playback should only be considered when time, cost and programming skills of hand-written scripts is not affordable [19].

2.1.3 Model-Based Test Automation

A Model based automated testing approach with a use of Test Automation Framework (TAF), supports modelling methods for requirement and design representation. A tester creates a model from available information provided by requirements engineer. T-VEC, a test generation component of TAF, creates tests after models are translated. T-VEC supports test vector and driver generation; requirement test coverage analysis and test results reports. Test vector consist of inputs and expected outputs. A test generator takes in outputs from test vector and test driver mappings as inputs to produce test scripts. Test

scripts are then executed and text execution analysis compares the actual output with expected outputs and produces a test report [20].

Benefits of Model-based approaches like TAF include: use of models help in requirement defect analysis, automating test design, generating test scripts, saving cost and producing high quality code. Models can use same driver schema to produce test scripts. When system's functionality changes only models get updated, by using existing driver schema scripts are regenerated. But if the test environment changes the schema needs to be updated and scripts are regenerated without changing models. Parallel modelling during development life cycle helps identify defects at early stage because testing team starts work at the start of the project and stays involved throughout the process [20].

2.2 Two approaches to perform Domain Analysis

Domain analysis is the first stage of DSL development which involves gathering knowledge about domain and building domain model. For this project domain knowledge is obtained by working on the ViBATA software explained in detail in section 4.2 and also by continuous involvement of domain experts in the development of this software.

In this section two approaches taken by [21] and [22] to perform domain analysis are discussed. One way of doing domain analysis is to develop ontologies for the domain. If ontologies for a particular domain already existed then those can be used otherwise it is a beneficial approach to develop them first. (Tairas, Mernik and Gray) investigates ontology development during domain analysis phase of DSL development and its contribution to the language design. "*Ontologies seek to represent the elements of a domain through a vocabulary and relationships between these elements in order to provide some type of knowledge of the domain.[21]"* Authors discovered two properties of ontologies: one vocabulary representation of domain e.g. elements of domain and second relationship between those elements [21].

The domain model defines [21]

- scope of the domain,
- the domain terminology (vocabulary, ontology), descriptions of domain concepts
- Commonalities and variabilities of domain concepts and their interdependencies.

Two competency questions are proposed by [21] to serve the purpose of ontology: one what are the concepts of the domain and interdependencies between those concepts? And what are the commonalities and variabilities of the domain? They develop ontology using a tool Protégé 2000 [55] for a domain which focuses on communication between an air traffic controller (ATC) at the airport and pilot in a plane by defining classes, slots and allowed values for these slots and filling in values for slots for instances of those classes. From the class definition a class diagram is created from which initial context free grammar (CFG) is formed for this domain and ultimately a small program using this DSL.

The same process of domain analysis is done by [22] by describing domain abstractions as a sub process of main process 'Define DSL core Language Model'. Describing domain abstractions means defining domain entities or elements like classes for the class model. These abstractions integrated to form the core language model. Next step is to explain the relationship between entities and constraints for the abstractions followed by checking of completeness and correctness from domain-oriented perspective. Software engineers with the help of domain experts check the language model if it is complete and correct. In case there is need to add or change abstraction they repeat the whole process until it is accepted by both, the process is shown in Figure 2.2.

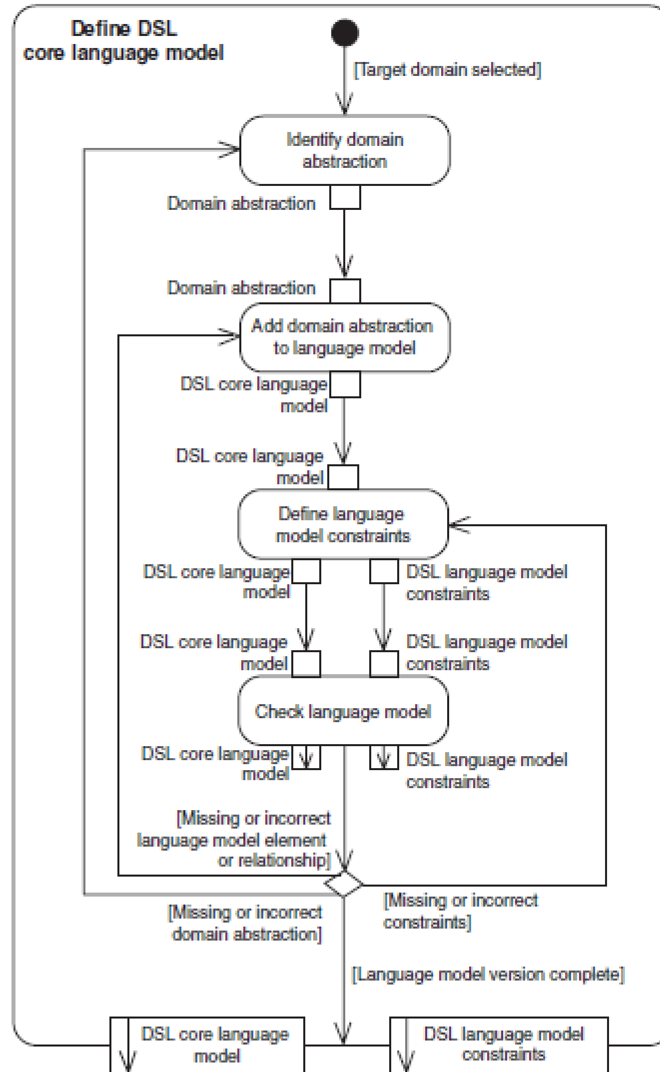

Figure 2.2 Subprocess define DSL core language model [22]

Certain guidelines are given by [23] for each activity of DSL development process out of which related to domain analysis phase 'Language Purpose' are: identifying the uses of the language, people who will use language should be asked questions by people who work on DSL development and the language should be platform independent.

2.3 External or Internal DSL

The authors of [12] have experience of creating several external DSLs in the field of trace analysis e.g. HAWK [56], EAGLE [57], RULER [58] and LOGSCOP [59], they observed two important things. One, it is difficult to amend an external DSL once it is created and secondly user demand features which can be handled more easily with general purpose programming language. This leaves an option to create an internal DSL instead. The authors created an internal DSL for trace analysis named as TRACECONTRACT [60] in SCALA [61]. They chose SCALA for two reasons: one, this language has built-in support for defining internal DSL; secondly, it supports functional as well as object oriented programming. Creating an internal DSL can be termed as shallow which means use of host language constructs as part of DSL, as well as deep which means a separate internal representation (abstract syntax) is made that is then interpreted or compiled like an external DSL. A shallow embedding is disadvantageous as it cannot be analysed easily. The arguments in favour of internal DSLs are: less effort is required to implement because of direct execution of DSL constructs; it gives direct tool support from the host language e.g. IDE, debugger, static analyser and testing tools. Disadvantages of an internal DSL include: it is difficult to analyse an internal DSL without working with the host language compiler; the domain user will need to be a programmer to work with DSL and will need to learn the big host programming language [12].

So in the light of arguments given above especially the learning costs involve for the DSL user in case of internal DSL, for the current project the decision is to make an external DSL.

2.4 Textual or Graphical DSL

After deciding the solution is DSL and gathering domain knowledge it is now time to decide which form of DSL to be made: a textual or graphical. A textual

DSL has syntax to write a program as described in section 1.1 whereas graphical DSL uses shapes and lines to express the intent rather than text. UML [62] is good example which uses activity diagrams, class diagrams and sequence diagrams for describing software systems. There are separate tools and plug-ins to create both kinds of DSLs, the tool comparison is given in next section.

There are many advantages of text-based modelling over graphical modelling for the user of DSL: e.g. it takes more space for graphical models to represent some information which is time consuming, writing and printing text is easy while for graphical models the size of graph can exceed the size of paper. During development process sometimes things can be described more efficiently by using text instead of drawing models like conditions and actions. Formatting text is easier and results of automatic algorithms are of good quality. Writing, reading, modifying text does not need any specific platform and can be done almost in every text editor. No additional tools or plug-ins are required. Version control systems are very important today during software development process like CVS [63] and SVN [64] which are text based and can be used for text based models [11].

Text-based models also have some disadvantages like graphics are more intuitive to give first orientation which is slightly compensated by text-based models by giving outline of code in the form of list or tree. Simulation and animation is more easy using graphics [11].

From a DSL's programmer point of view text-based models are advantageous too: A textual language can be written in any text editor and if auto-completion and syntax highlighting is required that can be done in any editing environment. Tools like MontiCore [65], ASF + SDF [66], TCS [67] and Xtext [31] support effortful but efficient way of creating text-based languages although MetaEdit+ [68] gives a simplified way of creating graphical language. Tools like parser can be easily developed by using ANTLR or DSL-Definition framework MontiCore which allows development of internal representation of abstract syntax according to the given textual model. Defining rules is much easier with the

textual models. Some languages are extension of a programming language like ArchJava [69] or LINQ [70] to improve the usability of programming language. The composition of modelling language which enables re-use of existing languages is much easier with textual languages [11].

The advantages of text-based modelling is further extended by [13]: textual artefacts integrated with existing tooling template, it is simple to update a textual model by using search and find technique and text-based DSLs are more appreciated because "*Real Developers don't draw pictures" [13].*

Since the arguments given above are more favourable towards text-based DSL, the decision is to create a textual DSL for this project.

2.5 Tool comparison

Although a decision has been taken to create a textual DSL, a comparison between tools to create graphical DSL is also given in the following section to give the reader an overview of these too.

2.5.1 Comparison of MSDSL tools and Eclipse modelling plug-ins Framework

A comparison is given by [14] between Microsoft DSL Tools (MSDSL) [72] and Eclipse modelling Framework (EMF) [71] on the basis of developing model-based languages i.e. Graphical DSL. An experiment was conducted with two groups of 48 undergraduate computer science students. One group was given MSDSL tools to develop a DSL including code generator and other Eclipse Modelling plug-ins. Students of each group did not know the features of tool using by the other group. They developed research questions in five categories which were Metamodelling, Graphical Editor, Code Generator, Satisfaction, and General Questions. On the basis of answers given by students to these research questions, comparison was formulated.

The main differences they presented were MSDSL Tools provide proprietary notation and graphical environment to build metamodel whereas EMF uses

Ecore which provides a complete metamodelling and model management environment. MSDSL Tools provide XML proprietary format where EMF supports XMI or user defined XML-schema format for the serialization of models. Eclipse provides Graphical Modelling Framework (GMF) which is more comprehensive graphical editor than that of provided by MSDSL tools. MSDSL tools lack support for model-to-model transformation and Eclipse provides plug-ins for such transformations. With regards to model-to-text transformations, MSDSL tools provide a primitive template language which enables the injection of C# or VB on the other hand Eclipse provides Java-based template languages [14].

The results obtained from experiment were: Ecore and EMF are easier to understand than proprietary notation provided by MSDSL tools. Graphical editor provided by both are difficult to use and generate incomplete graphical modellers. Using Eclipse users accepted to generate code with it while MSDSL tools users found it difficult and preferred some other language than the template language. Eclipse users were more satisfied than MSDSL tool users and they think Eclipse Modelling plug-ins are more mature and robust. Moreover, MSDSL tools are vendor dependent (Microsoft) without any support to Object Management Group (OMG) standards [14].

More or less same comparison is given by [15] but it included Xactium's XMF-MOSAIC [73] as well in his comparison. Microsoft DSL tools support more graphical DSL than textual one, but in the form of embedded DSL only which will be extension of languages like C# or VB [16]

2.5.2 A Comparison of Tool Support for Textual Domain-Specific Languages

A comparison between tools that support textual Domain specific languages is given by [17]. These tools included Xtext, Meta Programming Systems (MPS) [74], Monticore and IDE Meta-Tooling Platform (IMP). The criteria of

comparison were language, transformation and tool support. All tools represent concrete syntax as text but MPS stores model as XML document and present it as text in editor. Xtext and Monticore use single source to define concrete and abstract syntax, MPS uses abstract syntax in the form of concept which then defined as concrete syntax whereas IMP defines concrete syntax only which derives abstract syntax automatically. Xtext provides good transformation support by early error detection and code completion support. IMP has no support for built-in transformation. All tools except IMP support model-to-text mapping however MPS requires mode-to-model transformation prior to it. Monticore provides model-to-model mapping as well. All tools except MPS generate language workbench based on Eclipse platform. Xtext and MPS both give a comprehensive template support using constraint language with code completion and validation while typing but for the current study choice will be Xtext because MPS editor is cell based instead of free text and in MPS model-to-text transformation needs model-to-model mapping first.

2.6 Model Based Testing in Automotive Systems

Bringmann and Kramer [6] presented a model-based testing approach in automotive systems. They introduce a testing tool TPT (Time Partition Testing) which is based on graphical test models. There are three objectives of TPT [6]

1. Supporting test modelling technique to allow systematic selection of test cases
2. Providing representation of test cases for model-based automotive development in more precise and portable form
3. Providing an infrastructure for automated test execution and assessment even for real time environments

Test cases are modelled graphically, compiled into byte code and executed by a dedicated virtual machine. Assessment script which contains expected results also created for test case during compile time. Test assessment is done by evaluating recorded test data with the assessment script. TPT uses Python as

scripting language and Python interpreter is used as runtime engine. TPT test cases are reusable at different test platforms like MiL (Model in Loop), SiL (Software in Loop) and HiL (Hardware in Loop) [6].

Another approach of model-based testing is given by Siegl et al. [7]. They introduced Timed Usage Model (TUM) which is based on Markov Chain Usage Models (MCUM). It provides the possibility to describe timing and data dependencies of SUT (System under Test). Model supports test planning and generation. The applied models allow systematic generation of test cases and assessment with respect to coverage of requirements.

2.7 Example Implementations of DSL based Systems

Apart from commonly used DSLs like regular expressions, SQL and CSS there are other DSLs produced by people who needed them in a particular domain like embedded systems, mathematics, Smart Grids, electronics, bioinformatics etc. Some of them are illustrated in this section.

2.7.1 A DSL for Simulation Composition

Schutte [8] defines an approach to describe formal scenarios and simulation specification. A DSL in combination with a simulation framework is able to interpret the description and allows the automatic composition of the simulations. This DSL with the simulations framework is built for a GridSurfer project that analyses the impact of electrical vehicles on the distribution grid. The domain of this project is SmartGrids. He used Xtext and Xpand for the development of this DSL and Model-Integrated Computing (MIC) because it allows people without in-depth knowledge of simulation framework to create domain specific modelling layer [8]

The DSL is of external kind with own grammar and ultimately syntax. An interesting aspect in this project is that the scenario specification generated is loosely coupled with the simulation framework. In case of any change in simulation framework being made the Xpand generator will need to be adapted

instead of changing a large number of scenario specifications. Figure 2.3 shows the whole process

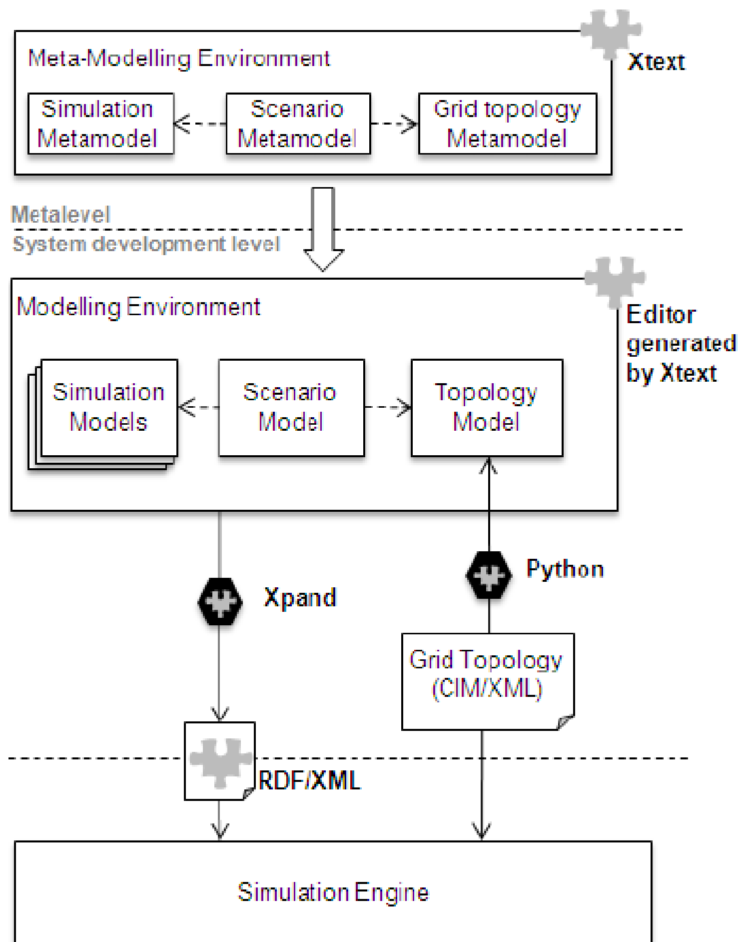

Figure 2.3 ***MIC-Based approach for SmartGrid Simulation [8]***

2.7.2 CAST: Automated Software Tests for Embedded Systems

Wahler [9] introduces CAST (Computer-Aided Specification and Testing) an approach to tests automation in embedded systems. CAST consists of three parts, a DSL named as TESLA (**TE**st **S**pecification **LA**nguage) which allows specifying test cases using familiar syntax, a test execution engine which allows executing tests either automatically or with human interaction and an interface which is a form of connection between engine and embedded systems. He used Eclipse IDE and Xtext plugin to create the execution engine and SCALA to write interpreter for TESLA. The architecture of CAST is shown in figure 2.4

This is an external type of DSL with grammar to describe syntax of the language. Components of CAST are loosely coupled; if Hardware builder is not required and other interface is needed to be used CAST can still be used by replacing this with any other interface which supports OPC and updating Device Interface. There are some test cases which need physical interaction and thus resist automation in which case tests cannot be run in batch and left for overnight or weekend. Test coverage is not part of generated test report by CAST at the moment. CAST used Scala interpreter for language generation which means no code is generating. Some aspects of DSL are platform specific like download and actions commands. This DSL cannot be used for the current study because of the fact it is using Scala interpreter for interpreting which will be needed if DSL applies to other embedded system and secondly DSL elements are specific to testing systems at Asea Brown Boveri (ABB) Ltd. [50]. The DSL does not support user with facilities such as error detection, scoping and content assistance. CAST architecture is shown in Figure 2.4

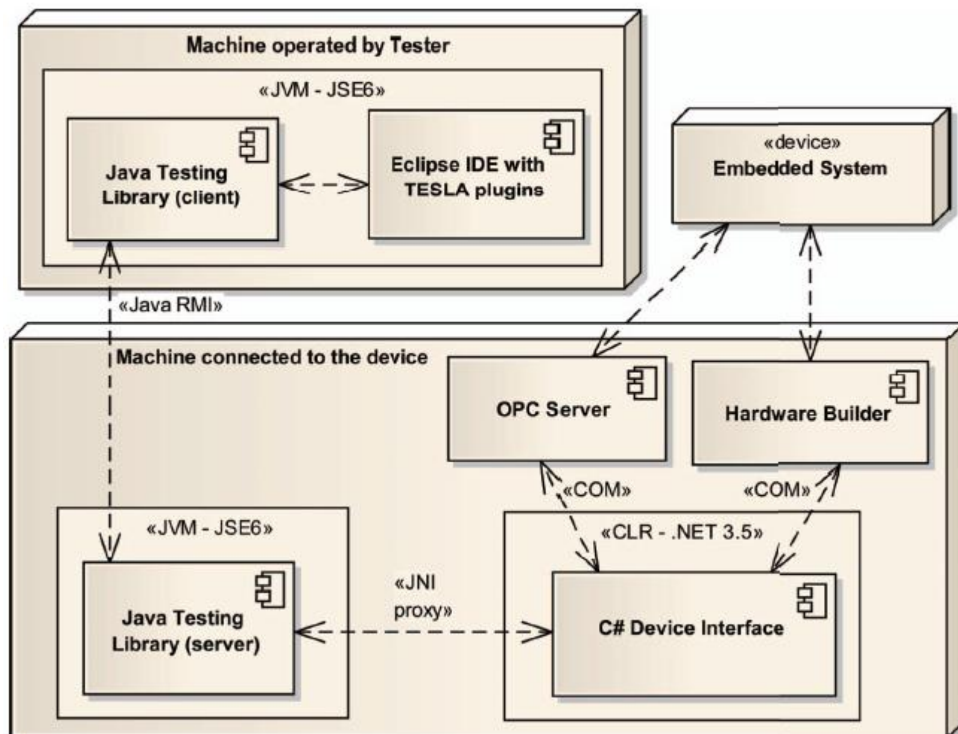

Figure 2.4 ***Architecture of CAST [9]***

2.7.3 Habitation: A DSL for Home Automation

Home automation uses MDE (Model-Driven Engineering) approach in reactive systems. It offers management of energy, security and communications through interaction with the environment. Habitation [10] (Development of home automation applications using a model-driven approach) combines DSL with MDE to handle the life cycle of home automation system design. The authors used Eclipse Graphical Modelling Framework (GMF) to develop the DSL which consists of three parts

1. A drawing area where graphic models for catalogue and applications are made
2. A graphic palette contains elements which can be dragged on drawing area
3. An area where properties like attributes and parameters are displayed and can be modified for an element

The author used Java Emitter Template tool (JET) for model to text transformations. To generate code developer needs a specific platform which must be supported by international standards and provide tools for programming the devices, in this case these requirements are fulfilled by KNX [79] and LonWorks [78]. So the environment is coupled for a moment and they are working on completing code generation implementation for commercial tool (ETS) [10].

2.7.4 A Domain-Specific Language for Ubiquitous Healthcare

Aspect Language for Pervasive Healthcare (ALPH) is a domain-specific language in ubiquitous healthcare domain. Ubiquitous healthcare means presence of healthcare everywhere. It is an emerging technology that consists of large number of environmental and patient sensors and actuators to improve patients' mental and physical condition. It provides a domain-specific aspect language (DSAL) which contains extensible high-level constructs. Use of any construct by a programmer initiates implementation of ubiquitous health-care concern from the library. It is a declarative language implemented as a pre-

processor to an existing aspect language AspectJ. The main entities of this domain are mobility, context awareness and infrastructure. The ALPH program is compiled by an ALPH compiler into an aspect language. The final executable ubiquitous health care application is composed of aspects which contain ubiquitous healthcare behaviour from the library which are merged into the base application using the aspect language weaver. ALPH is extensible in three ways: the language and compiler can be extended by extending language model definition and semantics; the aspect library can be extended by adding new constructs with the help of code and construct's parameterisation which supports customize behaviour. A formal definition of translating ALPH program into concrete base language (GPL) is defined in compiler generator which allows developers to provide definitions to translate ALPH program into multiple GPLs [24].

To evaluate the new language author conducted an experiment by implementing an application named as MedHCP based on a scenario from ubiquitous healthcare domain using a (GPL) Java as well as ALPH language. This application was deployed on the Motion C5, the mobile clinical assistant created by (DHG) at Intel Health. The results obtained showed reduction of coupling by 33-75%, dependencies on external modules by 40% and application size by 25%. ALPH language is significantly expressive and constructs can fulfil 50% of domain specific requirements by 20% of action terms from domain [24].

2.7.5 Domain Specific language for Cellular Interactions

CellSys is a DSL embedded in Haskell (GPL) specific to bioinformatics domain, is used to model life cycle of microorganisms like bacteria. The objective of this DSL is to allow biologist to create a model which can describe complex interactions between tissues and organisms with abstraction and accuracy, visualize organism's development by executing these models, help language user to improve understanding of organism's behaviour and structure by suggesting refinements and compare cellular system's models between

different organisms or stages of development of an organism. Each CellSys program has some actions to describe its behaviour with respect to itself and environment. This DSL bridges the gap between a biologist and computer scientist [25].

2.7.6 A DSL in Embedded Systems

DevC [26] is a DSL in the domain of embedded systems which allows concurrent development of device controller simulation model and device driver code by specifying different characteristics like services, constraints, sequence of commands, mechanism of communication between controller and processor and interface with the operating systems. The syntax of DevC is similar to SystemC [81] and ArchC [80]. Currently, the language is used to develop USB controller and graphic display [26].

2.7.7 MobDSL

In application development for mobile devices industry there is no platform which can be used to build an application which can be deployed to multiple mobile platforms like Apple iPhone, Google Android and Microsoft Windows Mobile. MobDSL (Mobile DSL) is made for the mobile application development domain and address the problem explained. Currently there are two approaches to create applications in this industry: by using frameworks and mobile web application. The authors have done domain analysis by presenting two iPhone application case studies on Tour de France and Lyrical Genius for local SME. Tour de France application was to help support people in following the 2009 series of Tour de France. Lyrical Genius was a game that consists of quiz questions relating to different lyrics in the songs. They identified domain features like limited screen size; layout control in XML; GUI element containership; event driven application; hardware features like camera, accelerometer, GPS, microphone and close range sensors; concurrency by using threading; object oriented language use (like C++, Java); and state machine transitional behaviour of mobile devices. The calculus for mobile

applications language is based on lambda-calculus extended with the widgets for managing mobile application components. Author describes how features required by mobile application development described above can be supported by the mobile calculus. Authors proposed architecture to implement this DSL for making platform independent applications consist of three tiers: the application written and compiled using DSL; DSL specific engine and libraries; and running platform which can be Java, C#, Android, or iOS. The virtual machine (VM) for target platform contains two parts: platform libraries (MobLib) which contains platform API calls, engine which will run the compiled code and make the appropriate platform calls. Benefits of the DSL with VM for different platforms include: avoidance of application installation source lock-in which gives security to the users and small application size because VM contains all the functionality which makes downloading easy as well [27].

2.7.8 SLCO

Simple Language of Communicating objects (SLCO) is designed and implemented by [29] in the domain of distributed communicating systems. The DSL is to model the structure and behaviour of the system consists of concurrent communicating objects. Models specified using this DSL can be transformed into models for simulation, verification and execution. It provides constructs for system objects that operate in parallel and communicate with each other. The authors used Eclipse Modelling framework to describe SLCO models and Xtext for defining concrete syntax with a textual SLCO editor. All transformations used to bridge the gaps in platform are implemented using Xtend model transformation techniques.

In this chapter approaches applied by people in area of automated testing are observed such as JUnit, record/playback testing, model based test automation. A comparison between external and internal DSL, textual and graphical DSL

and tools to build these forms of DSL are discussed. Some examples of DSLs are also given which people have used in their domains to solve the particular problems. For this project decision is to build a textual DSL using Eclipse framework to automate the test specification component of ViBATA.

3 Theory and Technologies

In this chapter detail of technologies used in this project is provided including an introduction to software testing, and the stages involved in the DSL development lifecycle, after domain analysis which was described in 2.2.

3.1 Technologies used in the Project

The technologies used in this project are Eclipse Xtext, Xtend, Java, and Simple framework. In chapters 4 and 5 the use of these technologies in DSL's development stages and analysis is documented. In this section an introduction to these technologies is illustrated which will help understanding the next chapters.

3.1.1 Eclipse Xtext

Eclipse [41] is open source software for individuals and organisations to build open development platform projects. These projects are comprised of extensible frameworks, tools and runtime for building, deploying and managing software across the lifecycle. Eclipse was originally created by IBM in November 2001 and supported by a consortium of software vendors [41].

Xtext [31] is part of openArchitectureWare (oAW) which is part of Eclipse. Xtext is a framework which allows creating external textual DSL by using Xtext's EBNF based grammar language [42]. It defines several application programming interfaces (APIs) to describe different aspects of language such as scoping API defines which elements are referable by a certain reference (section 4.4.5). It uses Dependency Injection (DI) framework, Google Guice [38], for integrating all of language components. That means if one component needs functionality of another component, it declares the dependency by providing @Inject annotation as shown in Figure 3.1 *Dependency Injection*. This line means that the code generator is using interface IQualifiedNameProvider which provides the functionality to define the full name of the element in AST.

```
class CATTGenerator implements IGenerator {
    @Inject extension IQualifiedNameProvider
```

Figure 3.1 ***Dependency Injection***

Xtext provides a language development framework and one can create his own language by creating grammar composed of number of rules. A rule consists of number of symbols or tokens which can be either a reference to another rule in the same grammar or super grammar from which new grammar is inherited i.e. Terminals or Xbase. A rule results in meta type, the symbols (token) used in the rule are mapped to properties of that type sometimes referred as features or attributes (3.2.2 for details). In an Xtext file, there is a generator declaration which generates artefacts such as a parser that can read textual syntax and returns an Eclipse modelling framework (EMF) based metamodel: abstract syntax tree (AST). AST is in-memory object graphs which are instances of EMF Ecore models. Ecore model consist of an EPackage containing EClasses, EDataTypes, and EEnums and defines the structure of instantiated objects. It also generates full-featured Eclipse Text Editor which provides syntax highlighting, code completion, a configurable outline view and validation for the given syntax. Java Runtime Environment (JRE) is necessary to install to work with Eclipse project. It provides full implementation of a language running on Java Virtual Machine (JVM). The compiler components of the language such as parser, abstract syntax tree (AST), serializer and code formatter, scoping framework and linking, compiler checks and validation, code generator or interpreter are based on (EMF). Xtext is used in this project to build the syntax of the DSL. Rules are formulated using Xtext in order to build the language syntax (sections 3.3.3 and 4.4.2). [31], [38], [42]

3.1.2 Xtend

Xtend [34] is programming language shipped with Eclipse which translates to Java source code. Syntactically and semantically it is compatible with Java programming language and provides interoperability but enhances on many aspects such as

- It removes syntactical noise: no need of semicolons and no empty parenthesis

- It extends existing Java APIs by providing extension methods and lambda expressions. For example method toFirstUpper(String s) is defined in StringExtensions library and takes string as an argument. But instead of passing string argument it can be used with string as if this method is defined for a string Figure 3.2

```
"hello".toFirstUpper() // calls StringExtensions.toFirstUpper("hello")
```

Figure 3.2*: Extension methods in Xtend*

Other features of Xtend are

- It is easy to learn for Java users because it uses existing Java concepts. It uses Java type system unlike Scala which is JVM language but implements a new type system.
- Xtend does not have statements instead everything is defined in expression which provide return value. Expressions are more concise, expressive and readable. For example use of try catch block on the right side of an assignment
- It provides great user experience by provision of better tool support in the form of Eclipse-based IDE integrated with Java Development Tools (JDT). Features such as call-hierarchies, rename refactoring, and debugging enhances IDE support. [34]

Template Expressions

Another powerful aspect of Xtend is the provision of 'Template Expressions' which allow readable string concatenation surrounded by triple quotes ('''). Template Expressions allow code generation in any GPL such as current project is using template expressions to generate code in Java. A template expression is composed of one or more lines. The expression to evaluate is placed inside template expression defined between guillemets «expression»

If and Switch conditional statements can be used between guillemets which have their own syntax. [34]

3.1.3 Simple Framework

Simple [43] is a configuration framework for Java and is used to perform XML serialization. In this project simple framework is used to create XML output from Java code generated by program in DSL. To define each element in XML file this framework used annotation for class and its properties. For example if an object is root element in XML file, @Root annotation needs to define above this object. This framework exposes two classes Serializer class which is an instance of Persister class to serialize an object in Java. A java.io.File object is created with name and location information to create XML file with specified name on specified location. The write() method of the Persister class performs serialization by taking Java object and file location as arguments and serialize object on the file location. For details and example see sections 4.5.2 and 4.5.3. [43]

3.2 Development Stages of DSL

As described earlier the aim of this project is to create an external type of domain-specific language for domain of embedded systems in general and automotive in particular to automate test case definition. To create this DSL we are using Eclipse IDE. There are stages involved in the development of DSL which are briefly described in this section. Implemental details of these stages for current project are described in sections 4.4 and 4.5.

3.2.1 Domain Analysis and DSL Behaviour

In the literature review, an introduction to first stage of DSL development 'Domain Analysis' and the definition of domain elements is defined in section 2.4. Implementation of domain analysis for current project is given in 4.4.1

Describing 'DSL Behaviour' means to investigate how DSL elements interact with each other to exhibit behaviour which is complete and correct as specified by domain experts. During this stage behaviour of single element or group of related elements is specified. The behaviour can be explained with the help of control flow models, detailed behavioural models that are used in model-driven generation or precise textual specification. The DSL behaviour specification

also referred to as dynamic semantics. It also defines how DSL element interacts at runtime [22]. For this project, behaviour of DSL as whole and its elements is defined in precise textual specifications in section 4.4 and 5.2.

3.2.2 Define Concrete Syntax and Rules (Grammar)

Concrete syntax of the DSL represents the user interface of the language. It is suggested by [22] to perform this activity of defining concrete syntax in parallel with defining DSL behaviour because these activities can have influence on each other especially in case of embedded DSLs because syntax and behaviour of host language will have effect on DSL. A concrete syntax can be a graphical or textual. Implementation of concrete syntax includes implementing GUI editor, grammar and a parser, or extending an interpreter. Defining concrete syntax starts from defining graphical or textual symbols or tokens for each rule. While defining rules standard programming language conventions should be taken care of such as how to define comments, strings and numbers which in this case defined in super grammar Terminals (org.eclipse.xtest.common.Terminals) [31]. Next is to define the composition rules of the syntax which explains how rules can be composed to make legal expressions in DSL. While creating these rules it is always useful to ask domain experts questions about ease of using syntax such as what keywords and expression formalism in the language is easy to use for them [22].
First rule of any grammar is used as an entry point like in the Figure 3.3 [31]

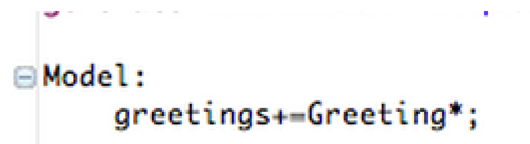

Figure 3.3*: Starter Rule [31]*

As this project is going to use Xtext for defining the grammar a snapshot of what grammar means and looks like is shown in Figure 3.4. In this figure grammar of the language is shown.

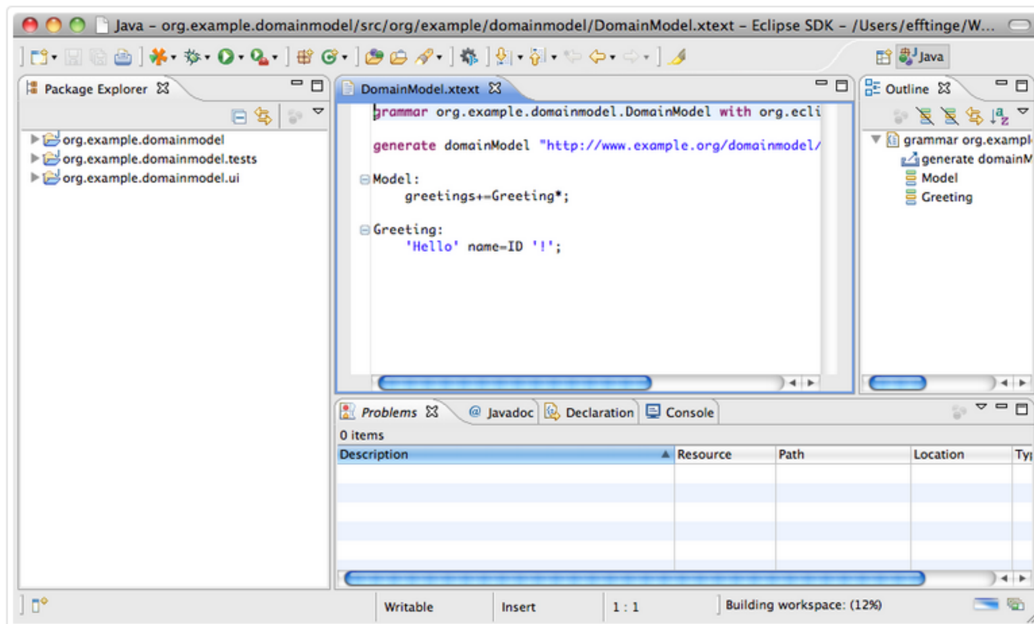

Figure 3.4*: Defining Grammar [31]*

In Figure 3.3 rule is 'Model', property is 'greetings' and 'Greeting' (token) is call to another rule defined in the same grammar . There are two kinds of assignments in defining rules. The '=' sign assigns the value returned from the token to the property (the property will have the type of token) which in Figure 3.5 is 'name' and '+=' signs add the value to the property (the property will have the type List<tokenType>) [42]. The rule in Figure 3.3 means that Model contains arbitrary (*) number of Greeting which will be added (+=) to feature greetings. Next rule defines Greeting in Figure 3.5

```
Greeting:
    'Hello' name=ID '!';
```

Figure 3.5*: Defining anther rule [31]*

This rule means Greeting starts with a keyword 'Hello' followed by an identifier which is parsed by a rule called ID. The rule ID is defined in the super grammar

org.eclipse.xtext.common.Terminals and value returned by the call to ID is assigned to feature name, followed by a keyword '!' [31].

3.2.3 Development of Language Artefacts

When Xtext project is created it consists of three sub projects. One is to define the language, second to define the tests for the language and third for user interface of the language. This thesis concentrates more on language project, test project is out of scope and a little customization is done in user interface project in section 4.4.6. Language project consist of a folder named as src which contains file with .xtext extension to define rules for the language. There are two other folders one is src-gen folder and other is xtend-gen folder which are empty in the beginning. Once rules are defined in the .xtext file language infrastructure will need to be generated. This would accomplish by right clicking .xtext file and choose Run-As ->Generate Xtext Artefacts. This step will populate src-gen folder with sub-projects for Validation, Scoping, Serializer etc and xtend-gen with generator project with Xtend file. Running the language project as new Eclipse application will allow testing the language in the editor. [31]

3.2.4 Model Constraint

After grammar is defined, the generated DSL editor can detect syntax errors in the program code but there is still a possibility of defining illegal models like several datatype definitions with the same name (Figure 3.6). To overcome such situations constraints are needed to define in *Check* file (Figure 3.7). *Check* language was provided by openArchitectureWare (Eclipse) to define constraint to ensure the validity of the models [42].. Syntax of Check Language is similar to Object Constraint Language (OCL). *A **constraint** starts with a keyword 'context' followed by name of type for which this constraint must hold* [28]. The error is highlighted (Figure 3.6) by defining same datatype.

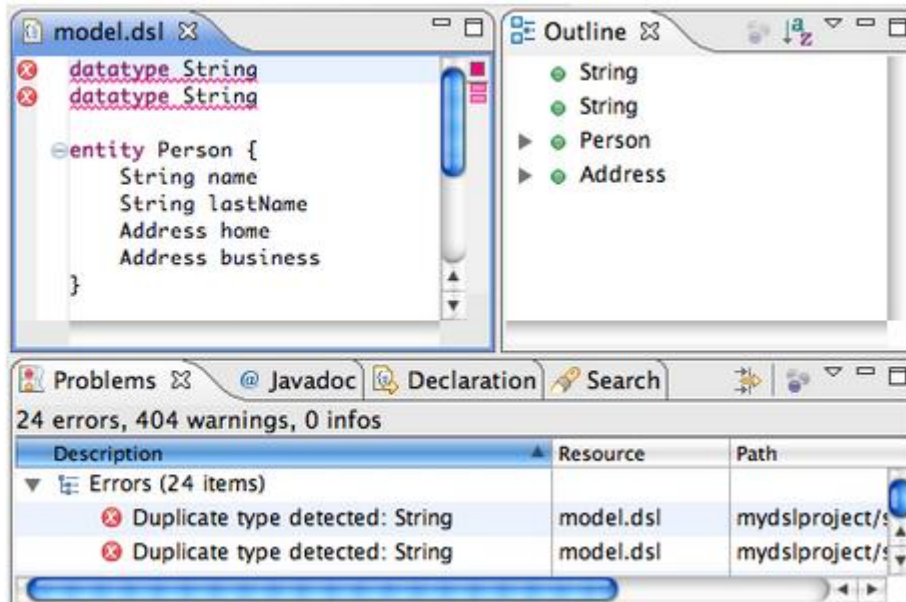

Figure 3.6*: Error in the output of grammar [28]*

```
context Type ERROR "Duplicate type detected: " + this.name :
        allElements()❶.typeSelect(Type)❷.select(e|e.name == this.name).size ==1;
```

Figure 3.7*: Model Constraint [28]*

The constraint line above in Figure 3.7 means each model may have only one dataType with same name. Check language was introduced in first release of Eclipse. In latest release of Eclipse Juno on creating artefacts a validation package is generated within src folder of the project which contains a .java validator class inherited from AbstractJavaValidator class to define the validation for model. A @Check annotation is used above each validation function defined in this class (details of implementation in section 4.4.4) [31].

3.2.5 Integrating DSL with target Platform

The last two activities of DSL development are interdependent. In these activities DSL artefacts are mapped with the target platform and code is generated according to it. There are two parts of the target platform: generic platform artefacts like Enterprise JavaBeans (EJB) or Microsoft.Net and DSL-

specific platform artefacts. DSL artefacts like language model, behaviour definition, and concrete syntax must be mapped to the target platform. The first activity is to decide which existing features of the platform can be used with artefacts, sometimes because of lack of feature support platform needs to be extended (Figure 3.9) [22]. Here in *figure 3.8* our target platform is Eclipse which will show the output of the grammar we created above and will generate the code in target language Java. The only extension is done in this platform is addition of Simple framework defined in section 3.1.3. It is needed to add the framework's .jar file into projects JRE System Library folder (Figure 3.8)

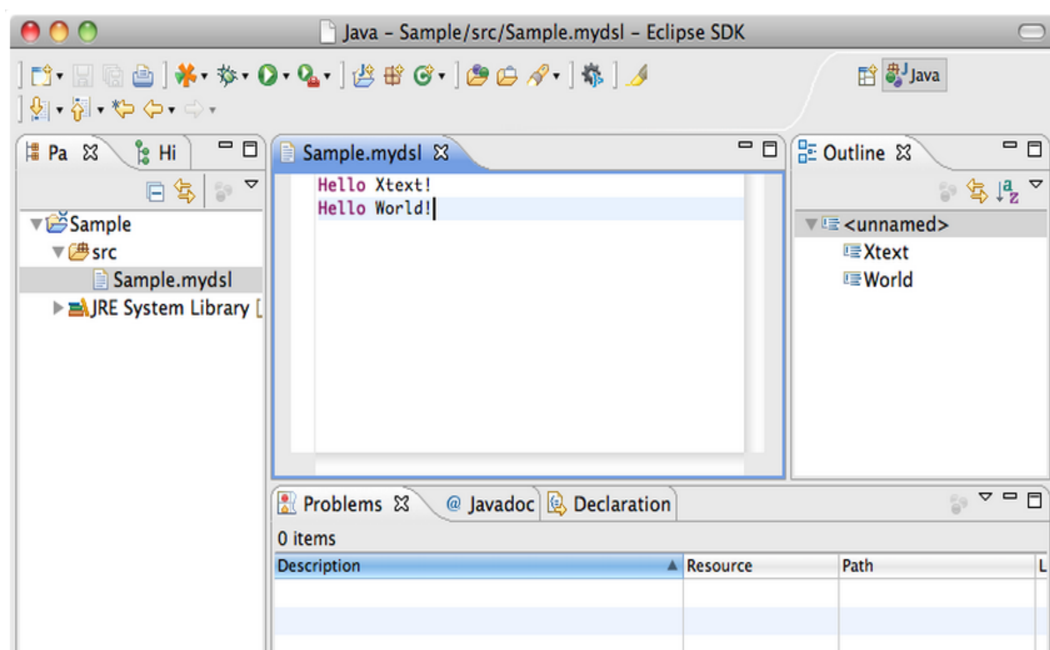

Figure 3.8*: Output of Grammar [31]*

3.2.6 DSL to platform Transformation

In this stage DSL-to-platform transformation is performed which is also referred as Code Generation. According to Fowler [1] there are two styles of code generation one is Model Ignorant Generation and the other is Model Aware Generation and two kinds of processes of code generation i.e. Transformer Generation and Template Generation. The difference between the two

transformations is the former uses Semantic Model directly to create the output while later uses an embedment helper. So generation using Xtend is Model Aware Generation because it uses process of Transformer Generation. Details of both styles and processes are given in his book. The transformation is a straight forward activity in case of embedded DSLs but in case of external DSLs transformation rules are defined. These transformation rules convert the DSL language models to the platform, the generator in openArchitectureWare (oAW) convert concrete syntax into EMF models and its transformation language Xpand, which is now replaced with Xtend, allows defining transformation rules which convert EMF model to the target platform. At this stage, integration testing can be performed to check if all artefacts are working properly. Unit testing should be done throughout the process and finally user acceptance test for the concrete syntax should be performed. If language is completed then language engineering process is over and DSL is ready to use [22]. The whole procedure of integrating DSL with target platform and transformation is shown in the Figure 3.9

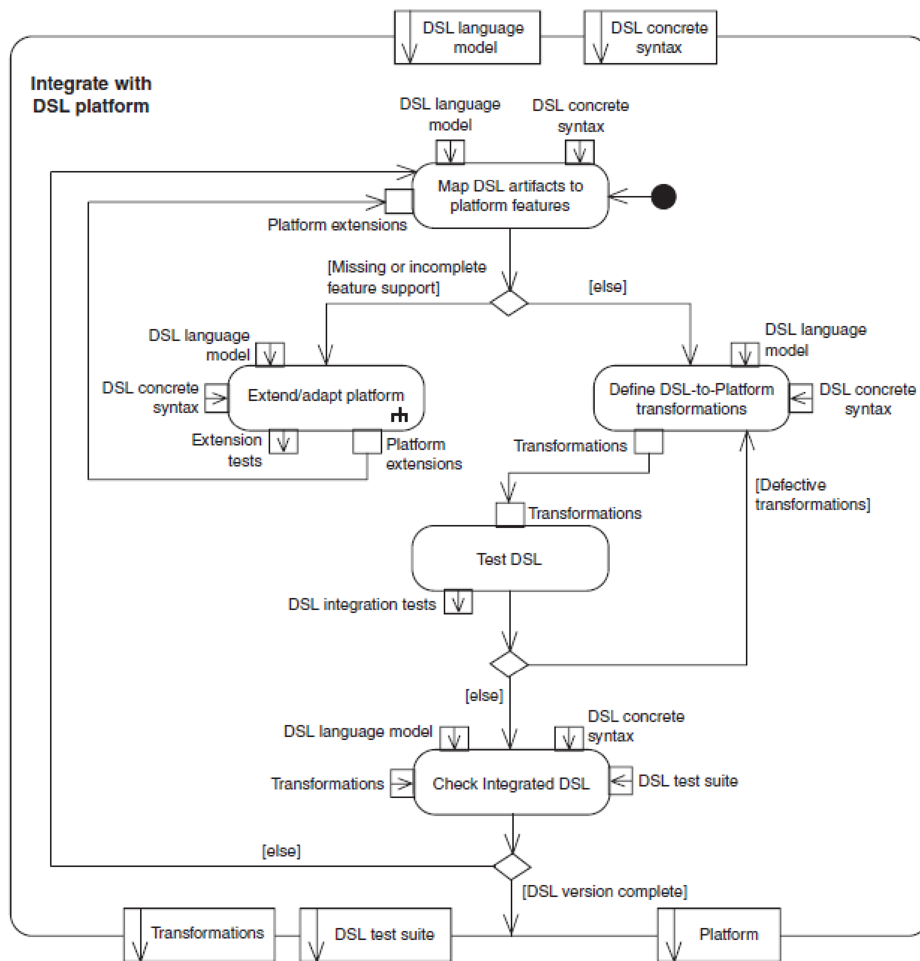

Figure 3.9*: DSL integration with target platform and transformation [22]*

3.3 Software Testing

Software testing is the most important part of software development life cycle to bring the quality and completeness. The software developed for JLR (ViBATA) is discussed in section 4.2, is developed to provide a facility to test the system automatically. This section is to provide an overview of software testing.

The most precise definition of software testing is given by [3]

"***Testing is the process of executing a program with the intent of finding errors.***"

Testing is an important part of software development cycle and is part of all software development models e.g. Waterfall model. People involve in software development have intuitive view of testing and its purpose, most common reasons of testing are: Ensuring software corresponds to its specification; finding defects in the software; confirming system works properly; understanding how far software can be pushed before it fails and the risks involved in releasing the software to the users [4].

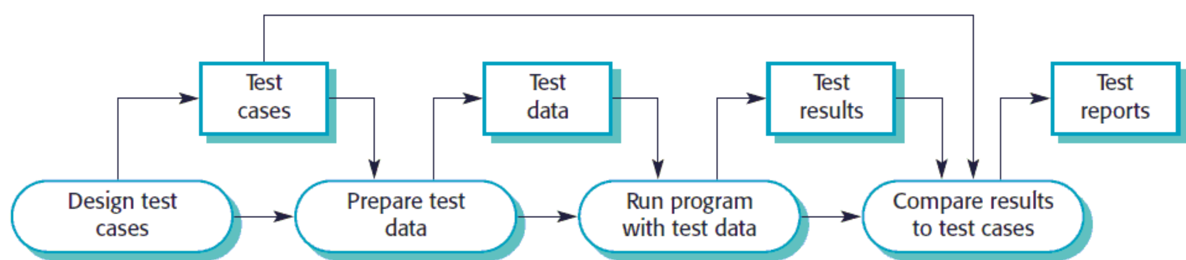

Figure 3.10*: A model of the software testing process [5]*

The software testing process shown in Figure 3.10*,* showing test cases which specify inputs to the test and expected outputs from the system along with a statement of what to test. Test Data is inputs used to test the system which can be generated automatically sometimes. The program runs with test data provided. Output of the test can only be predicted by people who understand the system and check the expected output with the actual output and decide whether test passed or failed [5]. This whole process is performed automatically in test automation software as discussed in section 2.1 of literature review.

This chapter gives an introduction to technologies used in this thesis such as Xtext to define Grammar, Xtend to generate code in Java and Simple framework to create output in XML. It also provides information about development stages of DSL which will be applied to build DSL for current study in Chapter 4.

4 Methodology

Visual Based Test Automation (ViBATA) is software built by Cranfield University to support automated testing in the automotive systems. This project gave an opportunity to identify, analyse and gather knowledge about the automotive domain which is most important and first step in the development cycle of DSL. This software has provided with an insight into automated testing in automotive and domain knowledge of automotive industry which was quite helpful to understand how the DSL should look like and what should constitute it. In this chapter details of previous testing procedures at JRL, ViBATA, and implementation stages of DSL in current study are given.

4.1 1st generation testing procedures - ControkDesk & Python scripts

This section provides an idea about the hardware used and previous testing procedures at JLR. Instrument Panel Cluster (IPC) consists of a LCD panel to display information. On the left of the panel a speedometer and right a tachometer graphic was located. The centre of the panel allowed the display of contextual information along with configuration of vehicle through a hierarchical menu system shown in Figure 4.1.

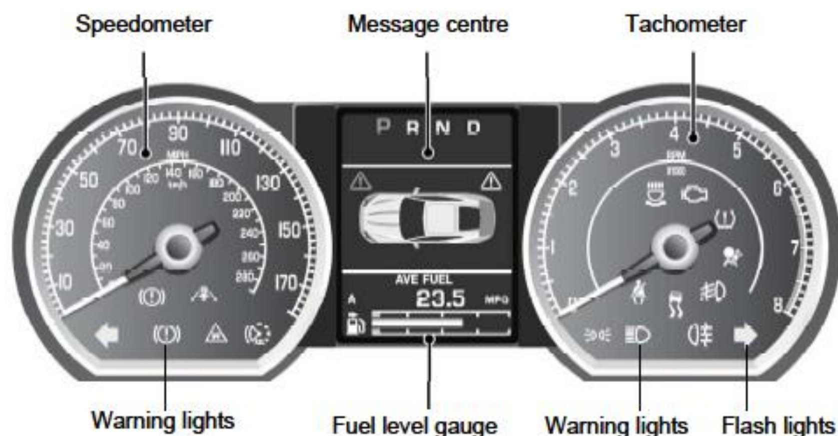

Figure 4.1: Instrument Panel Cluster [33]

Using a physical cursor pad located on the steering column this message centre display could navigated. To simulate the vehicle and form the hardware-in-the-loop testing environment a dSpace Autobox simulator was used to compile and

execute the car models. The simulator communicated with the IPC through CAN network connection. The camera used to monitor the output from IPC display panel was Cognex In-Sight camera system. The software provided with this camera installed on PC could store images for each test case and save it in .job file format inside camera's on board limited memory. Camera was communicating through an Ethernet connection. The manual testing system required the operator to identify the test cases with inputs and outputs from an Excel document and apply them to the SUT using the software ControlDesk supplied by dSpace Autobox. Using ControlDesk operator could create complex graphical representations to relate the values of CAN signals with visual displays of ECU on screen. In this way operator could follow the test case specification and instruct the values of inputs by activating the related graphical representation. This manual testing process was lengthy and complicated shown in Figure 4.2.

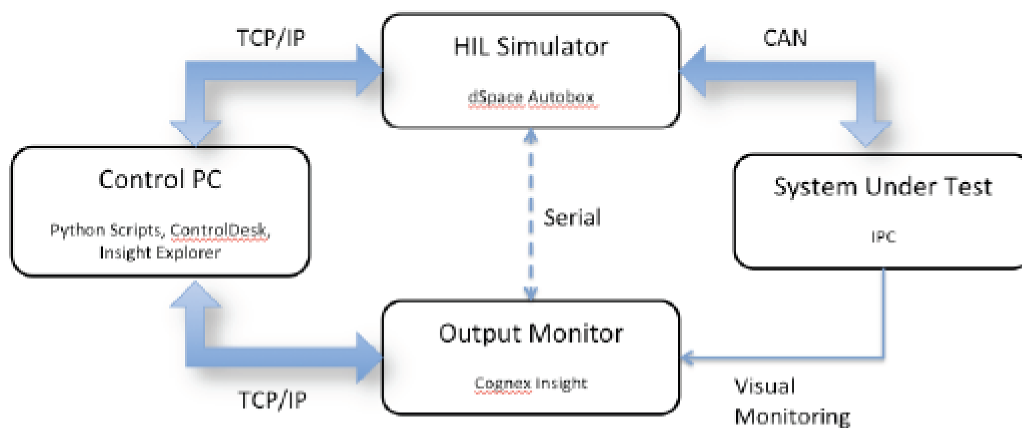

Figure 4.2*: Manual Testing system at Jaguar Land Rover [32]*

So the first level of test automation was to instruct inputs without using graphical components. To accomplish this semi-automated testing system was introduced. That semi-automated testing project used an application to generate python scripts and introduced a vision system to observe the output from IPC [33]. The testing system consisted of an application written in Python. This application could convert the test case specifications into the python scripts.

ControlDesk software provided the facility to execute the Python application within it. To make the process easier a template system was introduced to generate the python scripts. In this way a library of python scripts was created. This process still needed operator to locate and initiate the python script within ControlDesk software for a test case to send CAN signals to IPC. The drawbacks of this testing system were all the components involved in the system were tightly coupled for example camera was strongly linked with the simulation environment which provided the limited control of it. Also vision jobs could not be loaded into the camera's memory from the local storage but only those could be used present inside camera's limited memory. In case any of hardware changes the whole testing system will need to be implemented again from scratch. Secondly, change in version of IPC change slightly test specification and completely expected output which requires test to be rewritten to ensure correct result. This will need to manage the test specification separately from test execution which is addressed in current testing procedure [32]

4.2 2nd generation testing procedures - ViBATA

ViBATA is inherited from an earlier implementation of semi-automated testing procedure of IPC described in section 4.1. It eliminates the need of graphical interface in ControlDesk software and communicates directly with simulation hardware through plugin. The testing system is loosely coupled and introduces a plug-in architecture which means if any hardware changes a new plug-in can be written for that hardware only leaving rest of the system unchanged. The software is designed in a way to support automation of test execution on the HIL testing rig which provides flexibility of test reuse between different versions of IPC and reduces the dependence on specific test equipment. The software has four components: Test Specification Manager (TSM) ensures management and coordination of test cases after transferred from Excel sheet. This transference is still manual but provides an efficient way to perform it; Test Configuration Manager (TCM) ensures when test is executed correct output is selected for the version of IPC being studied; Test Execution Manager (TEM)

ensures tests are executed correctly on components in test environment; and Test Automation Core (TAC) ensures correct operation of test automation and communication between the different components [32]. The software architecture is shown in Figure 4.3

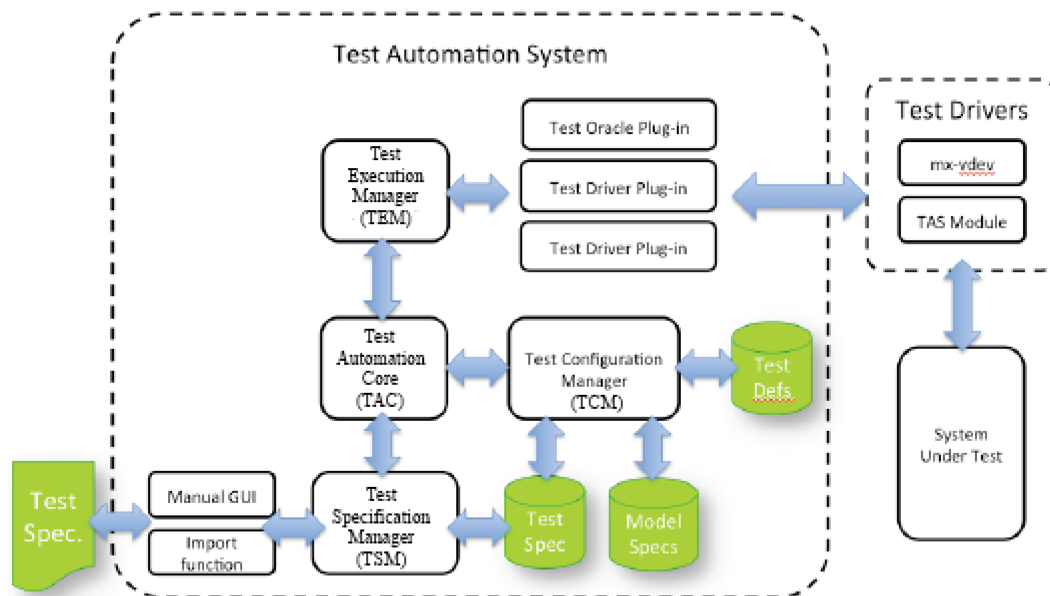

Figure 4.3*: Software architecture of ViBATA [32]*

Software Design

The software design used a modular approach to ensure that different components of the software work independently. This means that the component responsible for capturing output from the SUT works without direct connection with component responsible to specify and execute tests. For example a test case might check that if vehicle is in motion the seat sensor detects a passenger and the seatbelt sensor does not detect the belt, then a seatbelt warning image should be illuminated on the car dash board. In this case the test case success needs the illumination of the image only but oracle (section 2.1.1) function depends on the version of IPC because the warning image will be different for different versions of IPC. This will require creating a separate output for each version for the same or slightly changed test specification. This has accomplished by using decoupled architecture and

defining the test case in general way and storing version specific information into the database. The example is shown in Figure 4.4

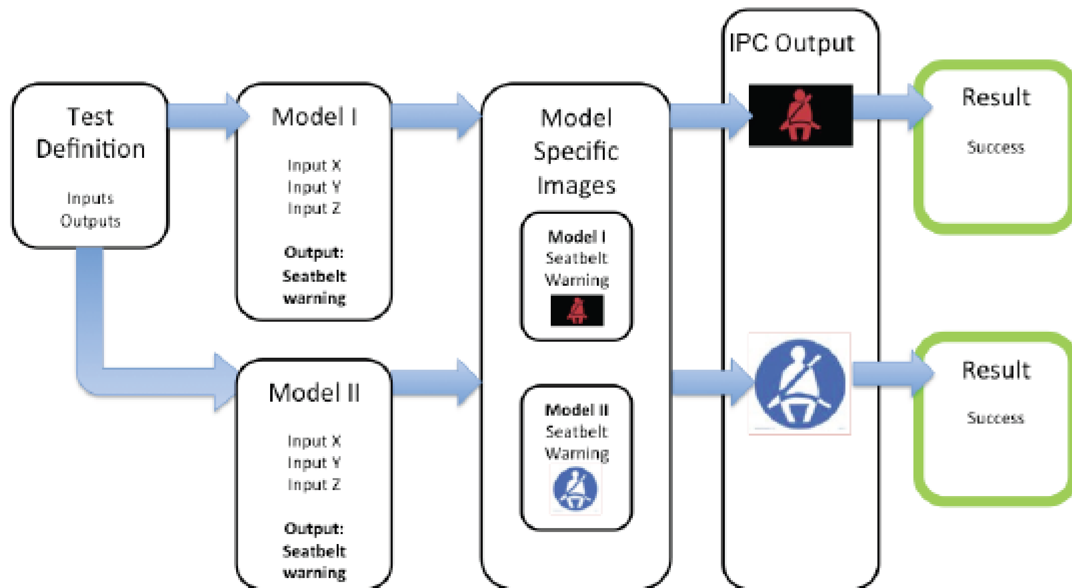

Figure 4.4***: Separating the oracle function from Test Management [32]***

In addition to the requirement of separating test case specification system from the test output capturing system the software supports the specific requirements for the current camera system known as Cognex Insight. The management functions of camera could be undertaken only by the interface exposed by the camera known as Insight-Explorer [40]. This led to the requirement of integration of these functions into the user interface of the software. For example many functions monitoring test output were based upon the pattern recognition operation exposed by the Insight-Explorer. The pattern recognition could be for a text or image output. So automation software had to provide the functionality for the operator to define job to monitor test output using these pattern recognition methods without the direct use of functionality defined in Insight-Explorer. To achieve this, a template system was introduced for each kind of pattern recognition method. The operator could choose a template for a particular test output according to its specification and supply the parameters for the template to generate a Cognex vision job in the background and the job could be saved on the local storage. The camera's functionality described in a generic way in a separate plugin which allowed the commands specific to the

camera system to be translated into the local language. The plugin architecture will allow replacing the camera system with any other vision system as long as it supplies the same functionality. In case of replacement the integration of new plugin for the new system will be required.

The decoupling of vision system from rest of the software led to idea of separating the rest of the system communicating with the other hardware components. TEM of the user interface is a point of contact to other hardware such as Simulink model. TEM is divided into two parts one is Test Driver (TD) and second is Hardware Driver (HD). TD is responsible for test execution by selecting the correct test specification for a version of IPC, send it to the hardware and receive the output from the camera system. It also defines the test workflow such as starting and stopping test, or putting delays between the different inputs of the test. HD is responsible for interpreting the instructions received by TD into the format understandable by hardware components. For example in case of dSpace Autobox, the HD consisted of Python interface which was exposed by Controldesk software and could search the required path for input signal in Simulink model and read/write its value. For Cognex camera HD consisted of telnet interface exposed by camera for communication. First level of decoupling is achieved by writing HD for each hardware contained in a separate plugin. In case of new hardware is introduced a new plugin for that hardware will be required to be written. Second level of decoupling required eliminating the test driver functionality from the TEM. In this case a third party application will be responsible of controlling the test cases such as Mx-vDev which could define its own relationship with hardware in test environment by exposing application programming interface (API) so that plug-in could be written [32].

Overview of User Interface

The software is made using Microsoft Windows Presentation Foundation with C# and Access database. This section illustrates an overview of user interface of ViBATA which gives an idea about the software and functionality of different software sections. An overview of software architecture is shown in Figure 4.5.

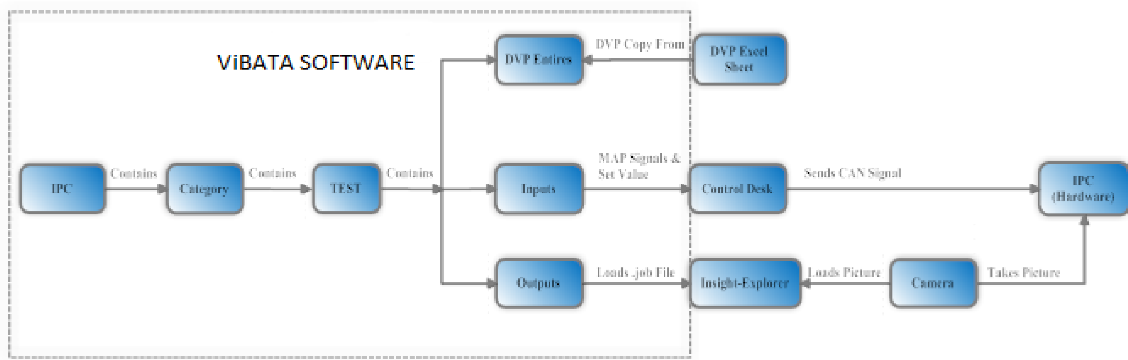

Figure 4.5*: Overview of ViBATA and Integration with other Components*

The user interface of the software is shown in Figure 4.6. The left hand side of the software shows information about all the IPCs, entered into the database, in hierarchical structure. On expanding IPC, categories listed are shown in each IPC and each category contains test cases which can be seen on expanding Categories. On clicking each test case right side of the software populates. Right side of the user interface contains five tabs relating to tests named as Test Definition which divides into Main Test and Pre-Requisite Test, Test Execution, History which further divides into Test History and Batch History, Batch Testing and Test Searching. The Test cases are comprised of more than one DVP Entries which consist of Input and Output Lines.

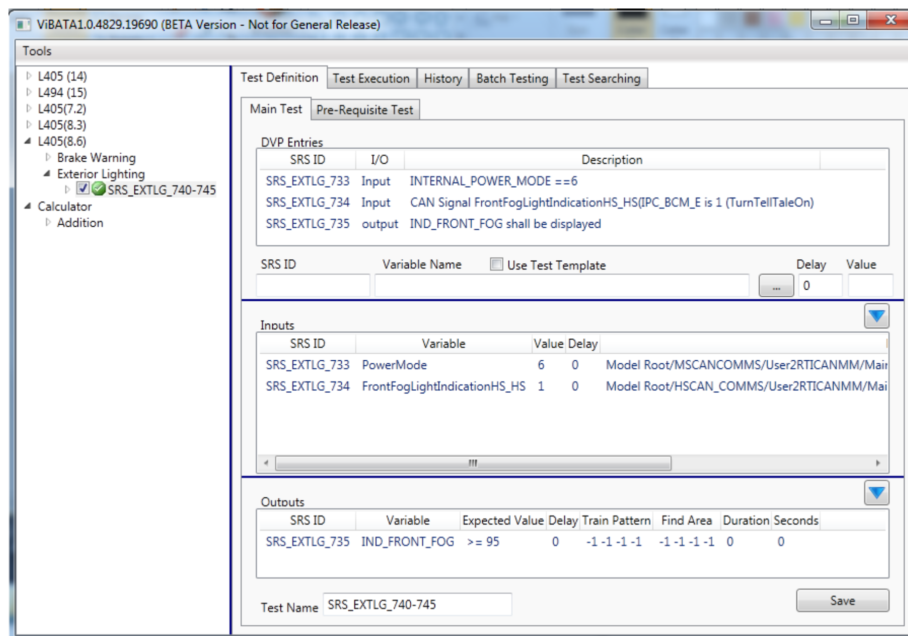

Figure 4.6*: User Interface of ViBATA*

Tests are identified in DVP (Figure 4.7) and are copied *(*Figure 4.8*)* and pasted into software's DVP Entries section from which input and output lines are identified and entered into respective sections.

SRS_EXTLG_721		PowerMode 6
SRS_EXTLG_722	Input(s):	CAN Signal FrontFogLightIndicationHS_HS(IPC_BCM_E) is 1 (TurnTellTaleOn)
SRS_EXTLG_723	Output(s):	IND_FRONT_FOG shall be displayed

Figure 4.7: ***Test case in Excel sheet (DVP Entries)***

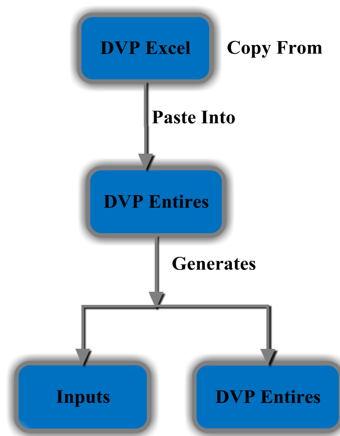

Figure 4.8*:* ***Copying and pasting DVP Lines from excel sheet into Software***

Input line(s) consists of value and path maps to CAN signal in .sdf file, and output line(s) consists of either a signal output *(*Figure 4.9*)* or a pattern output. A signal output returns a value from a CAN signal and pattern output matches the image/text stored into the system with the image/text captured by camera shown on the IPC in response to signals sent from the input lines.

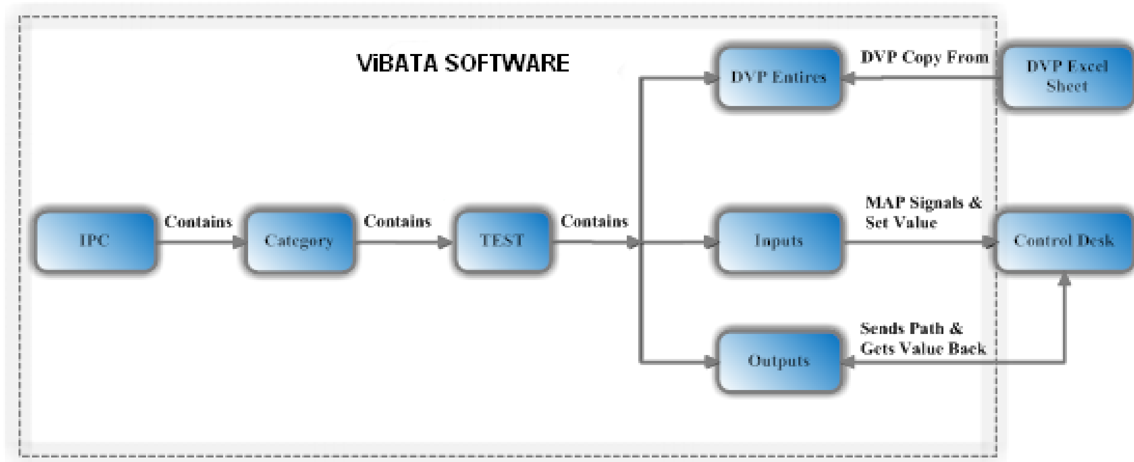

Figure 4.9: ***ViBATA Software sending output signal's path and getting value back***

When Test is entered and saved into the database. It can be run in Test Execution tab and results can be matched with the pattern saved in the system to decide whether test passed or failed as shown in Figure 4.10.

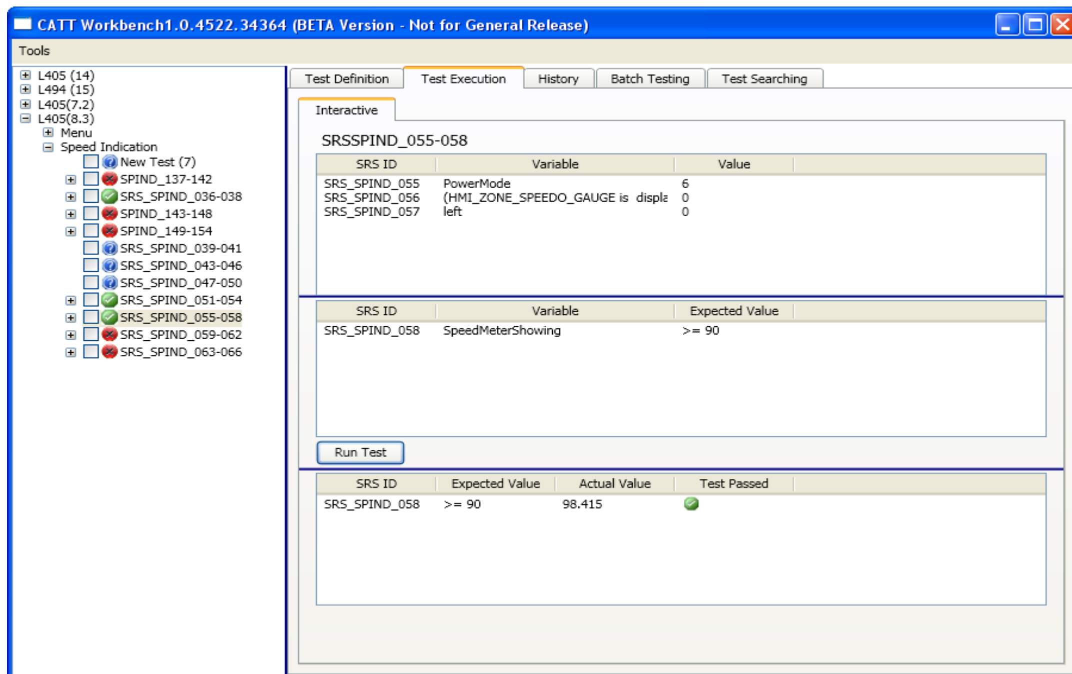

Figure 4.10***: Test Execution tab to run Test***

History tab is divided into Test History and Batch History sub tabs. Test history sub tab shows last 20 results of a test execution order by date. Batch History shows results of test executed in batch in last seven days showed in Figure 4.11.

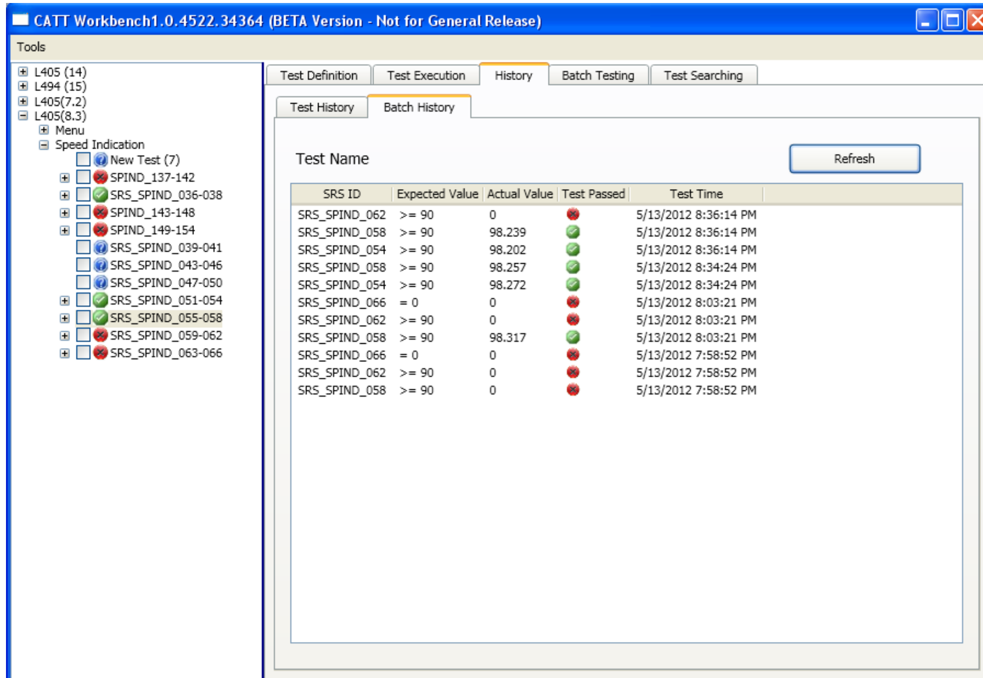

Figure 4.11*: Batch History of Tests*

Batch Testing allows running tests in batch and generating test reports into .csv format. Tests can be selected one by one by checking checkboxes in front of tests or by selecting radio buttons on the page with descriptive labels. Batch testing is shown in Figure 4.12.

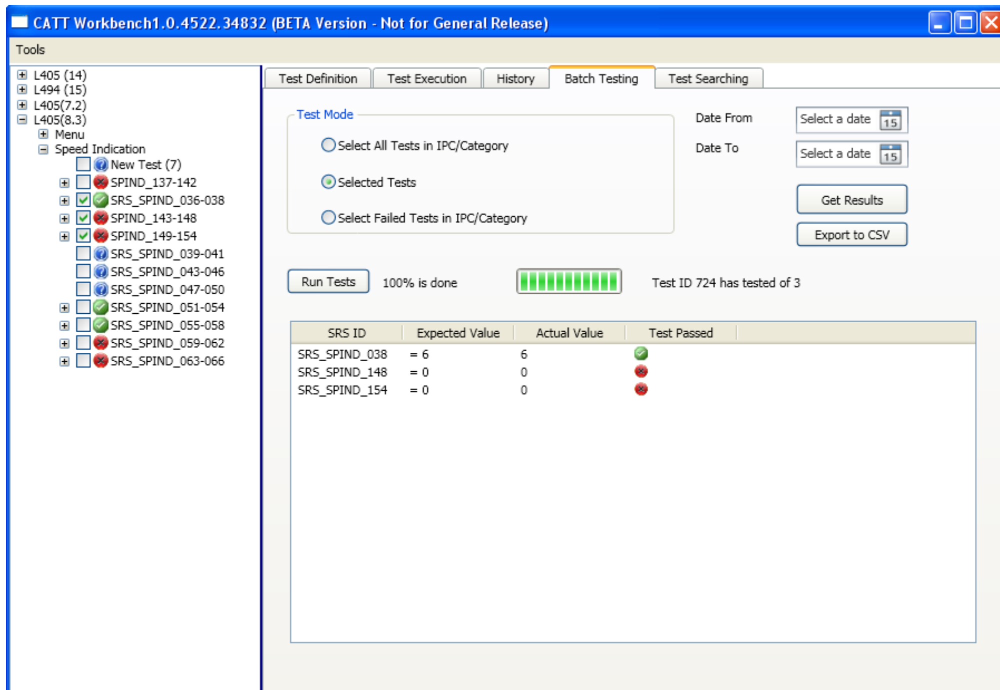

Figure 4.12*: Batch testing tab*

Test Search tab allows searching of already present test case in new DVP (excel sheet) file when it arrives. It allows browsing for excel file in the system and enter worksheet name and brings back the test case if it is present in the file. If found then it gets copied and can be pasted onto DVP entries section on Test Definition tab. Test Search tab is shown in Figure 4.13.

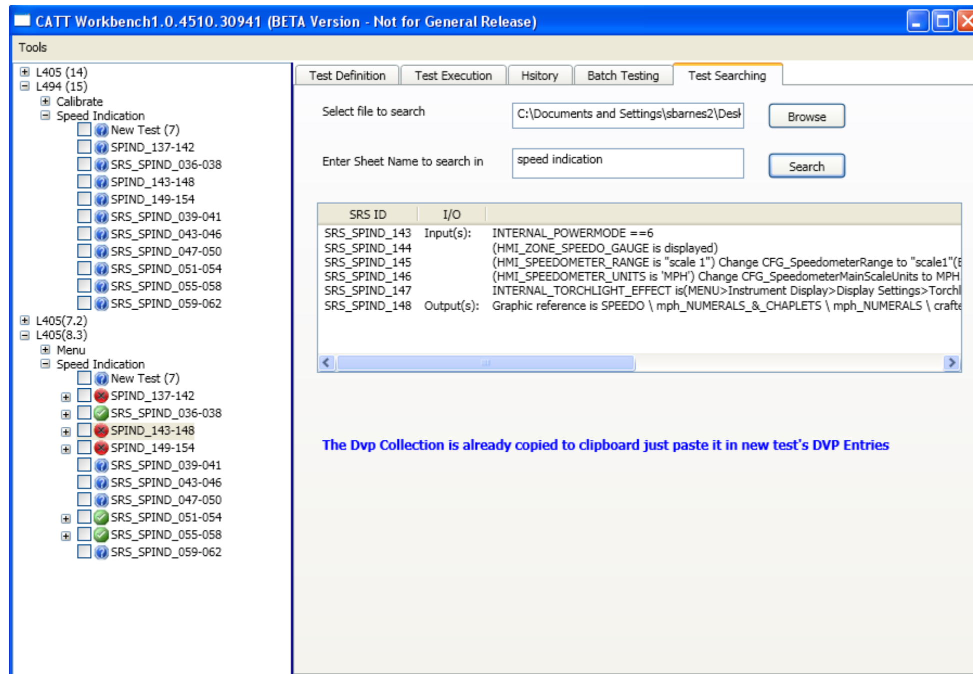

Figure 4.13*: Test Searching Tab*

4.3 Overview of DSL

For every car model a new excel sheet of test cases is built. As described earlier, the transference of test cases from Excel sheet to TSM component of ViBATA (section 4.2) is still manual which can be automated as well. This might achieve by implementing a functionality into software which would be smart enough to recognise start and end of test case in the excel sheet, IPC name from the name of the sheet's title, category name from name of the workbook, and enters into the system. No doubt it can be achieved but would this functionality be consistent with every release of excel sheet and can be used in long run. A minor mistake would enter all test cases in wrong category or input/output lines in wrong test case. If we take account of time consumption from typing test scripts in an excel sheet to entering these into the system. DSL consumes less time and makes the process more efficient. So DSL is the best choice to define test scripts. Learning DSL for a domain user is easier because of containment of abstractions familiar to him. He will write a program in a DSL to generate a code in GPL which will execute in target platform. The DSL studied in this research will address only TSM of ViBATA to automate it. The

user will instruct inputs and outputs of the test cases in program written using DSL which will generate code in Java to produce an output in the form of XML file readable by ViBATA. Just to remind you that we are using Eclipse for making this DSL. This is an external type of textual DSL. Xtext is shipped with Eclipse to define grammar rules defined in 4.4.2. The GPL for code generation in this project is Java because a program written in Java can execute within Eclipse and gives desired output.

4.4 Implementation of Development stages of DSL

This section provides a detail implementation of technologies defined in chapter 3 and whole process of DSL development studied in this thesis as shown in Figure 4.14.

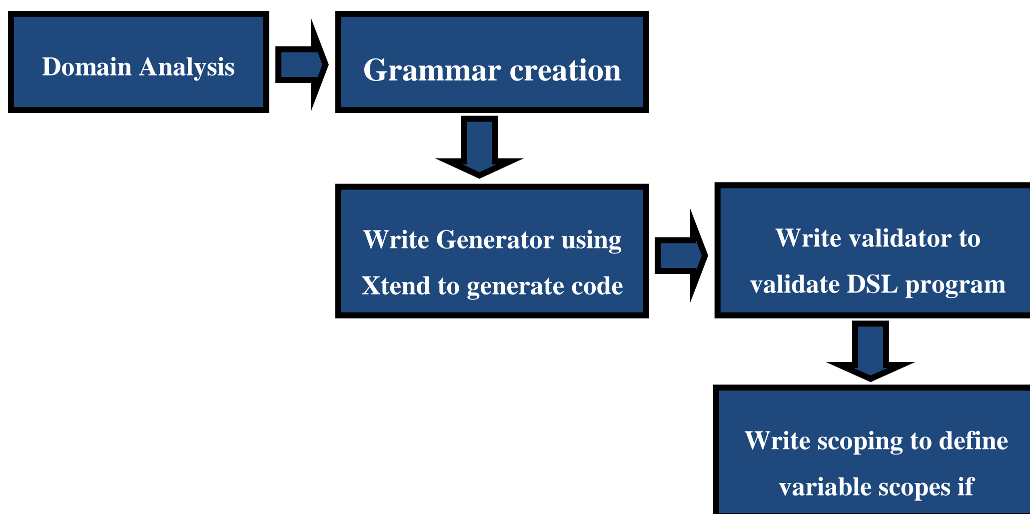

Figure 4.14*: Developmental stages of a DSL*

4.4.1 Domain Analysis

The first stage in development of DSL is the ‘Domain Analysis’ as defined earlier. This stage involves gathering information about the domain. The domain in this case is test specification in testing automotive systems. The structure of this DSL is made more general so it can accommodate test case specification for all embedded systems. For these reasons keywords common to a test case in testing environment are used such as Input, Output, Test, Device, TestCase,

and TestSuite. The main elements of this domain, having certain characteristics, are: Inputs, Outputs, Device, and Test Case. Inputs and outputs constitute a test case as showed in Figure 4.6. There are four characteristics of an input: SRSID, description, path, and value (Figure 4.6). Similarly characteristics of an output are: SRSID, description and expected value (Figure 4.6). Device such as camera has characteristics like: name, connection settings, username and password. SRSID for input and output are unique for each model of IPC which provides the basis of creating test name. Each test case belongs to a category which belongs to an IPC. Categories in one IPC are unique as well. So the first thing come up from these domain elements is we need classes in GPL with all these characteristics as properties and have some functionality inside main method of the program to manipulate these classes with set values to generate an output in the form of XML file.

4.4.2 Using Domain Elements to Create Grammar Rules

In this section some of grammar rules are defined to give an idea how to use domain elements to compose grammar rules. As explained in chapter 3, grammar rules make the concrete syntax of DSL and in this project Xtext is used for rule composition. Detail instruction on how to write the grammar rules is defined in [31]. Important part of a grammar is its header because it gives name to the grammar and decides whether project will be using generator or JvmModelInferrer class for code generation. Header also decides if this grammar will inherit from pre-defined super grammar and reused rules defined in it. Super-grammar Xbase and Terminals are part of Eclipse and a grammar can inherit any of these grammars. The grammar showed in Figure 4.15 is inherited from a super grammar Terminals which defines terminal rules like ID, STRING, and COMMENTS. Rule ID defines the name of the element and corresponds to a regular expression which means it is a sequence of characters, digits and underscore and rule SRING defines sequence of characters enclosed in single/double quotes. Rules ID and STRING are mostly used in the construction of rules in current study. Using Terminals on creating artefacts generator package will be created which is used to generate code for the model in standalone scenario. On the other hand if grammar is inherited

from Xbase, which supports expressions and cross links to Java types, instead of Generator IJvmModelInferrer stub will be generated which is used to translate model directly to Java code as explained in next section. Inheriting grammar from Xbase can also create Generator stub by changing runtime module of the source project as explained in [35] but that is out of scope in the current study. For this project, requirement is to implement Generator so that code can be generated in any GPL. This is the reason grammar for this DSL is created using Terminals grammar. The approach used for this project can generate code in any GPL which in this case is Java language. Figure 4.15 shows the first rule 'Domainmodel' (1) of DSL which states the program will start with a keyword Package followed by its name. It also defines that within open and closed curly brackets arbitrary number (*) of Import (5) and AbstractElement (2) can be added (+=) to properties imports and elements. An 'AbstractElement' (2) points to rules 'Type', 'Communication' (section 5.2.3), and 'Suite'. A 'Type' (3) can be a 'DataType' or 'Entity'. A 'DataType' is having property classifier 'DataType' (4) with a name. The property classifier is explained later in this section. The 'Import' rule starts with a keyword 'import' followed by a name 'importedNamespace' which if used in parser rule the framework treats the rule as an import and 'QualifiedNameWithWildcard' returns string as 'QualifiedName' [31].

```
grammar org.xtext.example.catt.CATT with org.eclipse.xtext.common.Terminals
generate cATT "http://www.xtext.org/example/catt/CATT"


Domainmodel:                                (1)
    {CattGenerator}
    'Package' name = QualifiedName '{'
    (imports += Import)*
    (elements+= AbstractElement)*
    '}'
;

  ;
  AbstractElement:                          (2)
      Type|Communication|Suite
  ;
  Type:                                     (3)
      DataType |Entity
  ;                                         (4)
  DataType:
      classifier='DataType' name = ID
  ;

  Import:                                   (5)
      'import' importedNamespace = QualifiedNameWithWildcard
  ;
```

Figure 4.15*: Grammar of DSL in CATT.xtext File (1)*

In the Figure 4.16 rules in addition to the above rules defined in Catt.xtext file are shown. The rule 'Declaration' (1) points to rules 'varDec' (2) and 'listVarDec' (3) which declare a single variable and a list variable with name and type refers to rule 'Type' defined above. This grammar borrows Entity and Feature rules from Xtext documentation [31]. The difference is the introduction of classifier with rule Entity (4) which can be Input, Output, Test and Device; and Node with rule Feature (5). The name property in any rule cannot be restricted. By introducing classifier entity declaration in DSL and class declaration in Java can be restricted. The reasons for this restriction are first DSL is for test domain so classifiers are domain elements and second this will allow user to build entities with these classifiers only and maintain consistency between the output of DSL which is in XML and plug-in in ViBATA. This is shown in section 4.4.4 that whatever 'name' user gives to the Entity the class generated in Java will be with name of 'classifier' and not with 'name' of Entity. For example entity with classifier 'Input' always generates Input.java this is shown in detail in section 4.5.3. The user can build only one entity with one classifier to avoid duplication. Name of the entity is to define the type of the variable only in 'Case' block. The 'Node' (6) rule defines that a Feature could have a node 'Ele', 'Attr', or 'EleList' which produces annotation for a Feature in generated Code e.g. 'Ele' node will create @Element annotation for a feature and will be generated as an element in XML file. It is described in section 4.4.3 that how to generated annotation from 'Node' and result will be shown in section of 5.2. To generate output in XML this project uses Simple framework which requires annotation for each property in a Java class as described in chapter 3.

```
Declaration:                          (1)
     varDec |listVarDec
;

varDec:                               (2)
     'Declare' name = ID ':' type=[Type|QualifiedName]
;
listVarDec:                           (3)
     'Declare' name=ID 'List of' type=[Type|QualifiedName]
;

Entity:                               (4)
     classifier=('Input'|'Output'|'Test'|'Device') name = ID
     ('extends' superType=[Type|QualifiedName])?
     '{'
     (features += Feature)*
     '}'
;                                     (5)
Feature:
      {Feature} name= ID ':' type= [Type|QualifiedName] node= Node |
      {Feature} name = ID 'List of' type=[Type|QualifiedName] node = Node
;

Node:                                 (6)
nodeType = ('Attr' | 'Ele' | 'EleList')
```

Figure 4.16*: Grammar of DSL in CATT.xtext File (2)*

For the DSL expressions are created in the grammar from scratch as shown in Figure 4.17. An expression could be a conditional or assignment but for this grammar only assignment expressions are used. Assignments in this DSL are of two types. One is for single variable and other is for the variable holds list of elements. There are two kinds for both of these assignments. One is assignment for variable declared in the 'Case' block and other assignment is for the block itself. Rule 'dotFunc' (Figure 4.17) defines an expression for variable with left part refers to rule 'varDec' followed by '.' and right part refers to rule 'Feature'. Rule 'myFunc' is same as 'dotFunc' with difference of inclusion of 'my' keyword for the 'Case' block (shown and explained later in this section) to assign values to its own features. Rules 'myFuncAssignment' and 'myFuncListAssignment' are for 'Case' block and rules 'Assignment' and 'listAssignment' are for declared variable in 'Case' block. Similarly there are two kinds of functions one is for 'Case' block to add it to 'Suite' and one is to add a variable in declared list variable. These functions are defined in rules 'AddFunc'

and ‘meAddFunc’. Left part of rule ‘AddFunc’ refers to a list variable followed by keyword ‘.add’ and right part refers to variable to be added in the list enclosed in brackets. Rule ‘meAddFunc’ has keywords ‘me.add’ and ‘=’ followed by bool literal which can be true or false. This is to decide whether specified ‘Case’ should be part of ‘Suite’ or not to be generated in final output XML file.

```
dotFunc:
    dec=[varDec] '.' feature=[Feature]
;
myFunc:
    dec= 'my' '.' feature=[Feature]
;
myFuncAssignment:
    dot= myFunc '=' literal=Literal
;
myFunctionListAssignment:
    dot=myFunc '=' lisVar=[ListVarDec]
;
Assignment:
     dot=dotFunc '=' literal=Literal
;
listAssignment:
    dot=dotFunc '=' lisVar=[ListVarDec]
;
AddFunc:
    ldec=[ListVarDec] '.add'  '(' varDec=[varDec] ')'
;

meAddFunc:
'me.add' '=' literal=BooleanLiteral
;
```

Figure 4.17*: Expressions and Assignments in DSL*

The same technique of using classifier attribute is used with rules ‘Case’ and ‘Suite’ as shown in Figure 4.18. Classifiers for rule ‘Suite’ are ‘TestSuite’ and ‘DeviceSuite’ and for rule ‘Case’ are ‘TestCase’ and ‘DeviceInfo’. For rule ‘Case’ an attribute request is also defined which sets the mode of Test/Device case and goes to an Enum Rule ‘RequestType’ defined in the grammar. Enum ‘RequestType’ can be ‘Create’, ‘Update’ or ‘Delete’ which means a test can be created, updated or deleted. This will be shown in chapter 5 under section 5.2.

```
,
Suite:
    classifier=('TestSuite'|'DeviceSuite') name = ID '{'
    (declaration += Declaration)*
    (Cases += Case)*
    (listassigns += listAssignment)*
    '}'
;

Case:
    request =RequestType classifier=('TestCase'|'DeviceInfo') name=ID
    '{'
    (declarations+=Declaration)*
    (myassignments += myFuncAssignment)*
    (assignments += Assignment)*
     (functions += Function)*
     addFunc = meAddFunc

    '}'
;

enum RequestType :
    Create | Update | Delete;
```

Figure 4.18*: Grammar rules for Case and Suite*

4.4.3 Writing Code Generator in Xtend

Eclipse introduces Xtend Language to write a code generator for a program written in DSL. There are more than one ways to implement a code generator in Xtend. It can be generated in any GPL by using template expression or any specific language by injecting a compiler or interpreter. A generator can be written by implementing the Xtext interface IGenerator or extending AbstractModelInferrer in Xtend. Full documentation on how to write a code generator using Xtend is available at [34]. If grammar is inherited from Xbase the code will be generated only in Java. Xbase is integrated with Java Type system and provides both control structures and program expressions. Most of the programming languages share common understanding of expressions which is an effort to build from scratch for a new DSL. This is the reason Xbase is introduced so programmers can use it in Xtext to define expressions, assignments and type-systems [36]. In this section, first type of code generation

used which is template expression in the current study, challenges faced using this technique and later example using Xbase and why it is not used is explained.

Template Expression

As mentioned before the current project is using Template Expression to generate code for each element in Semantic Model. This is not Template Generation mentioned in section 3.2.6 but in 3.1.2. A template expression can be composed of multiple lines and is used to allow string concatenation surrounded by three single quotes [34]. In this part of section, the use of 'Template Expression' to generate code is defined. The 'Generator' class in this project generates two types of java classes. One for the main java program and other for each entity defined in the DSL program. The code stub which does this in Xtend is shown in Figure 4.19.

```
class CATTGenerator implements IGenerator {
    @Inject extension IQualifiedNameProvider
    override void doGenerate(Resource resource, IFileSystemAccess fsa) {
        for (e: resource.allContents.toIterable.filter(typeof (Entity))){
            //val name=e.eContainer.fullyQualifiedName+("/")    + e.classifier
            fsa.generateFile(e.eContainer.fullyQualifiedName+"/" + e.classifier + ".java", e.compile)
            fsa.generateFile(resource.className +".java", toJavaCode(resource.contents.head as Domainmodel,e))
    }
    }
    def className(Resource res) {
        var name = res.URI.lastSegment
        return name.substring(0, name.indexOf('.'))
        }
```

Figure 4.19*: Xtend stub to generate .java file for main program and for Entity*

The code consists of a function 'doGenerate' which takes arguments of type 'Resource' and IFileSystemAccess. It takes the 'Resource' which is DSL program and iterate over each element in it to look for 'Entity', creates a .java file with name defined as classifier in the entity and goes to a function 'compile' for entity. Secondly, it calls two functions one 'className' for 'Resource' and brings back the name of the file on the left side of '.' extracted from the resource's URI and second toJavaCode with arguments of type 'Domainmodel' and 'Entity' explained later in this section. The code stub of function compile() for Enity is shown in Figure 4.20. This part of function checks if entity's classifier

is Test/Device if yes it creates a class named as TestSuite/DeviceSuite in the same .java file for Entity and declare a variable and property which define, set and get list of elements of type Test/Device.

```
def compile(Entity e) '''
«IF e.eContainer != null»
package «e.eContainer.fullyQualifiedName»;
«ENDIF»
import org.simpleframework.xml.Attribute;
import org.simpleframework.xml.Element;
import org.simpleframework.xml.Root;
import org.simpleframework.xml.ElementList;
import java.util.List;
«IF e.classifier == 'Test' || e.classifier == 'Device'»
@Root
class «e.classifier»Suite{

@ElementList

private List<«e.classifier»> «e.classifier»s;
public List<«e.classifier»> get«e.classifier»s() {
return «e.classifier»s;
}
public void set«e.classifier»s(List<«e.classifier»> «e.classifier»s) {
this.«e.classifier»s = «e.classifier»s;
}
}
```

Figure 4.20*: Xtend stub to generate TestSuite/DeviceSuite Class*

The second part of function which is shown in Figure 4.21 performs two tasks. First it defines class declaration and an additional property 'Mode' for entity with classifier Test/Device which will be set by 'RequestType' in the rule 'Case' will be shown later in section 5.2.

```
@Root
public class «e.classifier»
«IF e.superType !=null» extends «e.superType.fullyQualifiedName» «ENDIF»{
«IF e.classifier == 'Test' || e.classifier == 'Device'»

@Attribute
private String mode;
public String getMode() {
return mode;
}
public void setMode(String mode) {
this.mode = mode;
}
«ENDIF»
«FOR f:e.features»
«f.compile»
«ENDFOR»
}
'''
```

Figure 4.21*: Defining Mode property of the class Test/Device*

Secondly, it calls another function compile() for each 'Feature' of the entity which will get and set the java property with 'name' and 'type' of 'Feature' defined under entity as shown in Figure 4.22

```
def compile(Feature f) '''
«IF f.node != null»
«f.node.createAtt»
«ENDIF»
«IF f.type.toString.contains('Entity')»
«IF f.node.nodeType == 'EleList'»
private List<«f.type.classifier»> «f.name»;
public List<«f.type.classifier»> get«f.name.toFirstUpper»() {
return «f.name»;
}
public void set«f.name.toFirstUpper»(List<«f.type.classifier»> «f.name») {
this.«f.name» = «f.name»;
}
«ELSE»
private «f.type.classifier» «f.name»;
public «f.type.classifier» get«f.name.toFirstUpper»() {
return «f.name»;
}
public void set«f.name.toFirstUpper»(«f.type.classifier» «f.name») {
this.«f.name» = «f.name»;
}
«ENDIF»
def createAtt(Node n)'''
 «IF n.nodeType == 'Attr'»
 @Attribute
 «ELSEIF n.nodeType == 'Ele'»
 @Element
 «ELSE»
 @ElementList
 «ENDIF»
'''
```

Figure 4.22*: Creation of java property for each 'Feature' with annotation*

The code checks for the 'Node' first through a createAtt() function call and sets the annotation according to the node of the 'Feature'. Then it checks if 'Feature' is of type 'Entity' if true the type will come from classifier otherwise name of the type and if node is 'EleList' then will create a list variable as will be shown in section 4.5.2.

Challenges

The initial challenge faced in generating code using template expression was the iteration through model elements especially from one level to level down. First have a look at Figure 4.23 and Figure 4.24

```
;
Type:
    DataType |Entity
;
DataType:
    classifier='DataType' name = ID
;
```

Figure 4.23*: Accessing one model element from another element*

```
CATT.xtext    CATTGenerator.xtend

Case:
    request =RequestType classifier=('TestCase'|'DeviceInfo') name=ID
    '{'
    (declarations+=Declaration)*
    (myassignments += myFuncAssignment)*
    (assignments += Assignment)*
     (functions += Function)*
     addFunc = meAddFunc

    '}'
;

Function:
    AddFunc|listAssignment|myFunctionListAssignment
;

Declaration:
    varDec |listVarDec
;

varDec:
    'Declare' name = ID ':' type=[Type|QualifiedName]
;
listVarDec:
    'Declare' name=ID 'List of' type=[Type|QualifiedName]
;
```

Figure 4.24*: Accessing one model element from another element*

In these screenshots some rules are shown like ‘Case’, ‘Declarations’, ‘VarDec’ and ‘ListVarDec’. Rules ‘VarDec’ and ‘ListVarDec’ are on same level under rule

'Declaration'. To access a feature like 'name' and 'type' of rule 'VarDec' or 'ListVarDec' from 'Case' one will need to access 'Declaration' first then check what type of 'Declaration' it is. Same property names, of the rules on one level separated by vertical line, will appear on code completion window by pressing Ctrl and Space otherwise one will need to cast the top rule element into required low level rule element. To illustrate this first how to access features in code generator if rules have the same property names, and later if properties are different how to cast them is demonstrated. Consider Figure 4.25 and see how it is done in code generator

```
CATT.xtext    CATTGenerator.xtend
}
«FOR tSuite:dm.eAllContents.toIterable.filter(typeof (Suite))»
«FOR tc:tSuite.cases»
«IF tc.classifier == 'TestCase'»
public static Test buildTestCase«tc.name»(){
Test «tc.name» = new Test();
«ELSEIF tc.classifier == 'DeviceInfo'»
public static Device buildDeviceInfo«tc.name»(){
Device «tc.name» = new Device();
«ENDIF»
«FOR dec:tc.declarations»
            «IF dec.type.classifier.equals("DataType")»
               «IF dec.toString.contains('varDec')»
               «dec.type.name» «dec.name» = new «dec.t    toString : String - Object.toString()
               «ELSE»                                     type : Type - Declaration.getType()
               List<«dec.type.name»> «dec.name» = new
               «ENDIF»
            «ELSE»
               «IF dec.toString.contains('varDec')»
                «dec.type.classifier» «dec.name» = new «dec.type.classifier»();
               «ELSE»
               List<«dec.type.classifier»> «dec.name» = new ArrayList<>();
               «ENDIF»
            «ENDIF»
«ENDFOR»
```

Figure 4.25*: Code Generation snippet to understand Element Access*

ToIterable() extension method of class IteratorExtensions gives TreeIterator in for loop to iterate over the contents of a certain element and get all containing features and elements through getAllContents() method [31]. In Figure 4.25 'Suite' is accessed same way we got 'Entity' in the section above and rule 'Case' is contained in it as shown in Figure 4.18. As mentioned before here classifier is used to restrict user to create 'Case' of type 'TestCase' and 'DeviceInfo'. Here code is generating according to the classifier of the 'Case'. To check the type of 'Declaration' classifier property distinguishes between

Type of ‘Entity’ and ‘DataType’. Both rules VarDec and ListVarDec has features ‘name’ and ‘type’ so these features can be access directly from code completion window as shown in Figure 4.25. Now In Figure 4.26 rules ‘AddFunc’, ‘listAssingment’ and ‘myFunctionListAssignment’ are under main rule ‘Function’ on same level but having different property names for example ‘AddFunc’ is having first property ‘ldec’ and second varDec but rule ‘listAssignment’ and ‘myFunctionListAssignment’ both having first property ‘dot’ and second ‘lisVar’.

```
Function:
    AddFunc|listAssignment|myFunctionListAssignment
;
AddFunc:
    ldec=[listVarDec] '.add'  '(' varDec=[varDec] ')'
;

myFunctionListAssignment:
    dot=myFunc '=' lisVar=[listVarDec]
;

listAssignment:
    dot=dotFunc '=' lisVar=[listVarDec]
;
```

Figure 4.26***: Accessing rule from top level rule***

To access these low level rules one will need to cast the top level rule into lower level rule. In this case the property names will not appear in the code completion window. In Figure 4.27 to access ‘AddFunction’ from functrions property of ‘Case’ one will need to cast the rule ‘Function’ into ‘Addfunc’ and then can access its properties on code completion window to write code for them as shown in Figure 4.27.

```
«FOR caseT:tc.functions»
        «IF caseT instanceof AddFunc»
                «var AddFunc add = caseT as AddFunc»
                    «add.ldec.name».add(«add.varDec.name»);
        «ELSEIF caseT.toString.contains("listAssignment")»
            «var listAssingment addList= caseT as listAssignment»
                «addList.dot.dec.name».set«addList.dot.feature.name.toFirstUpper»(«addList.lis
        «ELSEIF caseT instanceof myFunctionListAssignment»
            «var myFunctionListAssignment addList = caseT as myFunctionListAssignment»
            «tc.name».set«addList.dot.feature.name.toFirstUpper»(«addList.lisVar.name»);
        «ENDIF»
«ENDFOR»
```

Figure 4.27*: Accessing rule from top level rule*

Why not Xbase

Now this part of current section explains how to use Xbase in Xtext first and then Inferrer class to generate code from it. Both Figure 4.28 *and* Figure 4.29 show an excerpt from Xtext documentation [31]. Consider Figure 4.28 first, the rules are defined in a grammar which are inherited from Xbase. Here type of rule 'Property' is JvmTypeReference which is given in super-grammar Xbase and defines Java-like type names.

```
Import:
  'import' importedNamespace=QualifiedNameWithWildCard;

QualifiedNameWithWildCard :
  QualifiedName  ('.' '*')?;

Entity:
  'entity' name=ValidID
    ('extends' superType=JvmTypeReference)? '{'
    features+=Feature*
  '}';

Feature:
  Property | Operation;

Property:
  name=ValidID ':' type=JvmTypeReference;
```

Figure 4.28*: Using Xbase in Xtext*

CattXbase.xtext | *CattXbaseJvmModelInferrer.xtend | org.xtext.example.cattXbase.ui

```
def dispatch void infer(Entity entity, IJvmDeclaredTypeAcceptor accept
    acceptor.accept(
        entity.toClass( entity.fullyQualifiedName )
    ).initializeLater [
        documentation = entity.documentation
        if (entity.superType != null)
            superTypes += entity.superType.cloneWithProxies
        val procedure = entity.newTypeRef(typeof(Procedure1), it.newTy
        members += entity.toConstructor() []
        members += entity.toConstructor() [
            parameters += entity.toParameter("initializer", procedure)
            body = [it.append("initializer.apply(this);")]
        ]
        val fields = <JvmField>newArrayList()
        for ( f : entity.features ) {
            switch f {

                Property : {
                    val field = f.toField(f.name, f.type)
                    fields += field
                    members += field
                    members += f.toGetter(f.name, f.type)
                    members += f.toSetter(f.name, f.type)
                }
```

Figure 4.29: ***JVM Model Inferrer Class***

In Figure 4.29 some methods are shown exposed by java model inferrer class like toClass, toGetter and toSetter. These methods generate class, setter and getter directly for the model object ‘Enity’ in Java.

It is important to note that in the inferrer class the acceptor.accept() method is used to recognise every JvmDeclared type which takes it as a parameter. Here it is taking ‘Entity’ so that it can be recognized as JvmType. In case this is not done an error will be shown in program written in DSL that states “Couldn’t resolve reference to JvmType”. For example consider a tutorial on Fowler’s statemachine example implemented with Xtext and Xtend 2.3 using Xbase and inferrer class in [37]. In this tutorial author defines rule ‘Service’ with type JvmTypeReference and name (Figure 4.30). In the inferrer method he is not using acceptor method for rule ‘Service’ to recognise its type. That is why when we create a program in DSL the error shows up (Figure 4.31). To overcome this we need to create a Java class with name of declared ‘Service’ type and put it

inside source folder of project and this error will be resolved as shown in Figure 4.32

```
Service :
    type=JvmTypeReference name=ID
;
```

```
acceptor.accept(stm.toClass(stm.className)).initializeLater [

  // add a field for each service annotated with @Inject
  members += stm.services.map[ service |
    service.toField(service.name, service.type) [
      annotations += service.toAnnotation(typeof(Inject))
    ]
  ]
```

Figure 4.30: ***Service Rule in DSL and Code Generation in Inferrer***

```
end

services
  DoorService door
                        Couldn't resolve reference to JvmType 'DoorService'.
en                                                        Press 'F2' for focus

state idle
  do {
    door.open
    panel.close
  }
```

Figure 4.31: ***Error shown because DoorService is not identified as JvmType***

```
myjava
  src
    (default package)
      DoorService.java
        DoorService
          close() : void
          open() : void
      PanelService.java
    myf.mydsl
  JRE System Library [JavaSE-1.7]
  src-gen
    myf.java
```

```
DoorService.java   myf.mydsl

services
  DoorService door
  PanelService panel
end

state idle
  do {
    door.open
    panel.close
  }
  doorClosed => active
end
```

Figure 4.32: ***Error resolved by creating Java class on runtime***

For current study the objective is to build a DSL which could be transformed into any GPL including Java. This was the top reason of building it with template expression otherwise using Xbase with Inferrer class was more convenient way

to generate code in Java. Besides it took time to understand each of the challenges described above. In this project Simple framework is used to serialize test cases into XML which requires annotation for each of the element in the test case. This was another challenge with inferrer class which was made possible by using template expressions as shown in this section.

4.4.4 Model Validation

Code analysis and validation are quite important features while building a language with Xtext. These features improve language user support while typing a program in DSL. Most of the validation is done automatically. There are three different kinds of validation exposed by Xtext.

1. Automatic
2. Custom
3. Manual.

Automatic involves mostly syntactic validation which is done by parser and error messages are shown by its underlying technology. Details of each validation type are given in Xtext documentation [31] [38]. Custom validation is more related to semantics of the language. So we are more interested in custom validation for the sake of current project. With the custom validation we can specify additional constraints for our Ecore model. On creating model artefacts a required EValidator API is registered in generator fragment which is Java-based known as JavaValidatorFragment. This will generate two java classes one is abstract class derived from AbstractDeclarativeValidator in scr-gen folder and other which is derived from this class in src folder of the project. The second class named as CATTJavaValidator.java is the one which we will modify and put custom validation code in it.

As explained earlier, names of the domain elements in the current study were restricted for the user so he can create entities or test case of given classifier. There was a need of validation so that user cannot create two domain elements with the same classifier. A Check annotation is placed above every method in this class which invokes automatically when validation takes place. These

methods take parameters to state what type the respective constraint method is for.

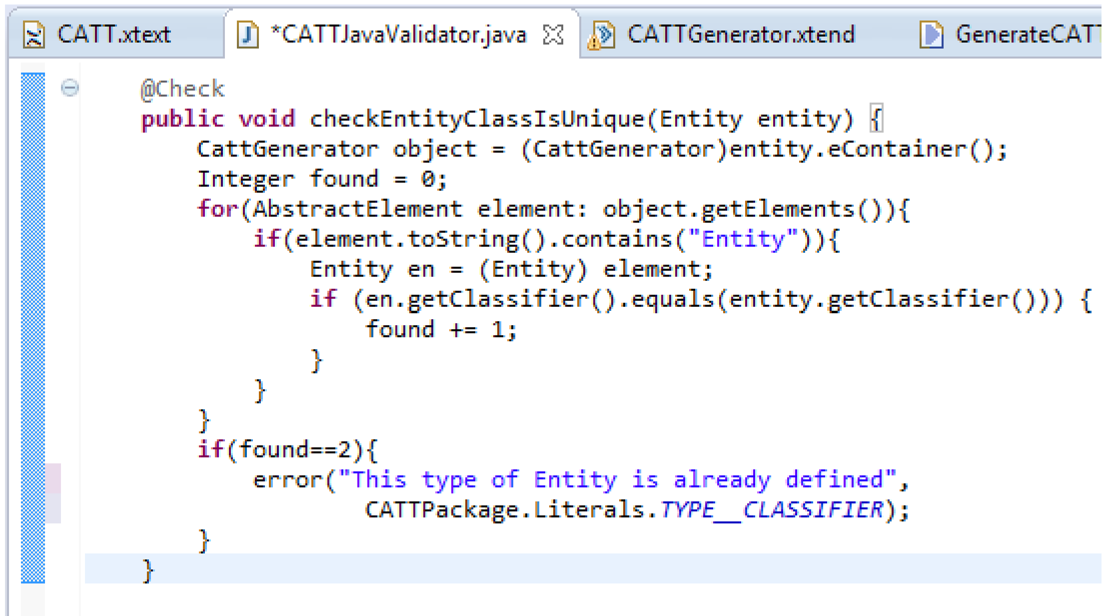

Figure 4.33*: Constraint method to validate Unique Entity*

Figure 4.33 shows a custom validation method which checks Entity's classifier is unique in the program. On creating two entities with same classifier it shows a custom error which states "This type of Entity is already defined". Figure 4.34 shows implementation of this validation in DSL program.

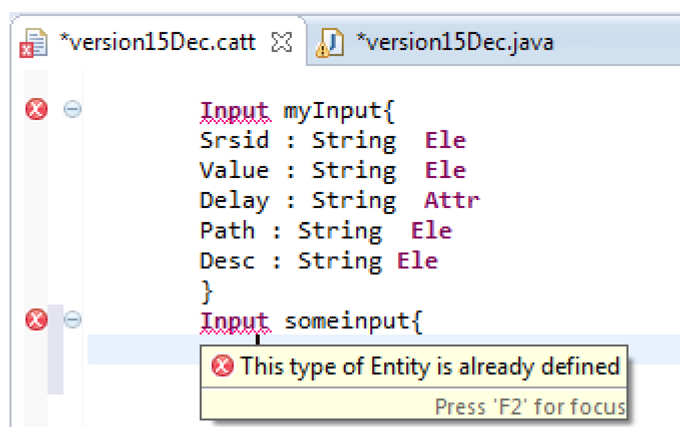

Figure 4.34*: Unique Entity Validation*

Similarly, other methods are in place to check if feature name is unique in certain Entity and test name is unique in test Suite of the program. There is

another important method which validates if assignment is unique i.e. same feature cannot have two assignments. Automatic fixes for an error and/or warning can also be implemented which fixes the error while typing. To do this the underlying cause of the error should be known first. This is done by providing QuickFixProvider fragment in generator fragment which generates, on creating artefacts, an empty QuickFixProvider class in DSL's UI project [31][38]. This is out of scope of this study.

This is our 5th objective to enable DSL to detect errors which is accomplished by validating model.

4.4.5 Model Scoping

Scoping defines which elements in a model are referable by certain reference. For example consider Figure 4.35. This grammar states the rule 'dotFunc' is having cross reference 'dec' which can have only instances of rule 'varDec' and 'feature' with instances of rule 'Feature' only. But this doesn't explain what is the type of 'varDec' and if type of Feature is compatible with it. This is explained by scoping implemented by IScopeProvider responsible for providing IScope for a given EObject and EReference. The returned IScope object should contain all target elements for a given EObject and cross-reference [31]

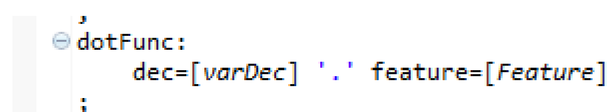

Figure 4.35*: Rule defining Scoping*

With other artefacts ScopeProvider Java class is also generated which can be customised to provide scope for objects in model. For the above rule a method is created which checks the type of variable and if type is 'Entity' it provides the features contained in the code completion window. Figure 4.36 shows this method in ScopeProvider class and Figure 4.37 shows its result in DSL program. There are two types of scoping Global and Local. If model definition is

spread across many files then scope for objects is provided by Global scoping. If every domain element is contained in single file then local scoping is used as shown in this section. Details on how to implement Global scoping is provided in [31].

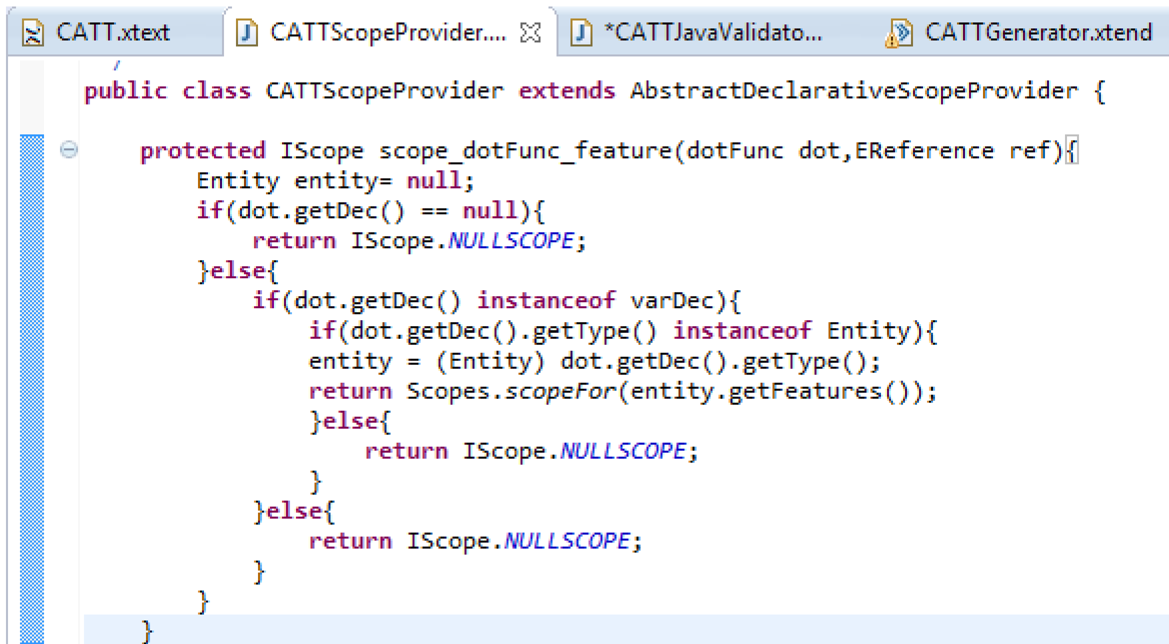

Figure 4.36: ***Implementation of Feature scope in DotFunc***

```
Input myInput{
   Srsid : String  Ele
   Value : String  Ele
   Delay : String  Attr
   Path : String  Ele
   Desc : String Ele
   }
```

```
TestSuite mysuite{
    Create TestCase thiscase{
    Declare Ins List of myInput
    Declare Outs List of myOutput
    Declare Input1 : myInput
    Declare Input2 : myInput
    Declare Output1 : myOutput
    my.Name ="SRS_EXTLG_733-735"
    my.IPCName="L405(8.6)"
    my.CategoryName="Exterior Lighting"
    my.RunTest= false
    Input1.Desc = "INTERNAL_POWER_MODE ==6"
    Input1.  .add
    Input1.  Delay
    Input1.  Desc
    Input1.  Path
    Input2.  Srsid
    Input2.  Value
    Input2.
    Input2.
    Input2.
```

Figure 4.37: ***Implementation of scope of Features according to Entity***

4.4.6 Content Assist

In the UI project of the language Xtext generates two files. One in *src-gen* folder named as AbstractCATTPropsalProvider and in *src* folder CATTProposalProvider. AbstractProposalProvider class contains complete_method for each assigned property and rule in the grammar. CATTProposalProvider inherits from AbstractProposalProvider which can be customised to facilitate user with content assistant [31]. Figure 4.38, Figure 4.39 and Figure 4.40 show rules, method in CATTProposalProvider class for rules and its result in program respectively.

```
*CATT.xtext    CATTProposalProvider.java

myFunctionListAssignment:
    dot=myFunc '=' lisVar=[listVarDec]
;

myFunc:
    dec= 'my' '.' feature=[Feature]
;
```

Figure 4.38: ***Rule for myFunctionListAssignment***

```
CATTProposalProvider.java    *CATT.xtext

public void completeMyFunctionListAssignment_LisVar(myFunctionListAssignment lisvar, .
    EObject model = null;
    String proposal;
    //super.completeListAssignment_LisVar(model, assignment, context, acceptor);

    myFunc func = null;
        func = lisvar.getDot();
        Feature feat = func.getFeature();
        Case tcase =(Case) lisvar.eContainer();
        EList<Declaration> dec = tcase.getDeclarations();
        for(Declaration d:dec){
            if(d instanceof listVarDec && d.getType() == feat.getType()){
                proposal = d.getName();
                ICompletionProposal completionProposal =
                        createCompletionProposal(proposal, context);
                //acceptor.accept(createCompletionProposal(proposal,context));
                acceptor.accept(completionProposal);
            }
        }
```

Figure 4.39: ***Method Implementation Content Assistant in ProposalProvider class***

```
 Ins.add(Input1)
 Ins.add(Input2)
 Outs.add(Output1)
 my.Inputs= Ins
 my.Outputs = Outs
 me.add = true      Outs


eate TestCase
 Declare Ins1
 Declare Outs1
```

Figure 4.40*: Showing possible Content according to method*

This is one of our objectives to facilitate user with code completion which is achieved by both scoping in section 4.4.5 and content assistance in this section.

4.5 DSL to Platform Transference

This is the last and final stage when DSL is completed and run in new instance of Eclipse to test in editor where code is generated and executed. As Eclipse is Java friendly IDE and needs JRE (Java Runtime Environment) to install, code generated in Java can be run within the environment on generation. So there is no need to transfer the generated code to the target platform. If the generated code was in some other language like C# it would need to be transferred to Visual Studio and run from there to get output. A full tutorial is given in the

documentation on how to configure settings and launch new instance of Eclipse to try DSL in editor. When new instance of Eclipse launches a new Java or Plug-in project is created. Within this new project a file with extension of DSL created in the language project is created. This file will go in src folder of the project and is used to write program using DSL as shown in next section. On saving this program a src-gen folder is created automatically which contains all the generated code shown in 4.5.2 [31].

4.5.1 Program in DSL

The program written in DSL is saved in file named as version15 with extension .catt which is short for (Cranfield Automated Testing TestBench). This file on saving generates one main java file named after it and one java file for each entity defined named after the entity's classifier. In this case java files for Input, Output, Test, Device and version15 (main file) will be generated as shown in next section. The main file contains a public java class version15 with main() function. Each file for entity contains two java classes if entity's classifier is 'Test' or 'Device' otherwise one java class. The other class for 'Test' or Device is for 'Suite' which contains a java property to get and set the list of tests or devices as shown in section 4.4.3. There are two kinds of cases defined one is 'TestCase' and other is 'DeviceInfo' within their respective suites. Last code stub is taking locations of file and generating the code to serialize the suite into XML file.

The program in our DSL is shown in Figure 4.41 and Figure 4.42

```
*version15.catt
Package cranfield {
import java.lang.*
import java.util.List
DataType String
DataType Boolean

Test thisTest{
Name : String  Attr
IPCName : String    Attr
CategoryName : String  Attr
Inputs List of myInput EleList
Outputs List of myOutput EleList
RunTest : Boolean  Ele
}
Output myOutput{
Srsid : String Ele
ExpectedValue : String  Ele
Variable : String  Ele
Desc: String Ele
}
Input myInput{
Srsid : String  Ele
Value : String  Ele
Delay : String  Attr
Path : String  Ele
Desc : String Ele
}
Device myDevice{
Name : String Ele
Host : String Ele
Port : String Ele
User : String Ele
Pwd : String Ele
```

```
*version15.catt
TestSuite mysuite{
Create TestCase thiscase{
Declare Ins List of myInput
Declare Outs List of myOutput
Declare Input1 : myInput
Declare Input2 : myInput
Declare Output1 : myOutput
my.Name ="SRS_EXTLG_740-745"
my.IPCName="L405(8.6)"
my.CategoryName="Exterior Lighting"
my.RunTest= true
Input1.Desc = "INTERNAL_POWER_MODE ==6"
Input1.Srsid='SRS_EXTLG_733'
Input1.Value="6"
Input1.Delay="1000"
Input1.Path="Model Root/MSCANCOMMS/User:
Input2.Desc="CAN Signal FrontFogLightIn
Input2.Srsid="SRS_EXTLG_734"
Input2.Value="1"
Input2.Delay="1000"
Input2.Path="Model Root/HSCAN_COMMS/Use
Output1.Srsid="SRS_EXTLG_735"
Output1.ExpectedValue=">=95"
Output1.Variable="IND_FRONT_FOG"
Output1.Desc="IND_FRONT_FOG shall be di
Ins.add(Input1)
Ins.add(Input2)
Outs.add(Output1)
my.Inputs= Ins
my.Outputs = Outs
me.add = true
}
```

Figure 4.41*: Program in DSL (1)*

```
DeviceSuite devSuite{
    Create DeviceInfo devInfo{
        my.Name = "Sensor"
        my.Host= "138.250.81.16"
        my.Port="23"
        my.User ="admin"
        my.Pwd ="pwd"
        me.add = true
    }
}
Write XML File myFile{
    FileLocation loc = "E:/TestCases.xml"
    FileLocation deviceLoc = "E:/DeviceInfo.xml"
        run (devSuite,deviceLoc;mysuite,loc)
    }
 }
```

Figure 4.42*: Program in DSL (2)*

4.5.2 Generated Code

When the above program is saved in the project a *src-gen* folder is created which contains all the generated code. This includes four Java Beans named as Input, Output, Test and Device and one file with main() Java method named after file created for DSL program. Figure 4.43 shows the folders in the project and Figure 4.44 shows the generated code respectively.

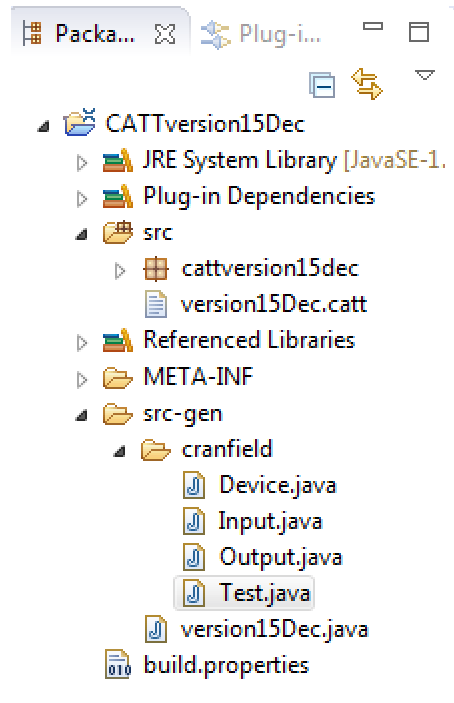

Figure 4.43***: Eclipse Plug-in Project***

Input.java | Test.java | *version15Dec.catt

```java
package cranfield;
import org.simpleframework.xml.Attribute;
import org.simpleframework.xml.Element;
import org.simpleframework.xml.Root;
import org.simpleframework.xml.ElementList;
import java.util.List;
@Root
public class Input{

@Element

private String Srsid;
public String getSrsid() {
return Srsid;
}
public void setSrsid(String Srsid) {
this.Srsid = Srsid;
}

@Element

private String Value;
public String getValue() {
return Value;
}
public void setValue(String Value) {
this.Value = Value;
}
```

Input.java | Test.java | *version15Dec.catt

```java
package cranfield;
import org.simpleframework.xml.Attribute;
import org.simpleframework.xml.Element;
import org.simpleframework.xml.Root;
import org.simpleframework.xml.ElementList;
import java.util.List;
@Root
class TestSuite{

@ElementList

private List<Test> Tests;
public List<Test> getTests() {
return Tests;
}
public void setTests(List<Test> Tests) {
this.Tests = Tests;
}
}
@Root
public class Test{

@Attribute
private String mode;
public String getMode() {
return mode;
}
public void setMode(String mode) {
this.mode = mode;
}
```

version15Dec.java | Input.java | Test.java | *version15Dec.ca

```java
package cranfield;
import java.beans.XMLDecoder;

public class version15Dec {

    public static void main(String[] args) {
            Serializer serializer = new Persister();
            TestSuite mysuite = new TestSuite();
            List<Test> tests = new ArrayList<>();
            Test thiscase = buildTestCasethiscase();
                tests.add(thiscase);
            Test thiscase1 = buildTestCasethiscase1();
                tests.add(thiscase1);
            mysuite.setTests(tests);
            DevSuite devSuite = new DevSuite();
            List<Device> devices = new ArrayList<>();
            Device devInfo = buildDeviceInfodevInfo();
                    devices.add(devInfo);
            devSuite.setDevices(devices);
            File loc = new File("E:/TestCases.xml");
            File deviceLoc = new File("E:/DeviceInfo.xml");
            try{
            serializer.write(devSuite,deviceLoc);
            }catch(Exception ex){
             }
            try{
            serializer.write(mysuite,loc);
            }catch(Exception ex){
             }
    }
```

Figure 4.44*: Generated Code*

The code is generated in Java by using template expressions which can be replaced by code in any other GPL according to client requirement. This way code generation is made flexible which is one of our main objectives.

4.5.3 Output of the Code

On executing the main Java file the output produced is shown in Figure 4.45. TestSuite is the root element in this file. Each ‘testSuite’ element can have one or more children test elements. Each element ‘Test’ has attributes ‘CategoryName’, ‘IPCName’, ‘Name’ and mode; and children elements ‘Inputs’, ‘Outputs’ and ‘Run’. Details of the output are given in section 5.2


```
  <?xml version="1.0"?>
- <testSuite>
  - <Tests class="java.util.ArrayList">
    - <test CategoryName="Exterior Lighting" IPCName="L405(8.6)" Name="SRS_EXTLG_733-735" mode="Create">
      - <Inputs class="java.util.ArrayList">
        - <input Delay="1000">
            <Srsid>SRS_EXTLG_733</Srsid>
            <Value>6</Value>
            <Path>Model Root/MSCANCOMMS/User2RTICANMMMainBlock/BCM_F/PowerMode/Value</Path>
            <Desc>INTERNAL_POWER_MODE ==6</Desc>
          </input>
        - <input Delay="1000">
            <Srsid>SRS_EXTLG_734</Srsid>
            <Value>1</Value>
            <Path>Model Root/HSCAN_COMMS/User2RTICANMM/MainBlock/IPC_BCM_E/FrontFogLightIndicationHS_HS/Value</Path>
            <Desc>CAN Signal FrontFogLightIndicationHS_HS(IPC_BCM_E is 1 (TurnTellTaleOn</Desc>
          </input>
        </Inputs>
      - <Outputs class="java.util.ArrayList">
        - <output>
            <Srsid>SRS_EXTLG_735</Srsid>
            <ExpectedValue>>=95</ExpectedValue>
            <Variable>IND_FRONT_FOG</Variable>
            <Desc>IND_FRONT_FOG shall be displayed</Desc>
          </output>
        </Outputs>
        <RunTest>false</RunTest>
      </test>
    - <test CategoryName="Brake Warning" IPCName="L405(8.6)" Name="SRS_BRAWAR_079-083" mode="Create">
      - <Inputs class="java.util.ArrayList">
        - <input Delay="1000">
            <Srsid>SRS_BRAWAR_093</Srsid>
            <Value>1</Value>
            <Path>Model Root/HSCAN_COMMS/Triggering2RTICANMM/MainBlock/ABS_P_29B/ABS_P_29B_Enable/Value</Path>
            <Desc>abc</Desc>
          </input>
        - <input Delay="1000">
            <Srsid>SRS_BRAWAR_093</Srsid>
```


Figure 4.45*: Output Generated by executing main java File*

4.6 XML Plugin for DSL output in JLR Project

A plugin has been written for manipulating output of the DSL for the JLR project. Using this plugin ViBATA can read test cases in the XML file and add them to database and run them. This plugin is a class library project named as XMLplugin as shown in Figure 4.46

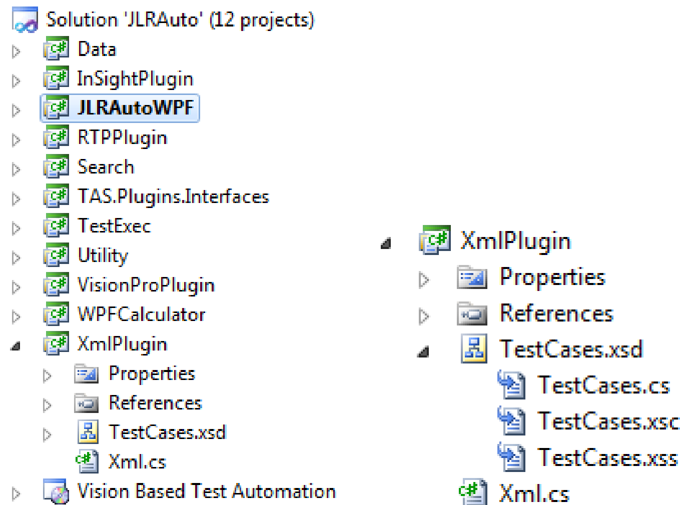

Figure 4.46*: XMLPlugin for JLR Poject*

This plugin consists of a generated xml schema named as TestCases.xsd which exposes a class act as an object to work on data provided by XML file; a class named Xml.cs which is used by a presentation layer to define the XML file to work on and manipulate data using schema class. From user interface user browses and selects XML file generated from DSL which saves all the tests, categories and IPC information defined in XML file if new and updates if already existed. If RunTest attribute of a test is set to true it checks the checkbox next to the test name. Figure 4.47 and Figure 4.48 show user interface in ViBATA

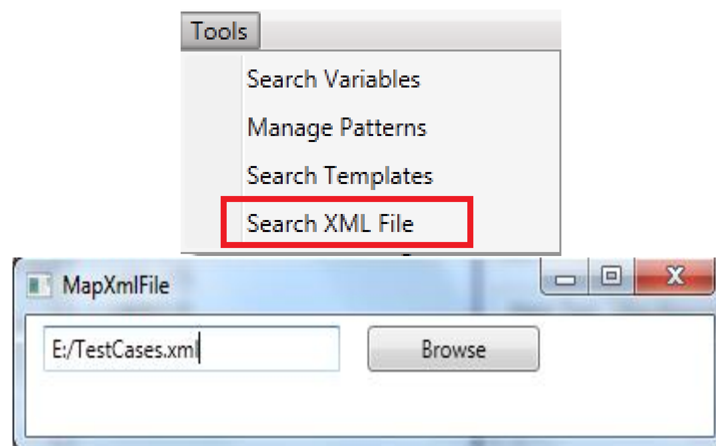

Figure 4.47: *Searching and browsing for XML file into the system*

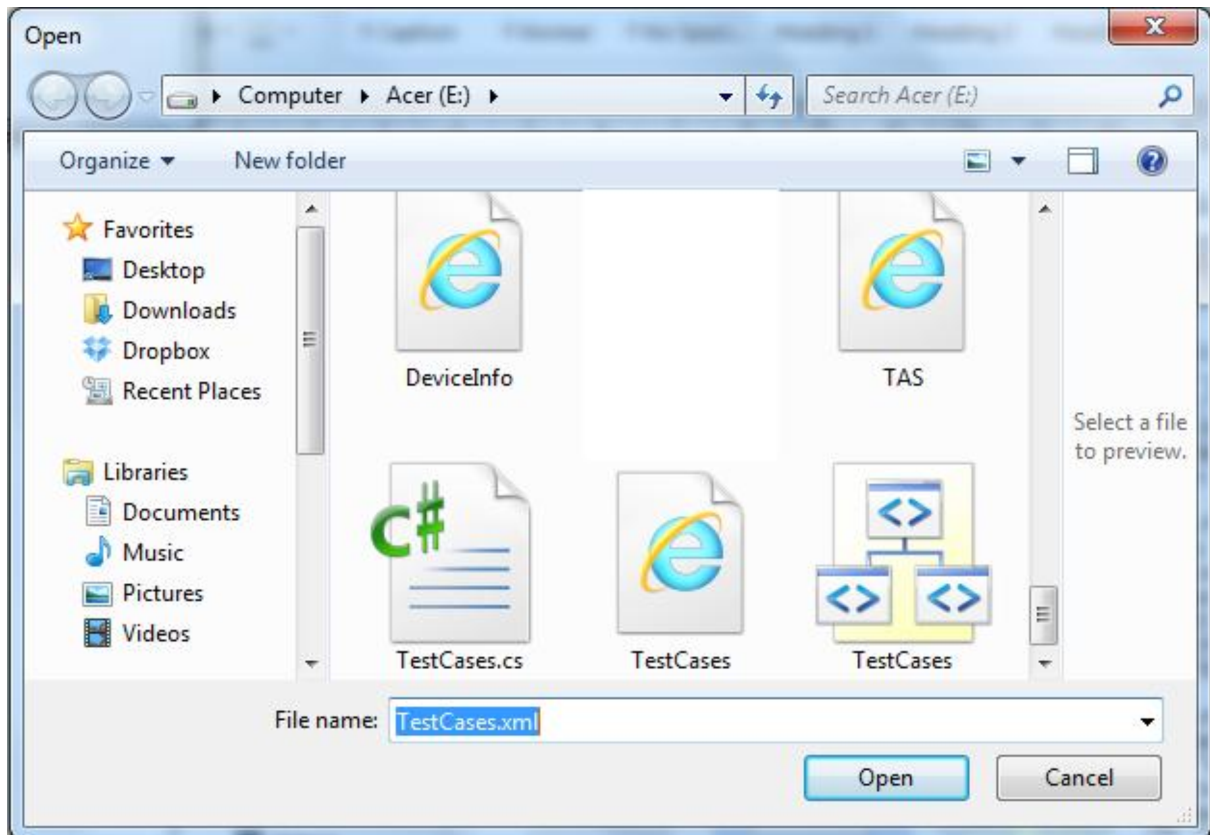

Figure 4.48***: User Interface of ViBATA to choose generated .xml file***

In this chapter an introduction to previous testing procedures used at JLR are described. An overview of ViBATA and DSL with full implementation of development stages of DSL using technologies defined in chapter 3 are illustrated to meet the objectives defined in chapter 1. Methodologies are applied to build the syntax of DSL and do its validation; code is generated from program in DSL which can be executed to give the desired output.

5 Results, Analysis and Discussion

In this chapter use cases of this DSL are illustrated and validated. A use case is a list of steps taken by the user interacting with software to achieve a goal. For each use case an introduction, DSL script, generated script, integration with ViBATA and result is given. Research questions and implications are also parts of this chapter.

5.1 Use Cases

Use cases are to check if objectives set in the beginning are met. In this section list of use case definition is given only. Why these are chosen and comparison of each use case with the objective is given in next section.

1. User can define the environment he is going to work in such as device information e.g. for camera and Controldesk.
2. He can define the initial setup of the test
3. He can define the test case with the information about Category and IPC it is in and the inputs and outputs it contains.
4. He can send instructions to create, delete and update a test
5. He can instruct to run the test case by defining the test name in specific Category of specific IPC
6. He can serialize the test cases he wants by giving instructions

5.2 Validation of Use Cases

In this section use cases are defined according to objectives and validated. Use case number 1, 3 and 4 are accommodated in section 5.2.1 because building of these use cases in DSL is related. Use case 5 and 6 are validated in section 5.2.3 and 2 is described in section 5.2.2.

5.2.1 Define Environment and Test Case

Introduction

Different devices together make the environment of the software. Our first objective was to build a DSL to provide the domain user a facility to define,

update and delete test cases and information about device used. That is why use cases number 1, 3 and 4 are set to achieve this objective.

In ViBATA each of devices has a plugin developed in the software and has certain configuration settings describe in the XML file named as TAS.config. A class Config.cs read these configurations and supply when it comes to establish a connection between ViBATA and the device. For example TAS.config file has connection settings for Insight camera which include settings for host, port, username and password as shown in Figure 5.1.

```
TASconfig.xml
1  <?xml version="1.0" encoding="utf-8" ?>
2  <settings>
3    <sensor>insight</sensor> <!-- insight, visionpro -->
4    <testdriver>tas</testdriver> <!-- tas, mxvdev -->
5    <sdf>C:\stuart\ControlDeskFiles\L405_VS_C_Test\vision_model_l405.sdf</sdf>
6    <simulator>rtp</simulator> <!-- rtp, vector -->>
7    <board>ds1005</board>
8    <extension>PPC</extension>
9    <sensorhost>138.250.81.16</sensorhost>
10   <sensorport>23</sensorport>
11   <sensoruser>admin</sensoruser>
12   <sensorpw></sensorpw>
13   <standalone>yes</standalone> <!-- Yes, No -->
14 </settings>
15
```

Figure 5.1*: TAS.config file in ViBATA*

In ViBATA test case is defined in Test Configuration Manager (TSM) section of the software. User copies the test case from Excel sheet and paste on this section which can be saved into the database by clicking ‘Save’ button as showed in the section 4.2 of this thesis. This is how test case environment and test case definition works in ViBATA. In next sections the same task is done through DSL is shown.

DSL Script

In this script Device and Test entities are declared with features. Each feature has a node (Attr, Ele, ELelist). DeviceSuite and TestSuite are declared with

DeviceInfo and TestCase inside with variables to set the features for inputs and outputs and for case itself.

Figure 5.2 is showing the implementation of DSL script

```
version15.catt    Device.java

Device myDevice{
        Name : String Ele
        Host : String Ele
        Port : String Ele
        User : String Ele
        Pwd : String Ele
        }
DeviceSuite devSuite{
    Create DeviceInfo Camera{
        my.Name = "Insight"
        my.Host= "138.250.81.16"
        my.Port="23"
        my.User ="admin"
        my.Pwd ="pwd"
        me.add = true
    }
}
```

```
Test thisTest{
Name : String  Attr
IPCName : String    Attr
CategoryName : String  Attr
Inputs List of myInput EleList
Outputs List of myOutput EleList
RunTest : Boolean  Ele
}
TestSuite mysuite{
    Create TestCase thiscase{
    Declare Ins List of myInput
    Declare Outs List of myOutput
    Declare Input1 : myInput
    Declare Input2 : myInput
    Declare Output1 : myOutput
    my.Name ="SRS_EXTLG_740-745"
    my.IPCName="L405(8.6)"
    my.CategoryName="Exterior Lighting"
    my.RunTest= true
```

Figure 5.2***: DSL script for defining the Device and Test***

Keyword ‘Create’ sets ‘mode’ attribute of a DeviceInfo/TestCase to set ‘mode’ to ‘Create’ in XML file which will tell ViBATA to create new DeviceInfo/TestCase in the system. Using DeviceInfo/TestCase for each Device/Test can set its feature values. To set a feature value for Case **my** keyword is used and to include a case in suite **me** keyword is used both keywords showed in rules section 4.4.2. The reason of using these keywords is to avoid declaration of variable of type Case to set its features. When Case is defined an instance of type case is declared in Java. In Figure 5.2 a ‘DeviceInfo’ case is setting all the features of the device to a value for example in this case it is setting feature values for ‘Insight’ camera such as Name, Host, Port, User and Password. And for the TestCase it is setting its name, category and IPCName. In ‘Case’ declaration when a feature is of type another entity then a variable declaration of that type is needed to set its feature’s value. As in case of TestCase two kinds of variables of type Input are declared one of which is list variable and other is single. Then values are assigned to single variable’s features. Once that is

done, single variable is added to the list variable and then it is assigned to list feature of the test for example in this case variable 'Ins' is assigned to 'Inputs' feature of TestCase as shown in Figure 5.3

```
TestSuite mysuite{
Create TestCase thiscase{
 Declare Ins List of myInput
 Declare Outs List of myOutput
 Declare Input1 : myInput
 Declare Input2 : myInput
 Declare Output1 : myOutput
Input1.Desc = "INTERNAL_POWER_MODE ==6"
Input1.Srsid='SRS_EXTLG_733'
Input1.Value="6"
Input1.Delay="1000"
Input1.Path="Model Root/MSCANCOMMS/User:
Input2.Desc="CAN Signal FrontFogLightIn
Input2.Srsid="SRS_EXTLG_734"
Input2.Value="1"
Input2.Delay="1000"
Input2.Path="Model Root/HSCAN_COMMS/Use
Output1.Srsid="SRS_EXTLG_735"
Output1.ExpectedValue=">=95"
Output1.Variable="IND_FRONT_FOG"
Output1.Desc="IND_FRONT_FOG shall be di
Ins.add(Input1)
Ins.add(Input2)
Outs.add(Output1)
my.Inputs= Ins
my.Outputs = Outs
me.add = true
```

Figure 5.3*: Assignments to single and list variables in TestCase*

Generated Code

In this section, code generated from Figure 5.3 is described and shown. Device entity generates a file Device.java and Test entity generates Test.java. Device/Test.java files contains two java classes one DevSuite/TestSuite and other Device/Test. These classes are having annotation of @Root which will show them at root level in output XML file. Device/Test class contains getter and setter for all the features with annotation above defined in node attribute for it. For example if feature is ending with attribute 'Ele' it will have an annotation of @Element above it. DevSuite/TestSuite will have only one feature which is list of devices/tests. Device.java and Test.java classes are shown in Figure 5.4.

```
@ElementList

private List<Output> Outputs;
public List<Output> getOutputs() {
return Outputs;
}
public void setOutputs(List<Output> Outputs) {
this.Outputs = Outputs;
}

@Element

private Boolean RunTest;
public Boolean getRunTest() {
return RunTest;
}
public void setRunTest(Boolean RunTest) {
this.RunTest = RunTest;
}
}
```

```
Test.java
package cranfield;
import org.simpleframework.xml.Attribute;
@Root
class TestSuite{

@ElementList

private List<Test> Tests;
public List<Test> getTests() {
return Tests;
}
public void setTests(List<Test> Tests) {
this.Tests = Tests;
}
}
@Root
public class Test{

@Attribute
private String mode;
public String getMode() {
return mode;
}
public void setMode(String mode) {
this.mode = mode;
}
```

```
Device.java
package cranfield;
import org.simpleframework.xml.Attribute;
@Root
class DevSuite{

@ElementList

private List<Device> Devices;
public List<Device> getDevices() {
return Devices;
}
public void setDevices(List<Device> Devic
this.Devices = Devices;
}
}
@Root
public class Device{

@Attribute
private String mode;
public String getMode() {
return mode;
}
public void setMode(String mode) {
this.mode = mode;
}
```

Figure 5.4*: Device/Test.java files Generated from Entity Device/Test*

Figure 5.5 shows the code in main() function for the above code stub in DSL. It creates an instance of Serializer class from Simple framework, and of class DeviceSuite and TestSuite. Creates an object of type class Device and Test and calls and creates methods buildDeviceInfoCamera () and buildTestCasethiscase() which returns new instance of Device/Test class. The code in main function and other functions for both cases DeviceInfo/TestCase is shown in Figure 5.5.

```java
public static void main(String[] args) {
        Serializer serializer = new Persister();
        DeviceSuite devSuite = new DeviceSuite();
        List<Device> devices = new ArrayList<>();
        Device Camera = buildDeviceInfoCamera();
                devices.add(Camera);
        devSuite.setDevices(devices);
        TestSuite mysuite = new TestSuite();
        List<Test> tests = new ArrayList<>();
        Test thiscase = buildTestCasethiscase();
            tests.add(thiscase);
        Test thiscase1 = buildTestCasethiscase1();
            tests.add(thiscase1);
        mysuite.setTests(tests);
        File loc = new File("E:/TestCases.xml");
        File deviceLoc = new File("E:/DeviceInfo.xm
        try{
        serializer.write(devSuite,deviceLoc);
        }catch(Exception ex){
         }
        try{
        serializer.write(mysuite,loc);
        }catch(Exception ex){
         }
}

public static Device buildDeviceInfoCamera(){
Device Camera = new Device();
Camera.setMode("Create");
Camera.setName("Insight");
Camera.setHost("138.250.81.16");
Camera.setPort("23");
Camera.setUser("admin");
Camera.setPwd("pwd");
return Camera;
}
```

```java
public static Test buildTestCasethiscase(){
Test thiscase = new Test();
List<Input> Ins = new ArrayList<>();
List<Output> Outs = new ArrayList<>();
Input Input1 = new Input();
Input Input2 = new Input();
Output Output1 = new Output();
thiscase.setMode("Create");
thiscase.setName("SRS_EXTLG_740-745");
thiscase.setIPCName("L405(8.6");
thiscase.setCategoryName("Exterior Lighting");
thiscase.setRunTest(true);
Input1.setDesc("INTERNAL_POWER_MODE ==6");
Input1.setSrsid("SRS_EXTLG_733");
Input1.setValue("6");
Input1.setDelay("1000");
Input1.setPath("Model Root/MSCANCOMMS/User2RTICANMMM
Input2.setDesc("CAN Signal FrontFogLightIndicationHS
Input2.setSrsid("SRS_EXTLG_734");
Input2.setValue("1");
Input2.setDelay("1000");
Input2.setPath("Model Root/HSCAN_COMMS/User2RTICANMM
Output1.setSrsid("SRS_EXTLG_735");
Output1.setExpectedValue(">=95");
Output1.setVariable("IND_FRONT_FOG");
Output1.setDesc("IND_FRONT_FOG shall be displayed");
    Ins.add(Input1);
    Ins.add(Input2);
    Outs.add(Output1);
thiscase.setInputs(Ins);
thiscase.setOutputs(Outs);
return thiscase;
}
```

Figure 5.5*: Code for Device and Test cases in main File*

Output and Integration with ViBATA

The execution of the main java file generates two XML files which are shown in Figure 5.6 and Figure 5.7*.* The XML file for DeviceSuite replaces TAS.config and TestSuite is read by the Xml.cs in XmlPlugin of ViBATA to perform declared tasks mentioned.

```xml
  <?xml version="1.0"?>
- <devSuite>
  - <Devices class="java.util.ArrayList">
    - <device mode="Create">
        <Name>Insight</Name>
        <Host>138.250.81.16</Host>
        <Port>23</Port>
        <User>admin</User>
        <Pwd>pwd</Pwd>
      </device>
    </Devices>
  </devSuite>
```

Figure 5.6*: Xml File for Device Suite*

```xml
<?xml version="1.0"?>
<testSuite>
 - <Tests class="java.util.ArrayList">
   - <test CategoryName="Exterior Lighting" IPCName="L405(8.6" Name="SRS_EXTLG_740-745" mode="Create">
     - <Inputs class="java.util.ArrayList">
       - <input Delay="1000">
           <Srsid>SRS_EXTLG_733</Srsid>
           <Value>6</Value>
           <Path>Model Root/MSCANCOMMS/User2RTICANMMMainBlock/BCM_F/PowerMode/Value</Path>
           <Desc>INTERNAL_POWER_MODE ==6</Desc>
         </input>
       - <input Delay="1000">
           <Srsid>SRS_EXTLG_734</Srsid>
           <Value>1</Value>
           <Path>Model Root/HSCAN_COMMS/User2RTICANMM/MainBlock/IPC_BCM_E/FrontFogLightIndic
           <Desc>CAN Signal FrontFogLightIndicationHS_HS(IPC_BCM_E is 1 (TurnTellTaleOn</Desc>
         </input>
       </Inputs>
     - <Outputs class="java.util.ArrayList">
       - <output>
           <Srsid>SRS_EXTLG_735</Srsid>
           <ExpectedValue>>=95</ExpectedValue>
           <Variable>IND_FRONT_FOG</Variable>
           <Desc>IND_FRONT_FOG shall be displayed</Desc>
         </output>
       </Outputs>
       <RunTest>true</RunTest>
     </test>
```

Figure 5.7*: Xml Output for Test Suite*

The Xml.cs in ViBATA checks the IPC, Category and mode of the Test Case. If mode is create/update it checks for test case name in the system if present it updates the test otherwise create new one. If the mode of Test is 'Delete' it deletes the test case for the category in IPC. The snapshot of code in this class is shown in Figure 5.8

```csharp
if (test.mode == "Create" || test.mode == "Update")
{
DataAccess.StoreIPCFromDSL(ipcID, ipcModel, ipcModelYear, ipcSdf, ipcNotes);
DataSetTest.tblIPCDataTable newipc = DataAccess.GetIPC(test.IPCName);
ipcID = (int)newipc.Rows[0]["ID"];

//DataAccess.StoreTest(ipcID,
catID = DataAccess.StoreCategoryFromDSL(ipcID, catID, test.CategoryName);
testID = DataAccess.StoreTest(ipcID, catID, test.Name);
DataAccess.updateActiveForTest(testID, Convert.ToBoolean(test.RunTest));
_runtest = new RunTest();
_runtest.Run = Convert.ToBoolean(test.RunTest);
_runtest.TestID = testID;
newlist.Add(_runtest);
List<InputData> inputsTAS = new List<InputData>();
List<testSuiteTestsTestInputsInput> inputs = s.DisplayTestInputs(test);
List<testSuiteTestsTestOutputsOutput> outputs = s.DisplayTestOutputs(test);
int order = 0;
foreach (testSuiteTestsTestInputsInput input in inputs)
{
    int next = DataAccess.StoreDVPLine(testID, input.Srsid, input.Desc, "Input", -1, ipcID);
    InputData inputTAS = new InputData();
    inputTAS.Path.Name = input.Path;
    inputTAS.SRSID = input.Srsid;
    inputTAS.Value = Convert.ToDouble(input.Value);
    DataAccess.StoreInputLine(ipcID, testID, order++, inputTAS);

}
```

```csharp
else if (test.mode == "Delete")
{
    if (ipcID != -1)
    {
        Data.DataSetTestTableAdapters.tblTestTableAdapter tbt = n
        DataSetTest.tblTestDataTable tblTest = tbt.GetID(ipcID, t

        foreach (DataSetTest.tblTestRow r in tblTest)
        {
            testID = r.ID;
            DataAccess.DeleteTest(testID);
        }
    }
}
```

Figure 5.8*: Xml.cs in ViBATA*

Creation of Test Case through ViBATA by reading XML file

In this section, test case creation is illustrated in ViBATA through the XML output obtained from DSL. In Figure 5.9 it is showed that category 'Exterior

Lighting' is not present for IPC named L405(8.6) and ultimately no test case for this category is present.

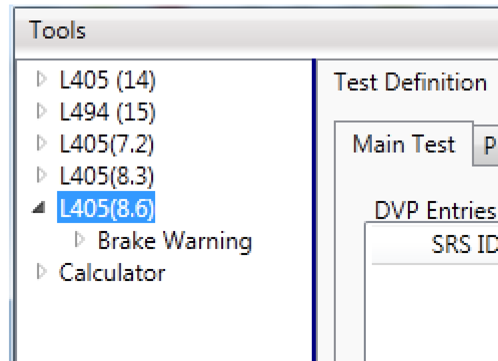

Figure 5.9*: Category Exterior Lighting is not present for the IPC*

The output from DSL is shown in Figure 5.10 which has mode 'Create' with Category/IPC Name and inputs/outputs for the Test with RunTest element set to 'true'.

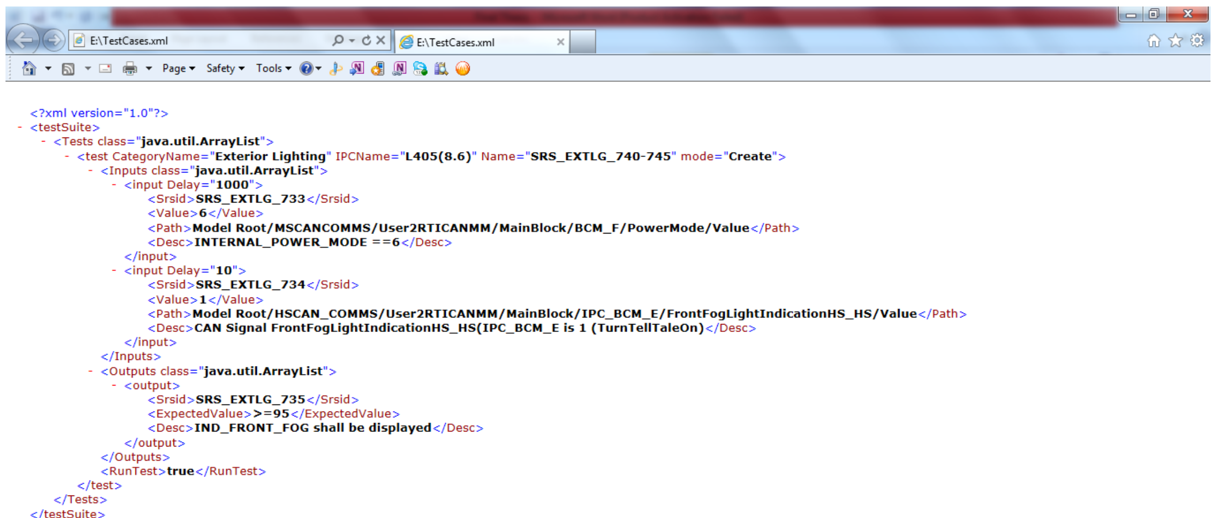

Figure 5.10*: The DSL output to insert Test Case into ViBATA*

In *figures* Figure 5.11 and Figure 5.12 it is shown how on single click of choosing the XML file from ViBATA creates Category and Test Case within IPC and checks the checkbox next to it. To select the XML file click on Tools on main screen. On dropdown click on option 'Search Xml File'. A window will

appear to browse the XML file into the system. Select the file and Click OK. The test case with its input lines and output lines is created. All is need to click on Test Execution tab to run the test to check if it works. There is code written to provide dummy values to inputs and outputs on the basis of which ViBATA decides whether test is passed or failed which is shown in Figure 5.13. The dummy values are provided because of the absence of actual hardware and Simulink model.

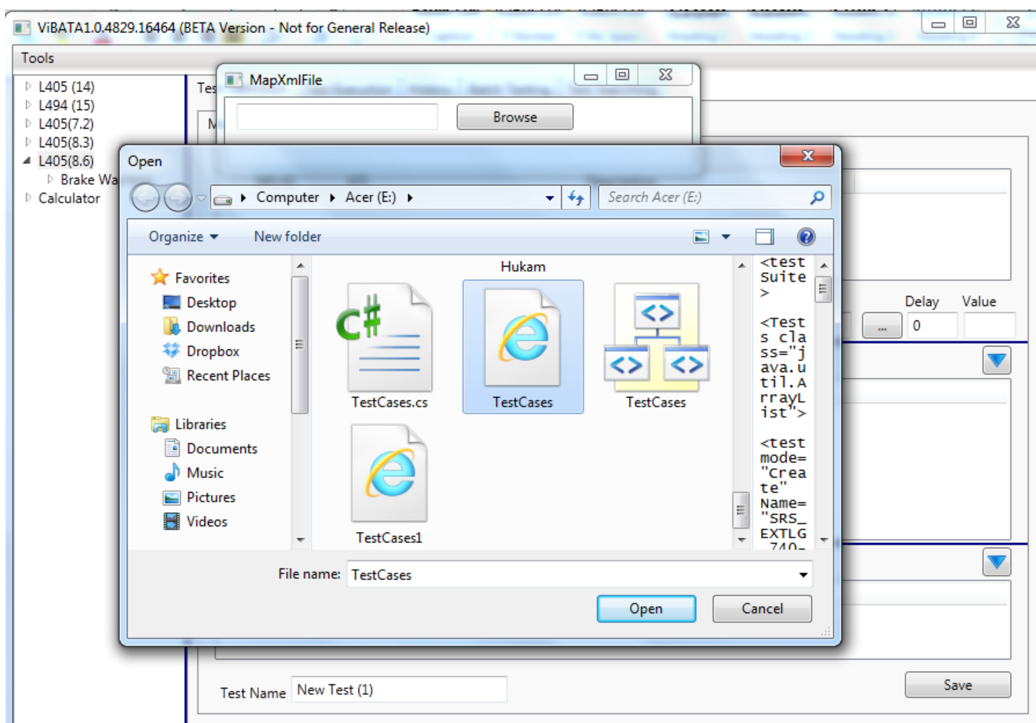

Figure 5.11*: Choosing TestCases XML file from ViBATA*

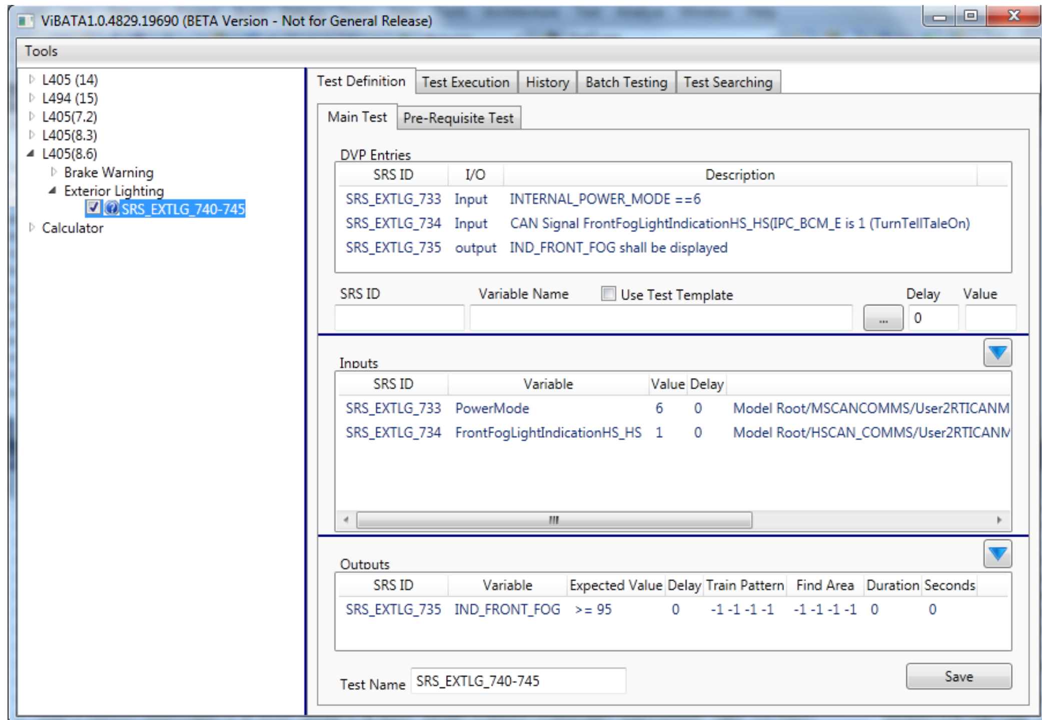

Figure 5.12*: Test creation on single click from ViBATA*

```
public static int returnPercentage(ObservableCollection<InputData> Inputs)
{
    Random rnd = new Random();
    int num,count=0;
    List<int> myints = new List<int>();
    myints.Add(0);
    myints.Add(4);
    myints.Add(6);
    myints.Add(9);
    foreach (InputData input in Inputs)
    {
        if (input.VarName.Contains("Power"))
        {
            if (myints.Contains(TryConvert.ToInt32(input.Value
            {
                count++;
            }
        }
        else if (input.VarName.Contains("Fog"))
        {
            if (input.Value == 1)
            {
                count++;
            }
            else
            {
                count = 0;
```

```
public static ObservableCollection<ResultData> DummyExecTest(ObservableCollection<InputData>
{
  ObservableCollection<ResultData> resList = new ObservableCollection<ResultData>();
      foreach (OutputData output in Outputs)
      {

              ResultData res = new ResultData();
              res.SRSID = output.SRSID;
              res.OutputID = output.ID;
              res.Expected = output.ExpectedValue;
              string t;


         if (output.HasPattern)
        {
         int num = returnPercentage(Inputs);
         t = num.ToString();
         res.Actual = TryConvert.ToDouble(t);

         //res.PassFail = Passed(output.ExpectedValue, res.Actual);
         res.PassFail = Expression.Evaluate(res.Actual, output.Expression, output.ExpectedValu

         resList.Add(res);
         }
         else
         {//TO DO: READ VARIABLE ETC...
         }
```

Figure 5.13*: Code stub with dummy values for Input and Output in Test case*

The result of test execution on Test execution tab after clicking Run Test button is shown in Figure 5.14

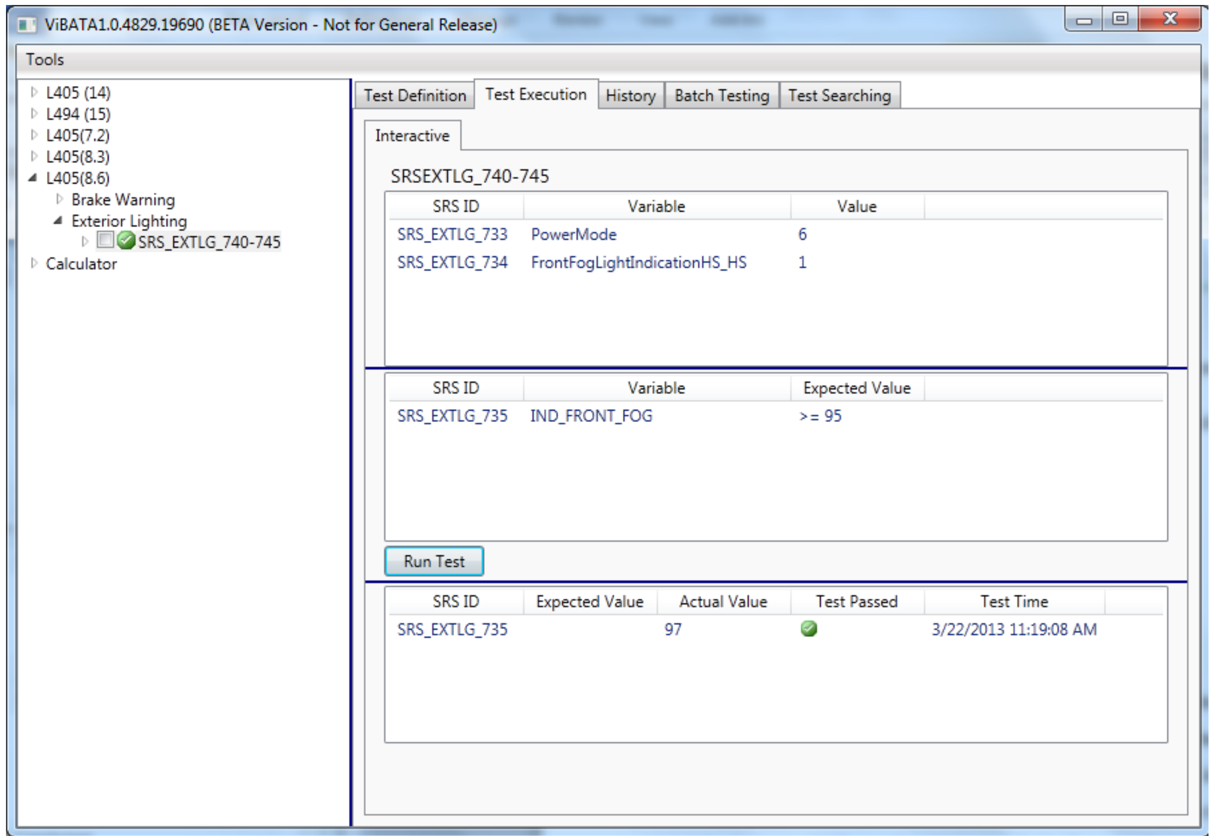

Figure 5.14*: Result shown on Test Execution Tab*

5.2.2 Define Test Setup

One of our objectives is to define the test setup. A test setup is set of instructions sent to IPC to put IPC in a certain condition before test case is run. Defining a test setup was part of use cases and objectives but not included in the DSL for two reasons. The first reason is the purpose of this DSL is to only implement the part of ViBATA which specify test cases (TSM) not running the test cases from DSL. For those test cases which require a setup a desired output is needed in return which decides the running of a test case or values of certain inputs or outputs. This desired output cannot be read by the DSL. For some test cases setups only require an instruction to put the system in certain condition and not output returned. These instructions are actually inputs into system. A test case in the DSL already consists of inputs so the setup instructions can be part of these test inputs. Secondly, this DSL is made in a general way so that it can accommodate most of the embedded systems which will not be accomplished by defining the setup as part of DSL.

5.2.3 Defining XML File and Location

Introduction

It is one of our objectives to generate consistent output readable by ViBATA. The XML is chosen because of this reason because it is a standard for interoperability. So the last use case defined is to allow serialization of desired test cases is achieved by specifying which suites should be part of generated XML file, how many files should be generated, where this file should be saved and what the name of the file is.

DSL Script

The rules for defining this in Xtext are shown in Figure 5.15

```
Communication:
    Serialize|DeSerialize
;
Serialize:
    'Write XML File' name = ID
    '{'
      (file += File)*
      'run' '(' command += RunCommand(';' command+=RunCommand)? ')'
    '}'
;
RunCommand:
    test=[Suite]','filename=[File]
;
File:
    'FileLocation' name=ID '=' destination= STRING
;
```

Figure 5.15*: Rules for Serialization*

These rules define how serialization will declare in DSL program. The 'Communication' rule is part of top rule AbstractElement. The rule 'Serialize' contains keywords 'Write XML File' then file feature calls rule 'File' to declare one or more 'FileLocation' and 'Destination'. The command feature calls rule 'RunCommand' starting with keyword 'run' this rule shows that command can either be one or many. The DSL stub for serialization is shown in Figure 5.16

```
Write XML File myFile{
    FileLocation loc = "E:/TestCases.xml"
    FileLocation deviceLoc = "E:/DeviceInfo.xml"
        run (devSuite,deviceLoc;mysuite,loc)
    }
 }
```

Figure 5.16*: DSL script for Serialization*

Generated Code

As defined earlier, for serialization this DSL is using Simple framework which uses Serializer to generate XML. Because this is necessary for the output generation the instance of this Serializer is declared in the beginning of the main() function through template expression. When DSL program contains the code for 'Serialization' and defines the file name and location then write() method of Serializer is called. In *figure* Figure 5.17and Figure 5.18 shows the code in Xtend and generated code in Java respectively.

```
public static void main(String[] args) {
        Serializer serializer = new Persister();
«FOR createfile:dm.eAllContents.toIterable.filter(typeof (Serialize))»
«FOR file:createfile.file»
    File «file.name» = new File("«file.destination»");
«ENDFOR»
«FOR command: createfile.command»
 try{
 serializer.write(«command.test.name»,«command.filename.name»);
 }catch(Exception ex){
  }
«ENDFOR»
```

Figure 5.17*: Code in Xtend for rules for Serializer*

```
File loc = new File("E:/TestCases.xml");
File deviceLoc = new File("E:/DeviceInfo.xml");
try{
serializer.write(devSuite,deviceLoc);
}catch(Exception ex){
 }
try{
serializer.write(mysuite,loc);
}catch(Exception ex){
 }
```

Figure 5.18*: Code generated in Java*

The output of this program is already shown in first use case

Result

The previous approach (ViBATA) consists of long excel sheet and test cases are entered into the system by copying test cases from this excel sheet on to the system. As compared to the previous approach defining a test case through DSL is made simpler. Using ViBATA interface on selecting XML file new IPC, Category and Test case is created in single click. If run attribute of test case is true the checkbox next to it get selected as well. If we consider applying both approaches for couple of thousands of test cases, DSL approach takes less time to define them. In the previous approach operator had to check manually if test case is created already in a particular category of a particular IPC. Then he had copy and paste the test case from excel sheet to software interface and save it. DSL uses a declarative approach which tells what should happen rather than how it should happen. It spares the DSL user from thinking about what is happening behind the scene. It provides a limited functionality which is easier to grasp for person in domain. Using search and find technique in DSL would make a lot easier for user to change the details of test cases. For example same test cases can be defined for different models with little detail change like name of IPC and all the cases created for one model can be created for another model in single click. In case there are thousands of tests which need to be run in a batch mode of software using DSL only ‘RunTest’ feature of a test case will need to set to ‘True’ by search and find technique. And with single click it will check the checkboxes in front of all those test case irrespective of their Category or IPC whereas in previous approach test cases will need to be ticked one by one if they belong to different IPCs or Categories.

5.3 Research Questions

1. What are the characteristics of a DSL for testing embedded systems?

The sole purpose of this DSL is to automate test case specification for testing in embedded system. The structure of this DSL is made more general so it can accommodate test case specification for all embedded systems. For these reasons keywords common to a test case in testing environment are used such as Input, Output, Test, Device, TestCase, and TestSuite. User is made restricted to use these elements which will allow consistency in subsequent releases. A test case consists of some inputs and outputs which will have certain features. For the current study the features of an Input are 'SRSID', 'Name', 'Value', 'Description'. For a different embedded system these features will be different. For example if we consider a calculator as another embedded system, the features of an input might be 'ID', 'Description' and 'Number' only. A 'Test' in a calculator will need to define an operation on numbers which will be its own feature in addition to its Name and Description. Similarly for a room controller device with temperature sensor the features of an 'Input' would be 'Power', 'Temperature', and 'On. For an output there will always be a 'Description', and 'Expected value' to compare it with the actual value in test Oracle. According to every embedded system there will be some inputs and outputs for tests or may be only tests with its own features. If DSL is extended according to other embedded systems the target platform specifications will need to be amended. The generated file will always be in XML. The xml schema will need to be generated according to target platform specifications and plugin will need to be written in order to manipulate XML file. So the DSL can specify test cases for all embedded system. The DSL is tested for a calculator in section 5.4.

2. What do we need to extend it to specific environment i.e. automotive?

The DSL can be extended for a specific environment like automotive by introducing detailed information on Inputs and Outputs as is done for all example programs written using current DSL. JLR uses a certain 'Path' for an

Input and Pattern/Variables for an ‘Output’. The ‘Path’ for an Input is not given in actual test specification. For this DSL ‘Path’ is used because for the test case used an example to demonstrate the use of DSL in automated testing ‘Path’ for inputs are known. So the DSL example shown in section 5.2.1 is particular to the JLR system. In this example the Paths to inputs are provided. But in actuality these paths are not known and there is a need to search this path through ViBATA functionality in file with extension .sdf in ControlDesk as explained in section 4.2. This DSL can be extended by defining ‘Path’ as separate entity with features and linked to Input. An instruction can be given to search for the path by supplying keywords if path is found it should be used for Input to enter into database. Same way there is pattern defined for ‘Output’. A pattern could be added as additional entity with certain features. There are some special test cases for JLR which need a Pre-Requisite Test. This pre-requisite test is attached to a certain input. In ViBATA a test case is created first and then a pre-requisite test is defined for an input. The output of the pre-requisite test decides the value of the input to which it is attached. The DSL can be extended to accommodate this functionality as well.

5.4 Implications

In this section, some implications are defined for the current system. And what would need to be done under such circumstances.

5.4.1 Can this DSL work with other embedded system?

To test if DSL works for other embedded system, a simple calculator is tested using this DSL and ViBATA software. This software application for calculator is chosen because it is simple and easy to use, developed as a WPF application like ViBATA and available at [39]. The calculator application will be tested from ViBATA and it needs to be saved in ViBATA database. So for this application, features of entities in DSL are not changed according to ‘Calculator’ domain

which can be changed as explained in section 5.3. Only ‘TestCase’ part of this Calculator DSL is shown in Figure 5.19

```
TestSuite calsuite{
Create TestCase thiscase{
Declare Ins List of myInput
Declare Outs List of myOutput
Declare Input1 : myInput
Declare Input2 : myInput
Declare Output1 : myOutput
my.Name ="SRS_EXTLG_740-745"
my.IPCName="Calculator"
my.CategoryName="Addition"
my.RunTest= true
Input1.Desc = "Number"
Input1.Srsid='SRS_EXTLG_733'
Input1.Value="6"
Input1.Delay="1000"
Input1.Path="Model Root/MSCANCOMMS/User2RTICANMM/MainBlock/BCM_F/Pow
Input2.Desc="Second Number"
Input2.Srsid="SRS_EXTLG_734"
Input2.Value="4"
Input2.Delay="1000"
Input2.Path="Model Root/HSCAN_COMMS/User2RTICANMM/MainBlock/IPC_BCM_
Output1.Srsid="SRS_EXTLG_735"
Output1.ExpectedValue="=10"
Output1.Variable="var1"
Output1.Desc="Add numbers to get Expected Value"
Ins.add(Input1)
Ins.add(Input2)
Outs.add(Output1)
my.Inputs= Ins
my.Outputs = Outs
me.add = true
}
```

Figure 5.19*: TestCase for Calculator DSL*

In Figure 5.19 the IPC for this test case is defined as ‘Calculator’ and ‘Category’ as ‘Addition’. The descriptions of the inputs are ‘Number’ and ‘Second Number’ and ‘Values’ are 6 and 4. Description of output is ‘Add numbers to get Expected value’ and ‘ExpectedValue’ is 10. The generated Xml file from this DSL is shown in Figure 5.20

```
<?xml version="1.0"?>
- <testSuite>
  - <Tests class="java.util.ArrayList">
    - <test CategoryName="Addition" IPCName="Calculator" Name="SRS_EXTLG_740-745" mode="Create">
      - <Inputs class="java.util.ArrayList">
        - <input Delay="1000">
            <Srsid>SRS_EXTLG_733</Srsid>
            <Value>6</Value>
            <Path>Model Root/MSCANCOMMS/User2RTICANMM/MainBlock/BCM_F/PowerMode/Value</Path>
            <Desc>Number</Desc>
          </input>
        - <input Delay="1000">
            <Srsid>SRS_EXTLG_734</Srsid>
            <Value>4</Value>
            <Path>Model Root/HSCAN_COMMS/User2RTICANMM/MainBlock/IPC_BCM_E/FrontFogLightIndicationHS_HS/Value</Path>
            <Desc>Second Number</Desc>
          </input>
        </Inputs>
      - <Outputs class="java.util.ArrayList">
        - <output>
            <Srsid>SRS_EXTLG_735</Srsid>
            <ExpectedValue>=10</ExpectedValue>
            <Variable>var1</Variable>
            <Desc>Add numbers to get Expected Value</Desc>
          </output>
        </Outputs>
        <RunTest>true</RunTest>
      </test>
    </Tests>
</testSuite>
```

Figure 5.20*: Output of Calculator DSL*

This XML file is read by ViBATA which created IPC ‘Calculator’ with category ‘Addition’ and test case in it with checkbox selected because RunTest attribute of test case is set to true. A small procedure is created in calculator application which takes input values and category from the ViBATA and computes the values of inputs and brings output back. Where ViBATA takes this output value and compares it with the expected value and decides if test is passed or failed. The Figure 5.21 shows the test case in ViBATA.

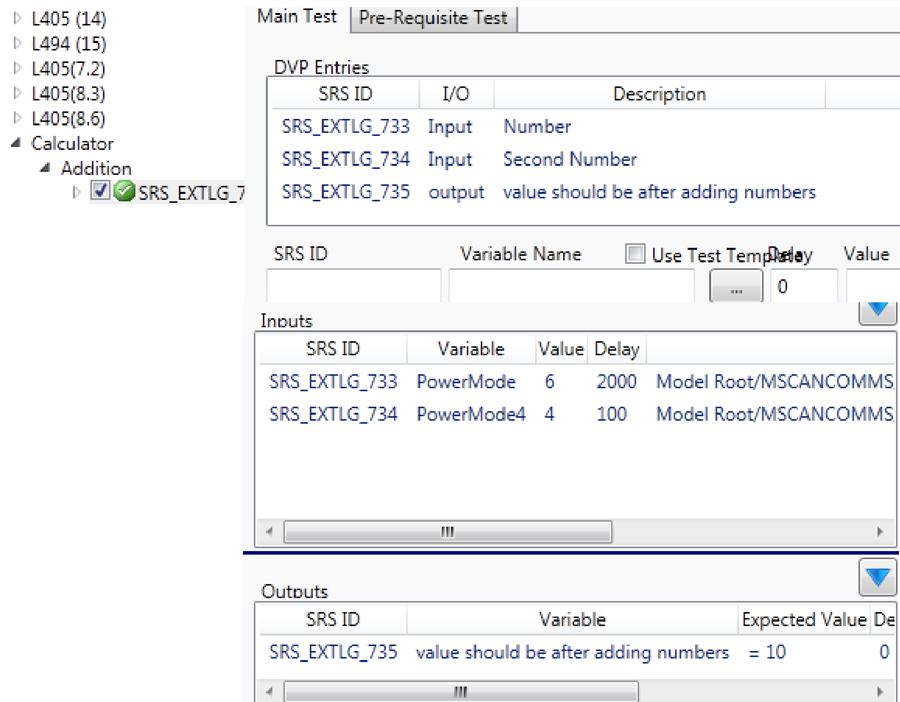

Figure 5.21*: Test case for Calculator in ViBATA*

A method is made in RunTest.cs of ViBATA named as DummyExecuteTest() which is a replacement of actual method to run test. For Catt DSL this method provides dummy values because of absence of Simulink model for actual hardware and SUT but for calculator it provides values from inputs and add them if category is Addition. The code of this method and result are shown in *figures* Figure 5.22, Figure 5.23 and Figure 5.24.

```
public static ObservableCollection<ResultData> DummyExecTest(ObservableCollection<InputData> Inputs, Observable
{
  ObservableCollection<ResultData> resList = new ObservableCollection<ResultData>();
      foreach (OutputData output in Outputs)
      {

              ResultData res = new ResultData();
              res.SRSID = output.SRSID;
              res.OutputID = output.ID;
              res.Expected = output.ExpectedValue;
              string t;

              if (IPCName == "Calculator")
              {
                  List<int> intlist = new List<int>();
                  foreach (InputData input in Inputs)
                  {
                      int val = (int)input.Value;
                      intlist.Add(val);

                  }
                  WPFCalculator.Window1 win = new Window1();
                  res.Actual = win.calculationFromXML(intlist, catName);
                  res.PassFail = Expression.Evaluate(res.Actual, output.Expression, output.ExpectedValue);
                  resList.Add(res);

              }
```

Figure 5.22*: DummyExecTest Function in RunTest.cs of ViBATA*

```
public double calculationFromXML(List<int> nums, string operation)
{
    //string [] ids = {num1,num2};
    for (int i=0;i<=nums.Count-1;i++)
    {
    string num = nums[i].ToString();
    char[] id1 = num.ToCharArray();
    ProcessKey(id1[0]);
    switch (operation)
    {
        case "Addition":
            ProcessOperation("BPlus");
            break;
        case "Subtraction":
            ProcessOperation("BMinus");
            break;
        case "Divide":
            ProcessOperation("BDevide");
            break;
    }
    }
    ProcessOperation("BEqual");
    double result =Convert.ToDouble(DisplayBox.Text);
    return result;
}
```

Figure 5.23*: Execution of Test case from ViBATA in Calculator Application*

SRSEXTLG_740-745

SRS ID	Variable	Value
SRS_EXTLG_733	PowerMode	6
SRS_EXTLG_734	PowerMode4	4

SRS ID	Variable	Expected Value
SRS_EXTLG_735	value should be after adding numb	= 10

Run Test

SRS ID	Expected Value	Actual Value	Test Passed	Test Time
SRS_EXTLG_735		10	✓	3/20/2013 5:50:

Figure 5.24*: Execution of Test case for Calculator*

Because of calculator validation through ViBATA major changes cannot be made in DSL for test specification such as change of features. So the difference between the DSL program for Catt and Calculator is the feature definition for entities input and output. For example for catt file full known path for input is given but for calculator it is just a string. When Xml.cs in ViBATA finds calculator.xml it doesn't enter this path for inputs. To validate test case for calculator only test case will be entered without input and output values. These will need to be selected from the stored templates in the database to run and validate the test case. For both cases it was needed to provide dummy values for inputs done by writing separate code stub.

5.4.2 What will happen if Device Changes

In case any of device changes for example Camera is changed from Insight to web cam. First of all, through DSL the device specific configuration will need to

be set. Second a plugin will be required to meet platform specific needs like how to connect the camera, how to capture image from camera, recognition of pattern of image, saving the pattern and comparing it with the one showed on actual device. For example with Insight camera software establishes a connection through TCP/IP. For web cam these connection settings will be different. Software comes with Insight camera exposes means of capturing the image, saving the image file and matching the pattern against the actual image to show the results.

5.4.3 What will happen if DSL program variable changes

If a variable changes in the DSL program that will not make any affect. For example consider *figure 5.2* the variable Input1 in the TestCase has type myInput which is name of entity Input. The serialized TestCase in Figure 5.7 has this Input1 as input because on serialization it takes the entity's classifier. So whatever name user chooses for entity's name identifier or type of declared variable's identifier the XML file will be consistent. Changing variables will have no effect on the resulting output.

5.4.4 What will happen if user selects an XML file having different elements

If user selects from ViBATA an XML file with different elements having root element other than TestSuite or it does not contain any Tests or he selects a different file like a text file then what will happen. This is the reason xml schema file in place which checks the formation of XML file. For XML file having different elements schema will not be able to match and exception will be thrown which will be caught and display a user friendly message in a message box as shown in Figure 5.25 if xml file does not contain any test it will check and display a message saying file does not contain any tests. For a different system with different domain elements the schema will need to be regenerated according to XML file.

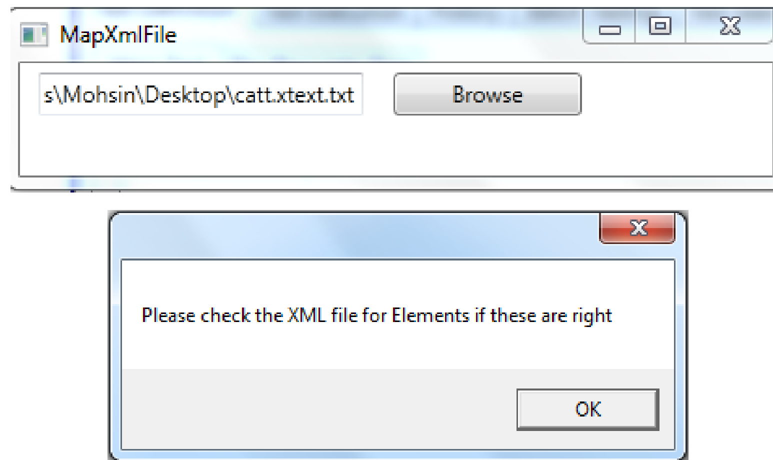

Figure 5.25*: Friendly message on choosing wrong file*

In this chapter use case are defined which are experiments to test the DSL to see if the required objectives are achieved. The detailed information about use cases and the reason to choose these is given. It is also explained how goals are achieved by testing all the use cases and results are shown. It is also described how the DSL is made general to accommodate embedded systems and how can it be extended for a particular domain such as automotive or calculator. Implications are defined as well to show what would happen if devices, program variables or xml file changes.

6 Conclusion and Future work

In this chapter detailed information about the background of the project, analysis of objectives and results of this project is provided. It gives the answers to what was the problem and reason behind building DSL and how it overcame the problem.

The main objective of this project was to provide user with a facility of test automation framework which could automate the testing procedure on HIL testing rig in automotive industry (JLR). This was done by building software ViBATA by Cranfield University. The test specification component of the software which has a major role in this software and is important for any testing framework was efficient but still manual. Instead of making another programming functionality to do this job a research was taken to identify what can be the best solution to this problem. The outcome of that research was that scripting or procedural languages can provide the means to create scripts for test automation. Domain-specific language is a scripting language with its declarative nature, limited expressiveness focused on a particular problem gave answers to all questions.

To build a Domain-specific language research was undertaken to understand what other people have performed work in this area and what the results were. The most related work is done by Wahler [9] but that DSL cannot apply to our problem because of its limitations described in section 2.7.2. Analysis of kinds and forms of DSL and tools available to build a DSL was also performed which resulted into building a textual DSL using Eclipse framework.

One of our aims of this DSL was to get an output which could specify test cases and read by our software ViBATA. To achieve this aim this DSL is producing output in XML which is a standard of providing interoperability between two applications. Web services are in place for interoperability between different software platforms also use XML to exchange data. This output will give the flexibility in this regard and test cases built by DSL can be read by any other

application provided that it has a plugin to process XML file such as XML plugin in ViBATA.

The second main objective was to enable DSL to generate code in any other GPL. To achieve this goal template expressions in Xtend are used which is producing code in Java for this DSL but can generate in other languages by replacing the code in Java, the implementation of which is shown in section 4.4.3 and results were shown in section 5.2.1.

Third aim was to know if DSL can work with other embedded system in addition to automotive. This aim is achieved by making structure of DSL in general way and proved by using the same DSL for calculator. We observed in section 5.4.1 that the DSL specify test cases for the calculator the result of DSL was generated in XML file which was read by Xml plugin and processed by the plugin for calculator built in ViBATA. We ran the test cases and got the results.

The fourth objective was that it should specify the settings of devices used in test environment. In case of ViBATA we were using XML file TAS.cofig to provide the settings for different devices which can be replaced by XML file produced by DSL for Device as shown in section 5.2.1.

Remaining objectives included to give the user an option to specify to create, delete and update test cases which also achieved by providing keywords in DSL as shown in section 5.2.1; to give user a facility of validation while typing program in DSL to improve his experience which is done by customizing validation folder in language infrastructure as shown in section 4.4.4; and to provide user code assistance while typing which is also achieved by customizing scoping folder in language project and proposal provider folder in UI project of language as shown in sections 4.4.5 and 4.4.6 respectively.

6.1 Future Work

A future work in terms of DSL can be integrating functionality, to interact with database directly, into the system as part of generated code. Our database built

in Microsoft Access for this project which could not be accessible through Java. The functionality in generated code should take commands from DSL and update the database which would be nice to have.

REFERENCES

1. Martin Fowler, Rebecca P., "*Domain-Specific Languages*", (2011), ISBN: 0-321-71294-3, publisher: Addison-Wesley Professional
2. Ayende Rahien, "*DSLs in Boo: Domain-Specific Languages in .NET*", (2010), ISBN: 978-1-933988-60-3, publisher: Manning Publications Co.
3. Glenford J. Myers, "*The Art of Software Testing*", (2004), pp. 10, Second Edition, ISBN: 0-471-46912-2, publisher: John Wiley & Sons, Inc.
4. John Watkins, Simon Mills, "*Testing IT*", (2010), pp. 92, Second Edition, ISBN: 978-0-521-14801-6, publisher: Cambridge University Press
5. Ian Sommerville, "*Software engineering*", (2007), pp. 561-563, 8th Edition, ISBN: 9781408251195, publisher: Addison-Wesley Professional
6. Eckard Bringmann, Andreas Krämer, "*Model-based Testing of Automotive systems*", In Proc. Software Testing, Verification, and Validation, pages 485-493. IEEE, (2008)
7. Sebastian Siegl, Kai-Steffen Hielscher, Reinhard German, Christian Berger, " *Formal Specification and Systematic Model-Driven Testing of Embedded Automotive Systems",* (2011)
8. Steffen Sch¨utte, "*A DOMAIN-SPECIFIC LANGUAGE FOR SIMULATION COMPOSITION",* (2011), pp.146-152
9. Wahler, M., Ferranti, E., Steiger, R., Jain, R. and Nagy, K., "*CAST: Automating Software Tests for Embedded Systems*", (2012)
10. Manuel Jiménez, Francisca Rosique, Pedro Sánchez, Bárbara Álvare, Andrés Iborra, "*Habitation: A Domain Specific Language for Home Automation",* (2009)
11. Hans Gronniger, Holger Krahn, Bernhard Rumpe, Martin Schindler and Steven Volkel, "*Text-based Modeling",* (October, 2007)
12. Howard Barringer and Klaus Havelund, *"Internal versus External DSLs for Trace Analysis Extended Abstract". In Proc. of the 2nd Int. Conference on Runtime Verification (RV'11), volume 7186 of LNCS, pages 1–3. Springer, 2011, (*2011)
13. Markus Voelter, *"A Family of Languages for Architecture Description", OOPSLA Workshop on Domain–Specific Modeling,* (2008)

14. Vicente Pelechano, Manoli Albert, Javier Munoz and Carlos Cetina, "*Building tools for model driven development. Comparing Microsoft DSL tools and Eclipse Modelling Plug-ins",* In *Proceedings of Desarrollo de Software Dirigido por Modelos--DSDM'06,* (2006)
15. B. Langlois, C.E. Jitia, E. Jouenne, "*DSL Classification",* In 7th *OOPSLA Workshop on Domain-Specific Modeling*, (2007)
16. Steve Cooke, Gareth Jones, Stuart Kent and Alan Cameron Wills, " *Domain-Specific Development with Visual Studio DSL Tools",* (2007) pp. 15-17
17. Michael Pfeiffer, Josef Pichler, "*A Comparison of Tool Support for Textual Domain-Specific Languages",* In Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling (October 2008), pp. 1-7
18. John Kent M.Sc., "*Test Automation: From Record/Playback to Frameworks",* at: http://www.simplytesting.com, paper given at EuroSTAR (2007)
19. G. Meszaros, "*Agile Regression Testing Using Record & Playback*," *Companion of the 18th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 03),* ACM Press, (2003), pp. 353–360.
20. Mark Blackburn, Robert Busser, Aaron Nauman, "*Why Model-Based test Automation is different and what you should know to get started",* In International Conference on Practical Software Quality and Testing, (2004)
21. Tairas, R., Mernik, M., Gray, J.: "*Using ontologies in the domain analysis of domain-specific languages",* In: Proceedings of the 1st International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering 2008. CEUR Workshop Proceedings., CEUR-WS.org, vol. 395 (2008)
22. Mark Strembeck and Uwe Zdun, "*An approach for the systematic development of domain-specific languages",* SOFTWARE—PRACTICE AND EXPERIENCE, (2009); vol. 39, pp. 1261–1273

23. Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, Steven Völkel, "*Design Guidelines for Domain Specific Languages",* In 9th OOPSLA Workshop on Domain-Specific Modelling, (2009)
24. Munnelly, J.; Clarke, S.; "*A Domain-Specific Language for Ubiquitous Healthcare, Pervasive Computing and Applications", (2008), ICPCA 2008. Third International Conference on* , vol.2, no., pp.757-762, 6-8 Oct. 2008
25. Harrison, W., Harrison, R., "*Domain specific languages for cellular interactions"*, In: Proceedings of the 26th Annual IEEE International Conference on Engineering in Medicine and Biology (2004)
26. Lisboa, E.B.; Silva, L.; Lima, T.; Chaves, I.; Barros, E.; "*An approach to concurrent development of device drivers and device controller*," Advanced Communication Technology, (2009), vol.01, pp.571-575
27. Dean Kramer, Tony Clark, and Samia Oussena. "*Mobdsl: A domain specific language for multiple mobile platform deployment"*. In Proceedings of the IEEE International Conference on Networked Embedded Systems for Enterprise Applications. IEEE, (2010)
28. Peter Friese, Sven Efftinge, Jan Köhnlein, "*Build your own textual DSL with tools from the Eclipse Modeling project*", at: http://www.eclipse.org/articles/Article-BuildYourOwnDSL/, (2008)
29. Marcel van Amstel, Mark van den Brand, and Luc Engelen. "*An exercise in iterative domain-specific language design*", In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution, (2010)* http://doi.acm.org/10.1145/1862372.1862386
30. Marjan Mernik, Jan Heering, and Anthony M. Sloane, "*When and how to develop domain-specific languages"*, pp. 316-344
31. Xtext Documentation at: http://www.eclipse.org/Xtext/documentation.html, (2012) (visited: 01.07.2012)
32. Stuart Barnes, Ambreen Hussain and Alexandros Mouzakitis, "*Automated Testing of a Vehicle Instrument Cluster*", International Conference on Systems Engineering, (2012).

33. YingPing Huang, Ross McMurran, Gunwant Dhadyalla, R. Peter Jones, and Alexandros Mouzakitis, *"Model-Based Testing of a Vehicle Instrument Cluster for Design Validation using Machine Vision",* (2009).
34. Xtend Documentation available at: http://www.eclipse.org/xtend/documentation.html, (2012) (visited: 12.07.2012)
35. Lorenzo Bettini, '*Xtext 2.1: Using Xbase Expressions*' (2011), available at: http://www.rcp-vision.com/?p=1640 (visited: 06.12.2012)
36. Sven Efftinge, Moritz Eysholdt, and Jan Kohnlein, "*Xbase: Implementing Domain-Specific Languages for Java*", (2012)
37. Sven Effting, "*Martin Fowler's State Machine DSL with Xtext 2.3*", (2012), available at: http://blog.efftinge.de/2012/05/implementing-fowlers-state-machine-dsl.html (visited: 20.09.2012)
38. Moritz Eesholdt and Heiko Behrens, *"Xtext – Implement you Language Faster than the Quick and Dirty way",* 2010
39. Calculator Demo, available at: http://msdn.microsoft.com/en-gb/library/vstudio/ms771362(v=vs.90).aspx (visited: 15.02.2013)
40. In-Sight Explore (2013) available at: http://www.cognex.com/in-sight-explorer.aspx (visited: 23.03.2013)
41. About the Eclipse Foundation (2013) available at: http://www.eclipse.org/org/ (visited: 24.03.2013)
42. Sven Efftinge and Markus Volter, *"oAW xText: A framework for textual DSLs",* 2006
43. Simple XML Serialization available at http://simple.sourceforge.net/home.php (visited: 24.09.2012)
44. Domain-Specific Language (2013) available at: http://en.wikipedia.org/wiki/Domain-specific_language (visited: 22.02.2013)
45. HTML Introduction (2013) available at http://www.w3schools.com/html/html_intro.asp (visited 22.05.2013)
46. CSS Introduction (2013) available at http://www.w3schools.com/css/css_intro.asp (visited 22.05.2013)

47. SQL Introduction (2013) available at http://www.w3schools.com/sql/sql_intro.asp (visited 22.05.2013)
48. What is ANTLR? (2012) available at http://www.antlr.org/ (visited 22.05.2013)
49. Introduction to Bison (2008) available at http://www.gnu.org/software/bison/ (visited 22.05.2013)
50. About ABB (2013) available at http://www.abb.com/ (visited 27.05.2013)
51. WinRunner – As a GUI based load testing tool (2004) available at http://www.loadtest.com.au/Technology/winrunner.htm (visited 27.05.2013)
52. QARun Documentation (2012) available at http://support.microfocus.com/documentation/ASQ/QARunDocs.aspx (visited 27.05.2013)
53. HP Unified Functional Testing (Quick Test Professional) (2011) available at http://www.automation-consultants.com/products-HP_Unified_Functional_Testing_(Quick_Test_Professional)-135 (visited 27.05.2013)
54. Rational Robot (2013) available at http://www-03.ibm.com/software/products/us/en/robot/ (visited 27.05.2013)
55. Welcome to Protege(2013) available at http://protege.stanford.edu/ (visited 27.05.2013)
56. M. d'Amorim and K. Havelund, "*Event-based runtime verification of Java programs"*, ACM SIGSOFT Software Engineering Notes, 30(4):1–7, (2005).
57. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. "*Rule-based runtime verification"*. In VMCAI, volume 2937 of LNCS, pages 44–57. Springer, (2004).
58. H. Barringer, D. E. Rydeheard, and K. Havelund. *"Rule systems for run-time monitoring: from Eagle to RuleR"*. J. Log. *Comput*., 20(3):675–706, (2010).
59. H. Barringer, A. Groce, K. Havelund, and M. Smith. "*Formal analysis of log files*". Journal of Aerospace Computing, Information, and Communication, 7(11):365–390, (2010).
60. H. Barringer and K. Havelund. "*TraceContract: A Scala DSL for trace analysis"*. In 17th International Symposium on Formal Methods (FM'11),

Limerick, Ireland, June 20-24, 2011. Proceedings, volume 6664 of LNCS, pages 57–72. Springer, (2011).

61. Scala (2013) available at http://www.scala-lang.org (visited 27.05.2013)
62. UML (2013) available at http://www.uml.org/ (visited 27.05.2013)
63. Introduction to CVS (2006) available at http://cvs.nongnu.org/ (visited 29.05.2013)
64. Apache Subversion (2011) available at http://subversion.apache.org/ (visited 29.05.2013)
65. H. Krahn, B. Rumpe, and S. Volkel. "*Integrated Definition of Abstract and Concrete Syntax for Textual Languages*". In Proceedings of Models 2007 (2007)
66. Mark van den Brand et al. "*The ASF+SDF Meta-Environment: a Component-Based Language Development Environment"*. In Proceedings of Compiler Construction 2001 (CC 2001), LNCS. Springer, 2001.
67. F. Jouault, J. Bezivin, and I. Kurtev. "*TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering"*. In Proceedings of the fifth international conference on Generative programming and Component Engineering 2006
68. MetaCase (2013) available at http://www.metacase.com (visited 29.05.2013)
69. J. Aldrich, C. Chambers, and D. Notkin. "*ArchJava: connecting software architecture to implementation"*. In ICSE, pages 187-197, 2002.
70. LINQ (2013) available at http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx (visited 29.05.2013)
71. Eclipse Modeling Project (2013) available at http://www.eclipse.org/modeling/ (visited 29.05.2013)
72. Microsoft Domain-Specific Language Tools (2013) available at http://www.microsoft.com/en-us/download/details.aspx?id=2379 (visited 29.05.2013)
73. Xactium (2013) available at http://www.xactium.com/ (visited 29.05.2013)
74. MPS (2013) available at http://www.jetbrains.com/mps/ (visited 29.05.2013)

75. Model-in-the-Loop (2013) available at http://www.emmeskay.com/verification-and-validation/model-in-the-loop-mil (visited 29.05.2013)
76. Hardware-in-the-Loop (2013) available at http://de.wikipedia.org/wiki/Hardware_in_the_Loop (visited 29.05.2013)
77. Software-in-the-Loop (2013) available at http://de.wikipedia.org/wiki/Hardware_in_the_Loop (visited 29.05.2013)
78. LonWorks (2013) available at http://www.echelon.com/technology/lonworks/ (visited 29.05.2013)
79. KNX (2013) available at http://www.knx.org/uk/ (visited 29.05.2013)
80. ArchC (2013) available at http://www.archc.org/ (visited 29.05.2013)
81. Bhasker, J. "*A SystemC Primer*", Star Galaxy Publishing, 2002.

APPENDICES

Appendix A Modifications done in Software

ViBATA was initially started by a developer who built the infrastructure of the software. Tests could be copied from the excel sheet into the software. Test cases for a category in an IPC could be created and running a test was implemented. My main responsibility was updating the software according to the client's requirement and maintaining it. In this Appendix list of some of amendments done in software are defined

- The searching in .sdf file was case sensitive and allowed user to search if he enters the exact word without spaces and underscores. Now two checkboxes are given to user one is to ignore the case and second is to use all words whether separated by underscores or spaces this change was done in python file which run >python dscontrol.py to register COM server
- Individual tests could run and show the result but ability to run a list of tests was required. So batch mode of testing is implemented. User can run as many tests as he wants by clicking check box in front of the tests and executing them in Batch Testing tab
- Because of limited camera's flash memory *.job file for all the tests couldn't store. Now when .job file is created it stores on disk on location C:/InsightJobs. When a test runs the software picks .job file for the particular test from this location and loads into camera's memory and after test result is shown it deletes this .job file from it.
- Implemented change flags which shows 'Do you wish to Save…' warnings to avoid loss of work.
- Previously only test could be copied from one category to another but now categories along with all test contained in it can be copied from one IPC to the other.
- Some tests require pre-requisite tests to be executed first. A pre-requisite test is related to the input line of the main test and the signal value of that

input line depends on the result of the pre-requisite test. If a pre-requisite test of kind decisive fails the whole test fails.

- Lazy loading of treeview for IPC and categories is resolved by implementing load on demand.
- The searching of test lines is implemented if new DVP arrives, the tests which are entered for the previous IPC from old DVP can now be searched on Test Searching tab by browsing new DVP file and entering name of the worksheet name to which test belongs to. The software search for the test lines from that worksheet and if it is found copied it onto the list below from where user can paste it onto DVP entries section of the new test.
- Input/output template in the software was to only ease the process if required by more than one test. The input/output line could be edited but now to make software consistent if input/output line gets populated from a template then it cannot be edited until that template is edited which will edit all input/output lines populated from that template. If user tries to update a line which is using a template a warning comes up. If a template is deleted then will be deleted from all the input/output lines having that template.
- On saving test, previously every time tests were getting deleted first and getting saved in this way input/output lines were getting assigned new ids in database but now if it is saved for the first time new ids are assigned but saving after that will update the previous input/output and add new line if there is any. This is done by comparing the new list of lines with the old list.
- Saving output template is now working. Expected value can be entered as >=90 also a template can be deleted as well.
- Some tests give output of images display in the cycle. To capture such an image in a cycle there was a requirement to refresh the camera after a certain interval and capture the image until image is found or the test runs for a specific duration of time. To accomplish this user can now enter duration and seconds fields on entering output for the test. If these

two fields are entered then image cycle will run after every duration for seconds long entered

- Selecting all tests in IPC/Category is implemented in batch mode
- Selecting only failed tests in IPC/Category is implemented in batch mode
- Exporting of Test Results to a .csv file is implemented in both batch and individual test execution

Detailed User and Developer's guide; class diagrams and sequence diagrams are developed for the software