Model-Driven Endpoint Development for Testing Environment Emulation

A thesis submitted in fulfilment of the thesis requirement
for the degree of Doctor of Philosophy

By

Jian Liu

School of Software and Electrical Engineering,
Swinburne University of Technology,
Melbourne, Australia

2017

Abstract

Modern enterprise software systems often need to interact with many heterogeneous systems in a distributed environment. As a result, integration testing has become a critical step in software development lifecycle. Conducting integration testing is a challenging task because production systems are generally neither suitable nor available for such testing and the cost to replicate such an environment for integration testing is usually too high. Testing environment emulation is an emerging technique for creating a testing environment with realistic executable models of server side production-like behavior. However, existing emulation approaches used to build these models either require a very significant amount of development effort or depend on the availability of special knowledge, tools and resources.

Aiming to achieve high testing environment development productivity and ease of use for domain experts, we have developed a novel domain-specific modeling approach for testing environment emulation. Our approach is based on model-driven engineering, where users work on high level abstraction models and executable code will be generated by transforming these models using code generators. Our approach divides enterprise system service components – or what we call endpoints - into three horizontal layers for processing messages and several vertical attributes for testing the conformance to specified non-functional requirements. Each of these layers or attributes represents a modeling problem domain. To model endpoints, we have developed a suite of domain-specific visual languages for users to model endpoints in horizontal layers and vertical attributes.

The key contributions of this research include: (1) a novel solution to model endpoint external behaviors for system integration testing, (2) a software interface description framework to abstract an endpoint into multiple logical layers and attributes, (3) a suite of domain-specific visual languages for users to model endpoints in layers and attributes, and (4) evaluation of these using case studies and a target end user survey. The scope of our approach is for emulating complex interactions between a system under test and an endpoint. Thus, other interactions behind the endpoint for providing composite services are not considered in this research. Applications of this approach include (but are not limited to) those using remote procedure calls with stateful session management. Our approach can be extended to cover most real-world situations.

Acknowledgements

I am in the fortunate position of working and living with many such wonderful and inspiring people. This thesis would not have been possible without their motivation, inspiration, guidance and support.

Firstly, I would like to express my deepest appreciation to my supervision team Prof. John Grundy, Dr. Mohamed Abdelrazek and Dr. Iman Avazpour for their enthusiastic and invaluable support throughout this research. The feedbacks, advices, and discussions provided by this team have been immeasurably valuable.

I thank Prof. Jun Han, A/Prof. Jean-Guy Schneider and Dr. Steve Versteeg for their early guidance helping me understanding of the key issues and concerns in this research area. My thanks also go to my review panel Prof. Chengfei Liu, Dr. Man Lau and Dr. Qiang He for their identification and solution of potential issues during my progress reviewing sessions.

I thank Swinburne University of Technology and the Faculty of Information and Communication Technologies for providing me the opportunity to undertake my PhD study and offering me an Australian Postgraduate Award scholarship.

Last but not the least, I am deeply grateful for my parents for raising me up, teaching me to be a good person. I am indebted to my family for their love, encouragement, patience and support throughout my doctoral program.

Declaration

This is to certify that,

- This thesis contains no material which has been accepted for the award of any other degree or diploma, except where due reference is made in the text of the examinable outcome; and

- To the best of my knowledge contains no material previously published or written by another person except where due reference is made in the text of the thesis; and

- Where the work is based on joint research or publications, discloses the relative contributions of the respective workers or authors.

Jian Liu

April, 2017

Melbourne, Australia

List of Publications

During my PhD project, a number of publications were produced.

- **Jian Liu**, John Grundy, Iman Avazpour, and Mohamed Abdelrazek, "A Domain-Specific Visual Modeling Language for Testing Environment Emulation," IEEE International Symposium on Visual Languages and Human-Centric Computing, Cambridge, UK, 4-8 Sept. 2016.

- **Jian Liu**, John Grundy, Iman Avazpour, and Mohamed Abdelrazek, "TeeVML: Tool Support for Semi-Automatic Integration Testing Environment Emulation," IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3-7 Sept. 2016.

- **Jian Liu**, John Grundy, Mohamed Abdelrazek, and Iman Avazpour, "Testing Environment Emulation: A Model-Driven Approach," International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Porto, Portugal, 19-21 Feb. 2017.

Table of Contents

1. Introduction **1**

1.1 System Integration Testing 1
1.2 Motivation 3
1.3 Testing Environment Emulation 6
1.4 Model-Driven Engineering 8
1.5 A Domain-Specific Approach to Testing Environment Emulation 10
1.6 Research Questions 12
1.7 Key Research Contributions 14
1.8 Thesis Structure 16

2. Literature Review **18**

2.1 Existing Approaches to Provisioning of Testing Environments 18
2.1.1 System Replication 19
2.1.2 Test Doubles -- Method Stubs and Mock Objects 20
2.1.3 Model-Based Testing Solutions 21
2.1.4 Approaches for Testing Environment Emulation 24
2.1.4.1 Interactive Tracing Approaches 24
2.1.4.2 Specification-Based Approaches 25
2.1.4.3 Other Test Bed Development Approaches 27
2.1.5 Comparison Summary 28
2.2 Software Interface Abstraction and Modeling 31
2.2.1 Software Interface Abstraction 31
2.2.2 Software Interface Syntax Specification 32
2.2.3 Protocol Modeling 34
2.2.4 Behavior Modeling 37
2.2.5 Security Modeling 41
2.3 Development of Domain-Specific Modeling Languages 44
2.3.1 Domain Analysis 45
2.3.2 DSL Design 48
2.3.3 Implementation 50
2.4 Domain-Specific Languages and Evaluation Criteria 52
2.5 Summary 54

3. Decision Making and Domain Analysis for Functional Layer DSLs **55**

3.1 Introduction 55
3.2 Domain-Specific Modeling Process 56
3.3 Decision Making 57

3.4 Domain Analysis 59
3.4.1 Applications Study 59
3.4.2 Software Interface Description Framework 63
3.4.3 Service Request Defects 66
3.4.4 Functional Layer Metamodels 68
3.4.4.1 Signature Modeling 68
3.4.4.2 Protocol Modeling 71
3.4.4.3 Behavior Modeling 73
3.5 Summary 74

4. Design and Implementation for Functional Layer DSLs 76

4.1. Introduction 76
4.1.1 DSL Development Principles 76
4.1.2 Domain Specific Rules 77
4.2 Domain-Specific Visual Languages 78
4.2.1 Visual Symbol Design 79
4.3 Design of TeeVML Domain-Specific Visual Languages 83
4.3.1 Signature Domain-Specific Visual Language 83
4.3.1.1 WSDL sub-DSVL 83
4.3.1.2 Operation sub-DSVL 88
4.3.1.3 Message sub-DSVL 89
4.3.2 Protocol Domain-Specific Visual Language 90
4.3.3 Behavior Domain-Specific Visual Language 96
4.3.3.1 Service Node 97
4.3.3.2 Node 98
4.3.3.3 Arc 99
4.3.3.4 Entrance and Exit Bars 99
4.3.3.5 Data Store 100
4.3.3.6 JDBC Operator 101
4.3.3.7 Evaluator 103
4.3.3.8 Conditional Operator 104
4.3.3.9 Loop 105
4.3.3.10 Variable and Variable Array 107
4.3.3.11 A Behavior Model Example 108
4.4 Implementation of Code Generators and a Domain Framework 110
4.4.1 Code Generators 110
4.4.1.1 Signature DSVL Code Generator 111
4.4.1.2 Protocol DSVL Code Generator 113
4.4.1.3 Behavior DSVL Code Generator 114
4.4.2 A Domain Framework and Target Environment 116
4.5 Metamodeling Language 122

4.5.1 Metamodeling Language Selection 122
4.5.2 Metamodeling Language and Toolset Chosen -- MetaEdit+ 123
4.6 Summary 127

5. Case Study - Functional Layer Modeling 128
5.1 Case Study 128
5.1.1 Example Signature Layer 128
5.1.2 Example Behavior Layer 130
5.2 Endpoint Modeling 133
5.2.1 Signature Modeling 134
5.2.2 Protocol Modeling 136
5.2.3 Behavior Modeling 139
5.3 Testing Environment Generation 144
5.4 Summary 147

6. QoS Modeling – Security Attribute Example 150
6.1 Introduction 150
6.1.1 Security Requirements 150
6.1.2 In-Premises Data Security 152
6.1.3 Data Security in Transit 153
6.2 Security Domain Analysis 155
6.3 Security DSVL Design 161
6.4 Case Study - Endpoint Security Modeling 164
6.4.1 Instantiation of Role, Operation and Resource 164
6.4.2 Definition of Sub-Roles and Users 165
6.4.3 Security Constraint Definition 166
6.4.4 ERP Endpoint Security Attribute Modeling 166
6.5 Implementation 167
6.5.1 Role-Based Access Control Implementation 168
6.5.2 Username and Password Security 170
6.6 Summary 176

7. Evaluation 178
7.1 Technical Comparison 179
7.1.1 Testing Functionality 179
7.1.2 Development Productivity 180
7.1.3 Ease of Use 180
7.2 Qualitative Comparison 181
7.3 User Survey 183
7.3.1 Overview 183
7.3.2 Questionnaire Design 184

7.3.3 Phase One 185
7.3.3.1 Participant Recruitment 185
7.3.3.2 Experiment Setup 186
7.3.3.3 Survey Results Analysis 186
7.3.4 Phase Two 189
7.3.4.1 Participant Recruitment 189
7.3.4.2 Experiment Setup 190
7.3.4.3 Survey Results Analysis 190
7.3.5 Open-Ended Questions 195
7.4 Summary 196

8. Conclusions and Future Work 201
8.1 Conclusions 201
8.2 Future Work 203

References 206

Appendix I Approval Letter from Swinburne University Human Research Ethics Committee **217**

Appendix II Phase One Questionnaire **219**

Appendix III Phase Two Questionnaire **224**

Appendix IV Phase One Survey Results Report **229**

Appendix V Phase Two Survey Results Report **279**

List of Figures

Number	Description	Page
1.1.	The example ERP and CRM interactions process flow diagram	4
1.2.	The conceptual model of testing environment emulation	8
1.3.	The conceptual model of model-driven engineering	9
1.4.	Endpoint modeling and runtime environment	12
2.1.	Model-Based Testing Process	22
2.2.	Interactive tracing approaches for TEE	25
2.3.	The Architecture view of Kaluta testing environment emulator	26
2.4.	An example FSM to represent online shopping account process	35
2.5.	An example BPMN diagram for a recruitment process	39
2.6.	An example DFP diagram for a CRM system	40
2.7.	NIST Core RBAC model	43
2.8.	An example metamodel of a warehouse definition	46
2.9.	A typical domain-specific modeling environment	51
2.10.	Implementation pattern selection guideline	52
3.1.	A DSM approach development process and application modeling	56
3.2.	The core banking system endpoint state transition diagram	62
3.3.	The LDAP server state transition diagram	64
3.4.	Endpoint signature metamodel	70
3.5.	Endpoint protocol metamodel	72
4.1.	Signature WSDL sub-DSVL dialog box	84
4.2.	An example endpoint signature WSDL model	88
4.3.	An example Operation instance	89
4.4.	An example Message instance	90
4.5.	ConstraintTransition relationship dialog box	91
4.6.	An example endpoint protocol model	92
4.7.	An example Service Node instance	98
4.8.	An example Node instance	99
4.9.	An example of Entrance and Exit bars	100
4.10.	An example Data Store instance	101

Number	Description	Page
4.11.	An example JDBC Operator instance	102
4.12.	An example Evaluator instance	104
4.13.	An example Conditional Operator instance	105
4.14.	Examples of For-loop and While-loop instances	107
4.15.	Examples of Variable and Variable Array instances	108
4.16.	The top view of an example behavior model	109
4.17.	A Service Node instance definition	109
4.18.	A code snippet of Signature DSVL code generator	112
4.19.	Type definition part of an example WSDL file	113
4.20.	A code snippet of Protocol DSVL code generator	114
4.21.	A code generator function to check the availability of input parameters	115
4.22.	An example endpoint skeleton implementation class	118
4.23.	An example operation method of PurchaseServer class	119
4.24.	The code of an operation SUT API class	119
4.25.	A code snippet of Ant build file for deploying Tomcat Web service	121
4.26.	The deployment view of an endpoint and its SUT	121
4.27.	MetaEdit+ 5.1 graph tool	125
4.28.	MetaEdit+ 5.1 symbol editor	126
4.29.	MetaEdit+ 5.1 code generator editor	126
5.1.	The entity relationship diagram of the ERP endpoint persistent data tables	130
5.2.	The activity diagram of supplier purchase process	133
5.3.	MetaEdit+ 5.1 diagram editor	134
5.4.	The example endpoint signature model dialog box	135
5.5.	The example endpoint signature WSDL model	135
5.6.	The example endpoint “paymentrequest” Operation instance	136
5.7.	The example endpoint “paymentrequest” Message instances	137
5.8.	The example endpoint protocol model	138
5.9.	The example endpoint protocol model constraint condition definition	138
5.10.	The example endpoint protocol model business scenarios simulation	139

Number	Description	Page
5.11.	The example endpoint top level behavior model	140
5.12.	The example “paymentrequest” behavior model	142
5.13.	The entrance bar definition of “poinformationretrieve” node	143
5.14.	The dialog box for database operation definition	143
5.15.	The example signature model transformation	145
5.16.	The example testing service through Tomcat	146
5.17.	The code of a dummy SUT class	147
5.18.	The request and response messages captured by TCPMon	148
6.1.	Static Separation of Duty relations (SSD) model – A RBAC component	154
6.2.	An example usage of public key cryptography	155
6.3.	The ERP endpoint application security requirement	156
6.4.	Endpoint security control process	158
6.5.	Endpoint security modeling metamodel	159
6.6.	Role visual construct	162
6.7.	Sub-role visual construct	162
6.8.	User visual construct	162
6.9.	Operation visual construct	163
6.10.	Resource visual construct	163
6.11.	Association relationship visual construct	163
6.12.	Security constraint visual construct	164
6.13.	Three sub-roles in a manager role and sub-role dialog box	165
6.14.	Two users in level1 sub manager role	166
6.15.	The example ERP endpoint security model	167
6.16.	Data modeling for endpoint security implementation	169
6.17.	WS-SecurityPolicy for plain text UsernameToken	171
6.18.	A code snippet of WS-SecurityPolicy for HTTPS connection	171
6.19.	A code snippet of WS-SecurityPolicy using a digest password	172
6.20.	The code of callback class PWCBHandler	173
6.21.	Two code snippets of a SUT client API class	174
6.22.	Request and response messages with a hash function UsernameToken	175
6.23.	A security modeling build property file for Ant auto-build tool	176

Number	Description	Page
7.1.	Phase One participants IT and software testing experience	186
7.2.	Phase Two participants' IT background	190
7.3.	The survey results of SUS questions	192
7.4.	In favour responses for different interface layers and usability dimensions	194

List of Tables

Number	Description	Page
1.1.	System integration testing	2
1.2.	Model-driven engineering description	9
2.1.	Approaches comparison for testing environment development	29
2.2.	The differences between GPLs and DSLs	53
2.3.	DSL engineering evaluation criteria	53
3.1.	Service request defect types	67
4.1.	PoN principles and our visual symbol design rules	81
4.2.	WSDL sub-DSVL visual constructs	85
4.3.	WSDL sub-DSVL domain rules	87
4.4.	Complex Element visual construct properties	89
4.5.	Protocol DSVL visual constructs	93
4.6.	Protocol DSVL domain rules	95
4.7.	JDBC Operator properties	102
4.8.	Evaluation summary results of metamodeling tools	124
5.1.	The example signature definition	129
5.2.	The ERP endpoint persistent data tables and fields	131
6.1.	Description of endpoint security metamodel	160
7.1.	Testing environment emulation approaches comparison	182
7.2.	Questions and responses from Phase One survey report	187
7.3.	System Usability Scale questions	191
7.4.	Questions and responses for functional layers and usability dimensions	193
7.5.	Endpoint modeling productivity questions and responses	195
7.6.	Open-ended questions, feedbacks and solutions	197

CHAPTER 1

Introduction

1.1 System Integration Testing

Emerging computing strategies, such as Service-Oriented Architecture (SOA), cloud computing, Business Process Management (BPM) and social computing, represent an ongoing shift from locked-down, siloed and monolithic applications to highly distributed, heterogeneous and shared computing environments [1]. Most software systems need to interact with other systems to provide composite services to their clients or end users. In a typical deployment scenario, an enterprise system might interact with various heterogeneous systems, such as a legacy mainframe system, directory servers, database servers, third-party middleware systems and many others. Thus, the performance of a software system is no longer determined only by its own internal components but is also subject to its increasingly complex interactions with external systems in its operational environment. This means that for effective testing of a software system, testing interconnections (static communication aspects) and interoperability (dynamic communication aspects) of the systems that it communicates with in a realistic production environment is critical.

The consequence of a System Under Test (SUT) failing to interact correctly with other systems in its operational environment may not only cause the failure of provision of its own service, but in the worst-case scenario can also bring a catastrophic failure to the entire enterprise environment. An example of such failure is the telephone network crash of USA telecom giant AT&T in the 1990's. The root cause of the disaster was due to the failure of one particular switching system. After a failure this system would send a message to its directly connected switching units to tell them that there was a problem. Unfortunately, the arrival of that message would cause those switching units to fail as well – resulting in a cascading failure rapidly spreading across the entire AT&T long distance telephone network [2].

System Integration Testing (SIT) is a testing process that exercises a software system's coexistence (integration) behaviors with other inter-connected systems. It tests the interactions between different systems and verifies the proper execution of the SUT in its deployment environment [3]. Table 1.1 summarizes what SIT objectives are and how SIT needs to be conducted.

Table 1.1. System integration testing

	Description
Objective	To verify the correctness of a SUT interacting with its environment in accordance with interface specifications (static aspects), and validate the interactive performance of the SUT with other systems at runtime (dynamic aspects).
How	Using black-box testing method that examines the external performance of a SUT without peering into its internal structures and implementations.
When	After unit and system tests that have been carried out by software developers to validate the correctness of internal implementations using white-box testing method.
Who	Software testing engineers, system analysts or even business users (we call them domain experts, hereafter), who have rich business domain knowledge, but might lack technical skills.

Conducting SIT is a challenging task, particularly in a large-scale, distributed and heterogeneous enterprise environment with large complex enterprise systems. Pawar et al. conducted a SIT survey and summarised three key issues to be addressed for a typical enterprise environment [4]:

- **Heterogeneity of Software Systems** -- Since the systems are implemented using various programming languages and run on different platforms, heterogeneity in a computing environment will introduce non-uniformity in message exchanges and certain forms of transmission protocol conversions would be needed;
- **System Communication** -- When a client system sends a request to its server, the server will reply back if and only if the client request is per specification. For most complex software environments, middleware communication, naming services and data models need to be taken into account;
- **Distributed System Issues** – Some specific issues may arise from a distributed computing environment, such as transaction control, deadlocks, and the

coexistence of two or more different versions of a software system. These distribution-related issues can only be detected during the SIT phase of development.

In addition, the provision of a suitable testing environment is another key issue to be addressed by IT professionals. To test a SUT's interactions with the systems in an enterprise environment, the testing environment must provide SIT functionality, which should encompass all the services of each system the SUT will invoke in the environment. A production environment is generally unsuitable to conduct this kind of testing, as a fault in the SUT may cause disruption of business operation or even irreversible damage to that production environment. With the increasing complexity of the environment an application is deployed in, it is getting more difficult or even near impossible to replicate such a production environment.

1.2 Motivation

To motivate the research of this thesis, we select a typical business case of a global company integrating its legacy system with a public cloud application and use this case to describe the potential interactions between an endpoint and its SUT. Let us assume that the company has an in-house Enterprise Resource Planning (ERP) system Oracle Corporation's PeopleSoft Finance [5] to support its daily operations. For the purpose of streamlining its sales process and improving operational efficiency, the company plans to introduce a public cloud Customer Relationship Management (CRM) service provider salesforce.com [6] as its sales frontend application. From operation and data security considerations, all company data will be still kept in-house in the ERP. Therefore, the CRM application must interact with the ERP system intensively for accessing persistent data and processing business logics.

The activity sequence diagram in Figure 1.1 illustrates a typical sales process flow among users, the CRM application and ERP system. Users access the CRM application for handling their client Purchase Order (PO). For every user request, the CRM must invoke a corresponding ERP operation[1] using Remote Procedure Call (RPC) communication

[1] To be consistent with the naming convention used by Web Services Description Language (WSDL) specification, we call all services provided by an application as "service" and individual service as "operation" hereafter.

style. Our main interest is on the interactions between the client CRM and server ERP, and they are described below.

Whenever the ERP receives a "logon" request (refer to #1 in Figure 1.1) from the CRM, it transits from idle state to home state and an interactive session starts. The next valid operation is a "porequest" request (#2), followed by an "inventorycheck" (#3). The returned value of the "inventorycheck" will determine whether or not supplier chain related steps will be executed. If the item inventory has enough stock for the PO, the process flow will jump over those supplier purchasing steps and directly go to "paymentrequest" (#8). Otherwise, we have to go through the supplier purchase steps (#4, #5, #6 and #7) to buy the missing quantity of the item. Both "supplierpoapproval" (#5) and "approvalnotification" (#6) are iteration operations, informing all approvers one-by-one to give their approvals. If all required approvals for the supplier PO are obtained, the rest of purchasing steps will be executed in the order as in Figure 1.1. Otherwise, the sales process will be aborted without success.

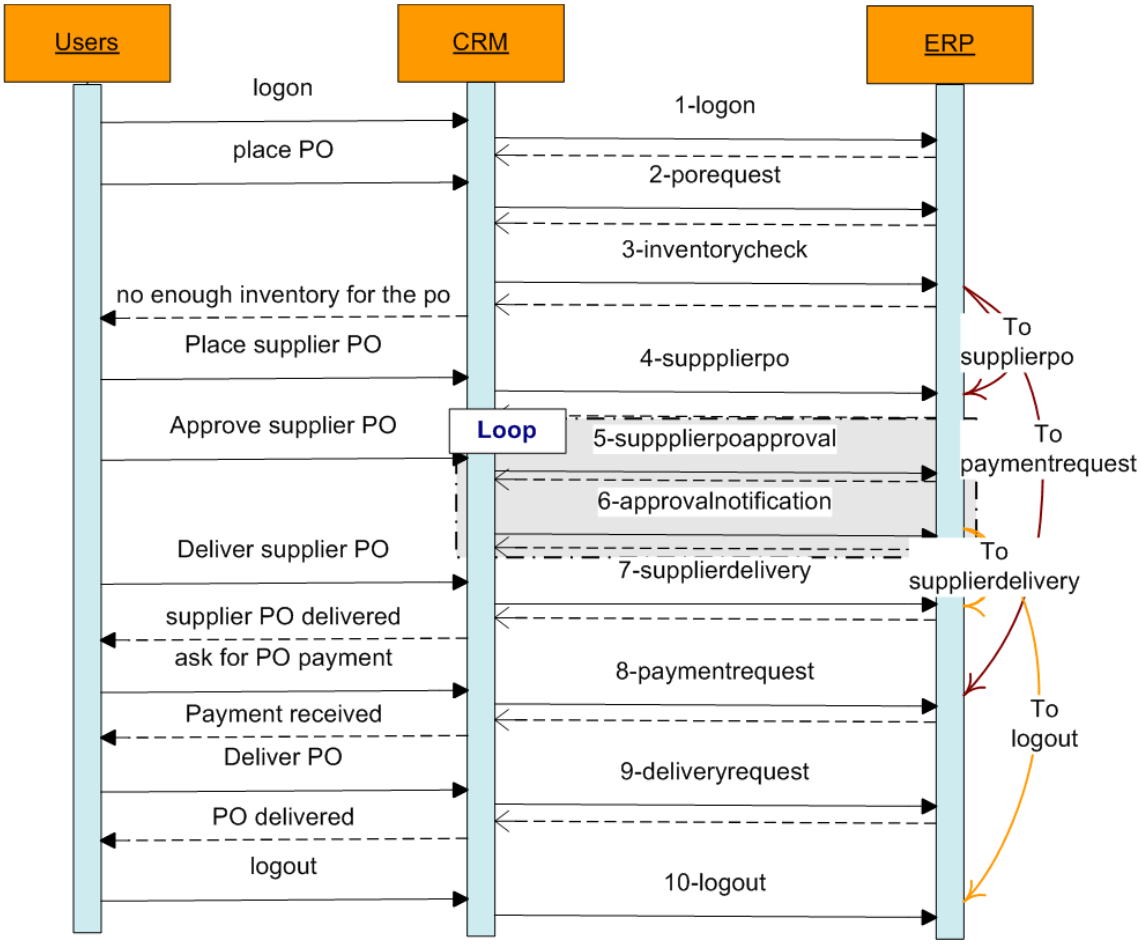


Figure 1.1. The example ERP and CRM interactions process flow diagram

To ensure the interconnectivity and mutual interoperability between the ERP system and the CRM application, SIT must be carried out before putting the CRM in production. For this study, we treat the in-house ERP as the endpoint that we need to model, and the cloud CRM as the SUT to be tested. An endpoint is a server-side application, receiving and processing operation requests from a SUT and generating responses to the SUT. Thus, the endpoint should be able to validate the correctness of the operation requests sent from the SUT.

The SUT request defects can be grouped to static and dynamic categories, depending on whether they will always cause interactive failures or under a certain runtime conditions only. Normally, a software application includes an interface specification to specify its provided operations and their parameters. A SUT as a client of the application must send its requests to the application in accordance with the interface specification. Otherwise, an interaction fault will occur due to a static interface defect. On the other hand, a dynamic defect happens under certain business scenarios. An example is the validity of the next request after "inventorycheck" (refer to #3 in Figure 1.1), which is subject to the inventory result returned by the request. In general, static defects can be found by code review against interface specification and SIT; while dynamic defects can only be captured by SIT.

From another angle, SUT request defects can also be grouped into functional or non-functional (also called QoS) categories. The functional defects are those directly related to operation request processing by endpoint. These defects can be the parameters mismatching in a service request and its corresponding endpoint operation, an invalid request for an endpoint state, and many others. The non-functional defects are those related to operational non-compliance with an endpoint. These defects could include those invoking an endpoint operation or accessing a resource without proper rights (security aspect), incapable of handling various endpoint faulty conditions (robustness aspect), etc.

Not only does the CRM application communicate with the ERP system in the enterprise environment, but also many others as well, such as Email, LDAP, DNS, middleware and other systems. Thus, the CRM interacting with each of these systems must be properly tested and verified, and a testing environment to test the CRM should include all these systems. Development of these systems will be a tedious and troublesome work and require IT professionals to undertake a tremendous effort, if a traditional software development process is adopted. Sometimes, such a development effort even overtakes the

effort spending on actual testing. From a technical and economic consideration, it is highly desirable to have a more productive approach to develop endpoints, instead of using some popular third-generation languages, e.g. Java.

As stated in Table 1.1, SIT is normally carried out by business domain experts. Due to their background, ease of use is one of their main concerns when they choose their preferred development tool. From the domain experts' perspective, ease of use may refer to three aspects: (1) a short learning curve to master a new tool, (2) design intent can be expressed declaratively rather than imperatively, and (3) directly mapping problem domain concepts to programming constructs.

From this example case study, we can conclude that SIT for a SUT is an essential part of a software development life-cycle and such a testing must be conducted in a production-like testing environment. The testing environment must be able to capture all the SUT interface defects, including functional and non-functional, static and dynamic defects. Furthermore, the approach to develop such a testing environment must have high development productivity and be easy to use by domain experts as well.

Over the years, a number of approaches have been proposed to provide testing environments, but most of them have their own limitations and shortcomings. Using hardware virtualization techniques, such as VMWare [7] and VirtualBox [8], requires elaborate installation, configuration and running of a replica of the real environment. Programmatic approaches, such as method stubs [9] and mock objects [10], abstract away from real communication complexities. This simplification may have some impact on their result accuracy. Interaction trace record-and-reply approaches [11] are infeasible, as the CRM is a complete new application for the company and there are not any trace records available. Existing specification-based approaches [12, 13] are difficult to use, as they require users to have both business domain knowledge and programming skills to develop endpoints manually.

1.3 Testing Environment Emulation

Testing Environment Emulation (TEE) is an emerging technique to provide SIT environment for a SUT that interacts with many external systems. The main idea is to model the static and runtime behavior of each system in the environment and replace the systems by the instances of the corresponding models in the emulation environment [14].

The aim is to make the emulated testing environment rich enough to "fool" the SUT that it is talking to real external systems. Other components and systems which sit underneath and in the background are ignored from the emulated environment perspective whatever possible. Particularly, an emulated endpoint is a simplified version of a real system with three assumptions:

- As an endpoint is used to provide a test-bed for a SUT integration testing, only the external behaviors of the endpoint application are considered and its internal implementations will be ignored;
- An endpoint is specifically developed for a SUT. Therefore, a subset of the endpoint application operations invoked by the SUT are provided;
- Serving as a defect detection tool for system debugging, an endpoint should be able to capture all SUT interface defects, together with their types, origins and other information.

Figure 1.2 illustrates the basic structure of an emulated SIT environment. A SUT is at the left, without any modifications before putting it to test. It is unaware that it does not interact with a "real" enterprise environment, but an emulation thereof. The testing environment is at the right, consisting of various endpoint types. Each endpoint type has one or more instance(s). An endpoint is represented as its real application's facade, meaning that from the perspective of the SUT, an endpoint will appear as if it is in a real deployment environment. However, there is no application software running behind the facade, rather a corresponding endpoint model is used to dictate the behavior of an emulated endpoint. The endpoint models differ according to the types of systems being emulated. The SUT sends requests to the corresponding endpoint facade; and the endpoint facade not only receives requests intended for the associated real endpoint system, but also returns responses seeming to be sent from the real application after verifying these requests.

The key benefits from using an emulated testing environment include:

- It provides a production-like test-bed in terms of the provisioning of testing functionality to SUTs in a much more cost-effective way than application replication;

- Development of such a testing environment could be quick and easy, as some internal logic implementations and auxiliary modules are ignored;
- The test-bed is easily configured and monitored for performing QoS aspects testing, such as simulating different numbers of instances of the same endpoint type for performance test;
- Software interface defects can be captured and the defect cause information can be reported.

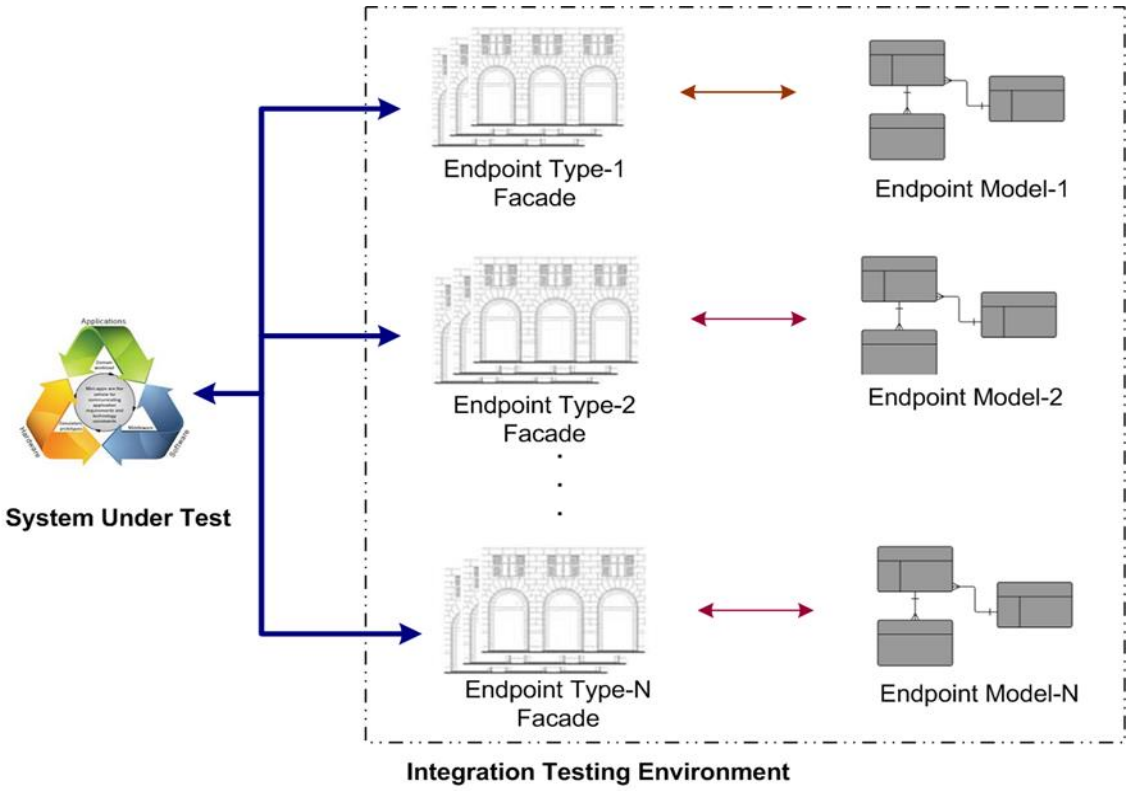


Figure 1.2. The conceptual model of testing environment emulation

1.4 Model-Driven Engineering

Model-Driven Engineering (MDE) is a software development methodology that separates the system functionality being developed from the implementation details of such a system using a high-level specification language [15]. The key benefits from using a MDE approach include the increased productivity by raising the level of abstraction beyond programming, simplified software development process by using models to represent design patterns in application domains, and enhanced communications among

software stakeholders by standardising terminologies and directly using problem domain concepts.

MDE is based on Meta-Object Facility (MOF) hierarchical architecture defined by Object Management Group (OMG) [16]. MOF provides a type system for entities in CORBA architecture [17] and a set of interfaces through which those types can be created and manipulated. The elements at an abstraction level define the elements' concepts, attributes and their relationships at one lower level, and the elements at the lower level are the instances of the elements at one higher level. Figure 1.3 shows the conceptual model of MOF architecture. The grey boxes represent MOF four abstraction levels, and the blue boxes are the key concepts used to develop Domain-Specific Languages (DSLs) and model software applications. Table 1.2 describes each MOF abstraction level and the concepts shown in Figure 1.3.

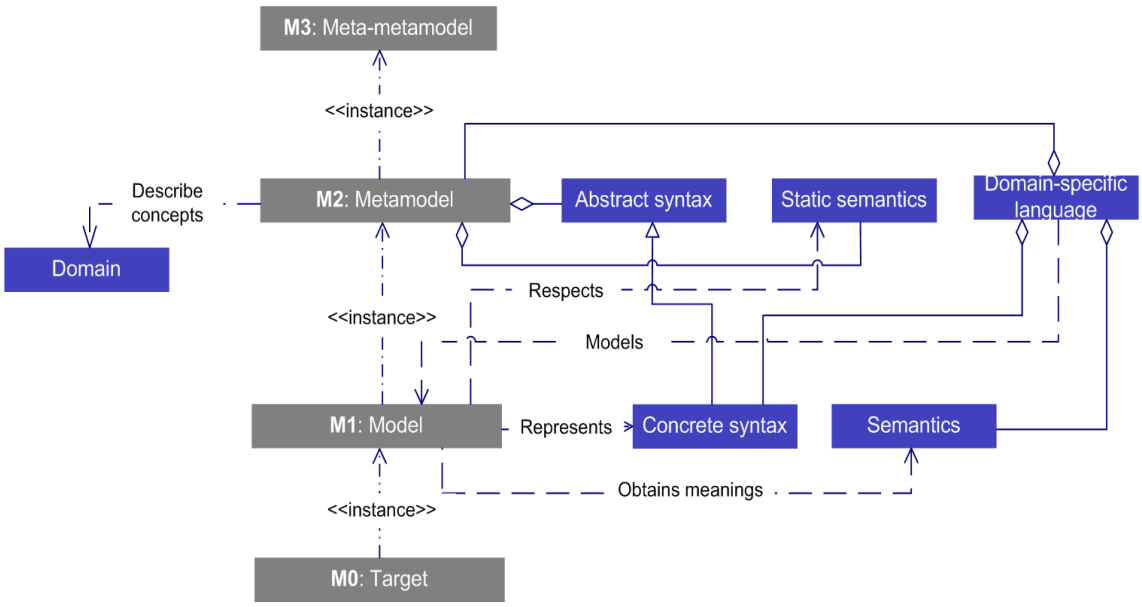


Figure 1.3. The conceptual model of model-driven engineering

Table 1.2. Model-driven engineering description

Concept	Description
M3: Meta-metamodel	This is the root level, where OMG has defined the MOF language as the root of all model-driven developments. It uniquely describes the concepts used to represent any metamodels in any domains. There are some available tools to implement the meta-metamodel, such as MetaEdit+ [18] and Eclipse Modeling Framework (EMF) [19].
M2: Metamodel	In this level, all concepts and their relationships are defined within a specific application domain. Moreover, the key semantics and constraints associated

	with these domain concepts are also specified. A domain metamodel is normally developed from a domain analysis by domain experts.
M1: Model	The model layer is a user specification layer of a software application using domain-specific languages, and it contains concrete definitions of the element types defined at the metamodel level.
M0: Target	This run-time layer is transformed from models by code generators in executable forms. It contains the objects instantiated out of the models.
Domain	To work with MDE it is necessary to always fix a specific domain, which delimits a field of knowledge. That is the reason why it could be desirable to create an ontology of the domain concepts.
Abstract and concrete syntaxes	The abstract syntax of metamodel focuses on the conceptual elements, whereas the concrete syntax of DSL focuses on how to represent the concepts.
Static semantics	The static semantics of metamodel are based on the abstract syntax, and they are used to make semantics checks on models to ensure they are well constructed.
Domain-specific language	It is a defined language used specifically to address problems in a specific domain, being the key to any domain specific solutions. A typical DSL contains programming constructs for users to model applications and code generators to transform the models to target.
Semantics	It is important to associate the elements of a language with the corresponding domain concepts, so that we can map the concepts of a language directly to concepts of the domain that is being modeled, without the possibility of misinterpretation.

1.5 A Domain-Specific Approach to Testing Environment Emulation

While TEE provides an effective means to emulate a server side production-like testing environment to test SUTs, it does not include a specific development toolset to develop endpoints. MDE addresses the traditional software development issues and promises higher development productivity and easier to use for non-technical background domain experts. Based on these two techniques, we propose a novel model-driven Domain-Specific Modeling (DSM) approach to develop testing environment.

DSM achieves high development productivity and product quality by focusing on a narrowed problem domain, so that specific high-level abstraction programming constructs and automatic tools can be created. Our TEE approach is based on a new software interface description framework, where software interfaces are abstracted into three horizontal layers and several vertical attributes. We use modular development

approach to model an endpoint – i.e. each module represents a particular interface layer or attribute.

Horizontal layers consist of endpoint interface signature, interactive protocol and internal behavior. They are directly related to service request processing in a top-down manner. The top level signature layer defines all operation requests, their parameters and properties. The next level protocol layer describes the validity of a temporal sequence of operation requests, which can depend on either endpoint states (static protocol behavior) or runtime constraint conditions (dynamic protocol behavior), or both. The bottom level behavior layer abstractly describes some of the endpoint internal operation request processing and response generation, and the returned values in response messages are used to capture dynamic protocol defects.

Vertical attributes are related to the QoS aspects of application interactions, and they specify the criteria that can be used to assess the operations of an endpoint system. Examples of vertical attributes include: compliance with endpoint security policies; robustness for handling various endpoint faulty conditions e.g. timeout, no response, wrong message in sequence, wrong message format; performance requirements to support the maximum required number of endpoint instances, throughput, etc. Vertical attributes are often validated first before processing horizontal layers. Unlike horizontal layers, different endpoints will have different vertical attributes depending on business scenarios, and operation requests are validated by an endpoint for the correctness of these QoS attributes in an arbitrary order[2].

Our DSM approach consists of an endpoint modeling environment and a runtime environment to provide testing services to SUTs. The modeling environment has a suite of domain-specific Visual Modeling Languages for Testing environment emulation (TeeVML), each for modeling a specific interface layer or attribute. The runtime environment is hosted in Axis2 SOAP engine [20] generated automatically by transforming the endpoint models. Testing service is provided through Web Service provided by Tomcat Servlet Container [21]. Figure 1.4 presents our DSM approach overview. The upper part of the diagram is the modeling environment provided by our TeeVML for users to model endpoints. An endpoint normally has both functional and

[2] This is the reason why we call QoS aspects as attributes, rather than layers for functional aspects.

non-functional models for the horizontal layers and vertical attributes, respectively. Code generators are in the middle to transform the endpoint models to the executable code of the runtime environment at the lower part. A SUT on the left sends requests to the endpoint runtime environment. These requests are validated by non-functional attributes in arbitrary order first, then by the signature and protocol layers. Once these requests pass these validations, they will be forwarded to the behavior layer for generating responses. The SUT will send different requests, depending on the results of the corresponding responses.

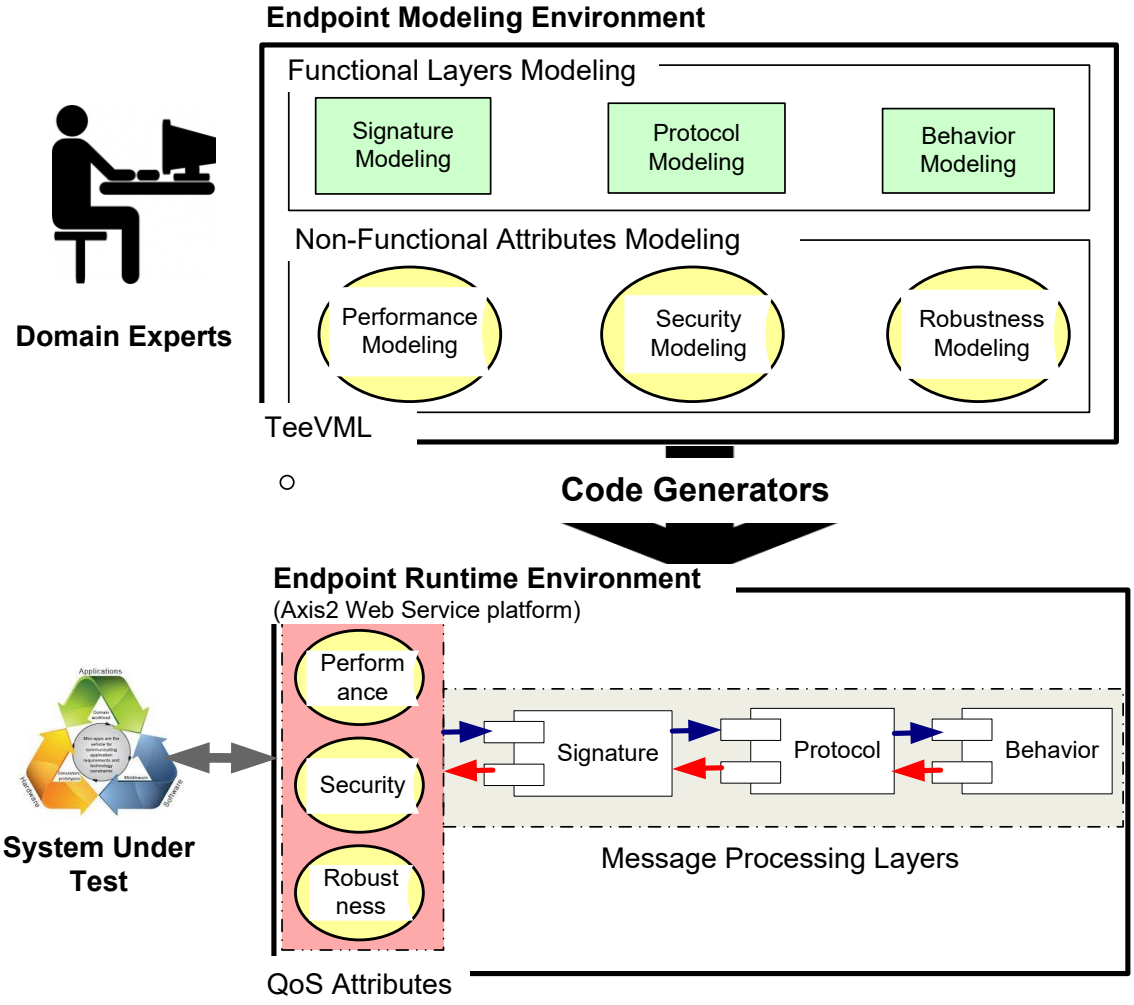


Figure 1.4. Endpoint modeling and runtime environment

1.6 Research Questions

Based the motivational case study described previously, we define three key research questions used in guiding our approach development. Furthermore, we elaborate these

research questions by adding some sub-questions to provide further details for specific aspects.

RQ1 – *Can we emulate a functioning integration testing environment capable of capturing all interface defects of an existing or a non-existing system under test from an abstract service model?*

- RQ1.1 Do the endpoints, developed by our approach, support both existing and new enterprise application SIT?
- RQ1.2 Do the endpoints, developed by our approach, report all types of signature defects?
- RQ1.3 Do the endpoints, developed by our approach, report all types of protocol defects, including static and dynamic defects?
- RQ1.4 Do the endpoints, developed by our approach, report QoS defects, such as security defects?
- RQ1.5 Can the endpoints, developed by our approach, simulate protocol scenarios, including time event, synchronous and unsafe operations?

RQ2 – *Would our model-based approach improve testing environment development productivity, compared to using third-generation languages (e.g. Java) to implement endpoints?*

- RQ2.1 Does our approach support a higher-level abstraction beyond programming?
- RQ2.2 Does our approach support component reuse within a DSL and across DSLs?
- RQ2.3 Can our approach provide error prevention mechanisms embedded in DSLs?
- RQ2.4 Does our approach automate endpoint generation process from models?

RQ3 – *Can we develop a user centric approach, easy to learn and use to specify testing endpoints by domain experts?*

- RQ3.1 Can we develop an approach that only uses problem domain concepts?
- RQ3.2 Can we develop an effective and usable approach that does not need any programming work?
- RQ3.3 Can we develop effective and usable endpoint modeling DSLs using visual notations?

RQ3.4 Do our DSL visual notations support acceptable cognitive effectiveness?

The development of our approach has been driven by these three research questions as the essential requirement inputs. Also, the developed approach has been assessed against the evaluation criteria that were deducted from these research questions.

1.7 Key Research Contributions

In this research project, we propose a novel model-driven DSM approach to TEE. Below we summarize the key contributions we have achieved in the SIT area.

A New Approach to Testing Environment Emulation – Our TEE approach simplifies endpoints with external behaviors only, and shifts implementation focus from endpoint programming to modeling. Not only can our emulated testing environment capture all SUT interface defects, but also report the causes of these defects. Unlike some existing approaches [11], our emulation approach is suitable for both new application development and existing application upgrade (regression testing).

A New Software Interface Description Framework – A DSM approach achieves high development productivity by focusing on a narrowly applicable domain. We adopt the horizontal DSL development approach[3] and propose a new software interface description framework. This framework abstracts software interface into three horizontal (functional) layers and a number of vertical (QoS) attributes. The horizontal layers consist of signature, protocol and behavior. They process service requests in a step-by-step manner from signature, protocol, down to behavior layer. The vertical attributes model endpoint QoS aspects, such as security, performance, robustness, etc. We use a modular development architecture to model an endpoint – i.e. each module represents a particular interface layer/attribute.

A Novel Model-driven DSM Approach – Our approach consists of an endpoint modeling environment and a runtime testing environment. The modeling environment is supported by a suite of Domain-Specific Visual Languages (DSVLs), one for each interface abstraction layer/attribute. Domain experts use these DSVLs to model an endpoint by functional layers and QoS attributes. The endpoint models are transformed to executable forms by corresponding code generators embedded in the DSVLs. The

[3] A specific technical domain, not belonging to a specific industrial sector.

testing environment is generated by transforming an endpoint signature model in Web Services Description Language (WSDL) [22] form to Axis2 SOAP engine [20], integrated with the Java classes of other interface layers/attributes.

Full Code Generation from Models – The only artifacts for users to manipulate are models, and they are transformed to codes by code generators. To make the solution ease of use and increase development productivity, we provide a toolset to fully automate endpoint generation process from models.

Signature Layer Modeling – Signature layer defines message syntax with provided operations and their parameters. Our Signature DSVL supports Remote Procedure Call (RPC) communication style [23] and uses WSDL 1.1 specification as its metamodel. One of the key design considerations is to increase modeling productivity, and our solution is to adopt a three-level architecture design to improve components reusability.

Protocol Layer Modeling – Protocol layer specifies the allowable temporal sequence of service requests. To model runtime protocol behavior, we develop an Extended Finite State Machine (EFSM). The EFSM uses behavior model to capture endpoint returned values as state transition constraint conditions. In addition, our testing environment has a rich set of functions for simulating typical business scenarios, such as time-driven state transition, synchronous and unsafe operations.

Behavior Layer Modeling – Behavior layer processes service requests and generates responses after validating signature and protocol layers correctly. Our endpoint Behavior DSVL is designed based on dataflow programming paradigm. We choose this metaphor as it allows complex specification of behavior models but is understandable by a wide range of software stakeholders. To handle complicated business logics, we design our Behavior DSVL using hierarchical node tree structure.

Role-Based Access Control Modeling – Endpoint security attribute validates whether a service request is compliant with its endpoint security requirements. Our endpoint QoS security attribute is modeled using the popular role-based access control architecture. The role structure is designed in two orthogonal dimensions: a user's functional role in an organization determines which operations he/she can invoke, while his/her divisional role determines whether a resource can be accessed through an operation invoked by the user. To secure username and password in transit, we apply WS-Security policy to encrypt

UsernameToken. This demonstrates one example of augmenting our models with QoS requirements and generating QoS testing framework in the modeled endpoint implementations.

1.8 Thesis Structure

Having set our motivation and defined the key research questions, this thesis describes the research project in eight chapters. Following this introductory chapter, the remaining chapters are organized as followings:

Chapter 2 – reviews the state of the art in software testing and testing environment development. The chapter covers a broad range of software interface modeling techniques and discusses the implementation details of existing model-driven DSM approaches.

Chapter 3 – presents the first two phases of our modeling approach development process: decision making and domain analysis. The decision to use DSM approach is based on whether such an approach can potentially address the issues, derived from the three research questions. We then introduce our software interface description framework to abstract software interface and our approach architecture design. This is followed by a discussion of our metamodels for modeling the three functional layers.

Chapter 4 – describes the designs of the functional DSLs and implementations of their code generators and a domain framework. Specifically, how the principles of Moody's Physics of Notations are applied to optimize the cognitive effectiveness of the DSL visual notations and operational endpoints are generated automatically from models by code generators and a domain framework.

Chapter 5 -- presents a comprehensive case study to model endpoint functional layers. To demonstrate how both static and dynamic interface behaviors are modeled by our new DSM approach, a complex sales process with a few decision points is selected. These decision points are subject to the runtime conditions of request parameters or returned values.

Chapter 6 – describes the complete process to develop an endpoint security modeling solution as one of the endpoint QoS attributes. To model endpoint security aspect, the example used for our endpoint functional layer modeling is modified with multiple users involved in a sales process.

Chapter 7 -- discusses the evaluation process of our approach to endpoint emulation and presents the results. The evaluation is first conducted by a technical comparison with two existing kinds of TEE approaches. This is followed by a user study to assess the extent to which our approach is accepted by software testing experts and application developers.

Chapter 8 -- concludes the thesis. Proposals for future enhancements and extensions are also discussed in this chapter.

CHAPTER 2

Literature Review

Over the years, a number of approaches have been proposed to provide testing environments suitable for software professionals to test their software applications. These approaches generally fall into the following four areas: (1) system replication, (2) programmatic "Test Doubles" (e.g. method stubs and mock objects), (3) model-based testing solutions, and (4) Testing Environment Emulation (TEE). The first part of this literature review covers the details of each of these approaches and compares their advantages and disadvantages to address the issues related to our three overarching research questions. Limitations with these current approaches motivated our new Domain-Specific Modeling (DSM) approach to TEE.

To develop a DSM approach for TEE, we need to abstract the interactions between software applications and investigate existing approaches to model software interfaces. A Domain-Specific Visual Language (DSVL) for endpoint development is created from the interface model. Our second part of this literature review focuses on these two areas, and we describe them in separate sections.

We conduct our literature review by selecting the key proposals and solutions in each relevant technical area, and study what techniques have been adopted and how these techniques have been implemented.

2.1 Existing Approaches to Provisioning of Testing Environments

A testing environment is a setup of software and hardware on which a testing team performs testing of a newly developed or upgraded software applications [24]. This testing environment consists of physical hardware and software setups that include operating system, database server, front end running environment, back end data and any other software components required to run the testing applications. The ultimate goal of such a testing environment is to mimic all the systems' functionality and performance as close as possible to their real counterparts externally.

2.1.1 System Replication

In general, an enterprise's production environment is not suitable for conducting SIT [25]. Not only may it interrupt daily business operations, but also cause damages to the other systems and/or persistent data in the environment. System replication is clearly the most primitive alternative to replace production environment. It simply clones production systems to a non-production environment for providing the same application testing services or other operations on the systems or data as their real counterparts.

System replication is the construction and configuration of all the systems in an enterprise environment from ground up one-by-one [25]. It involves the activities of hardware setup, operating and database systems installation and production data transformation. A key advantage of using physical replication is its interface behavioral accuracy, which is effectively identical to the real applications. This is because all the hardware and software configurations and data are the exact same as the production environment. Software professionals will have high confidence in results obtained from the experiments conducted in such an environment.

On the downside, to setup and maintain a physical replication of a large-scale and heterogeneous environment will incur very high initial investment and operational cost [26]. Therefore, if a SUT is expected to operate in an environment of relatively small scale with a few software systems, then physical replication can be a good option. To make the approach more economically attractive, the systems in a replicated environment can be downgraded to basic configurations, if they are not going to be used for conducting performance testing. Alternatively, it may be possible to adopt a hybrid approach, with a replicated core system (e.g. legacy mainframe system) and peripherals generated by other cheaper solutions.

System-level virtualization tools, such as VMWare Workstation [27], VirtualBox [28] and Xen [29], create a virtual machine image that constitutes an abstract representation of a full physical machine. This virtual image can be subsequently executed by a host computer. Generally speaking, a testing environment provided by virtual machines is far easier to manage than physical replication [30]. This is because the number of virtual machines that can be hosted in a single high-end computer is up to twelve, and virtual machines allow applications to run in environments that do not suit the native applications [31]. Obviously, another key benefit from using these tools is the savings on hardware

investment. Thus, such virtualization tools are finding increasing use in industry as a means to provide interactive representations of software testing environments, as the substitution to physical replication [25].

However, these tools and virtual machines have some limitations [25]. Firstly, not all computer programs can be virtualized, some applications need to run in shared memory space. Secondly, provisioning the whole environment through virtual machines is costly for a large-scale environment. Thirdly, it is not always possible to host applications by virtual machines, as the applications may not be available or accessible (e.g. hosted in another organization or in Cloud). Lastly, application virtualization bears great licensing pitfalls mainly because both the application virtualization software and the virtualized applications must be correctly licensed.

2.1.2 Test Doubles -- Method Stubs and Mock Objects

A test double is an object that stands in for a real object in a test. A test double does not have to behave exactly like the real component, but it merely has to provide the same API so that the component under test thinks it is the real one [32]. Test doubles were originally created for software component unit tests. They have recently been enhanced to develop testing environments for application SIT. Based on their usages and implementations, test doubles are broadly classified into two categories of method stub and mock object [33].

A method stub is a piece of programming code that does not actually do anything other than declares itself and the parameters it accepts. It returns some hardcoded data that are usually the values expected by the caller. In testing, stubs are programs that simulate the behaviors of software components that a component undergoing tests depends on. Stubs are often used in top down testing approaches, when the main component is ready to test but some sub-components are still not yet. Stubs are the called components, which are called in to test the main component's functionality. From as far back as 1987, method stubs have been used to represent some basic interaction behaviors of remote systems [34]. Low cost and quick deployment are the key driving factors to motivate the use of method stubs for testing purposes. They are mostly useful when the testing is simple and keeping the hardcoded data in the stubs is not a big issue.

Mock objects have the same interface as the real objects they mimic, allowing a client object to remain unaware of whether it is communicating with a real object or a mock

object. They provide a response based on a given request satisfying predefined criteria (also called request or parameter matching). A mock object also focuses on interactions rather than states so mocks are usually stateless. They are mostly useful for a large suite of testcases, a stub may not be able to handle because each testcase needs a different data set up and maintaining them in a stub would be costly. Mock objects are normally created using language-specific third party libraries, such as Mockito [35] for Java, RSpec [36] for Ruby, and Mockery [37] for PHP.

However, there are a few key limitations when interactive representations of testing environments are provided through a stub method or a mock object [10]. Firstly, these approaches abstract away from communication complexities which may significantly impact on the results encountered in the real deployment. Secondly, they are usually language specific and thus not suitable to provision a generic testing environment. And lastly, the behavior of a stub/mock is programmed in an ad hoc fashion. It may not be possible to configure the behavior of a stub/mock at a high level, and any changes to the test plan may lead to changing the stub/mock implementation.

2.1.3 Model-Based Testing Solutions

Model-Based Testing (MBT) is a software testing approach, in which testcases are generated from a model to describe functional aspects of a SUT [38]. Models are the first order artifacts and are used to represent the desired behaviors of a SUT. MBT encompasses the processes and techniques for the automatic derivation of models from application requirements, the generation of testcases from models, and the manual or automated execution of the tests by using the testcases. In general, MBT is a five-step process illustrated by Figure 2.1 [39]:

1) A SUT test model is created based on the testing objectives deducted from the SUT application requirements. The model must be sufficiently precise to serve as a basis for the testcases' generation. But the model must also be simpler and easier to check than the SUT itself;
2) Test selection criteria are chosen to guide the automatic test generation. Test selection criteria can relate to the system functionality, structure of the test model and data coverage;

3) Testcase specification is a high -level description of desired testcases to formalise the notions of test selection criteria and render them operationally;

4) A set of testcases are generated with the aim of satisfying all the testcase specifications;

5) Test execution can be run automatically by a test execution environment that provides facilities to automatically execute the tests and record testing verdicts.

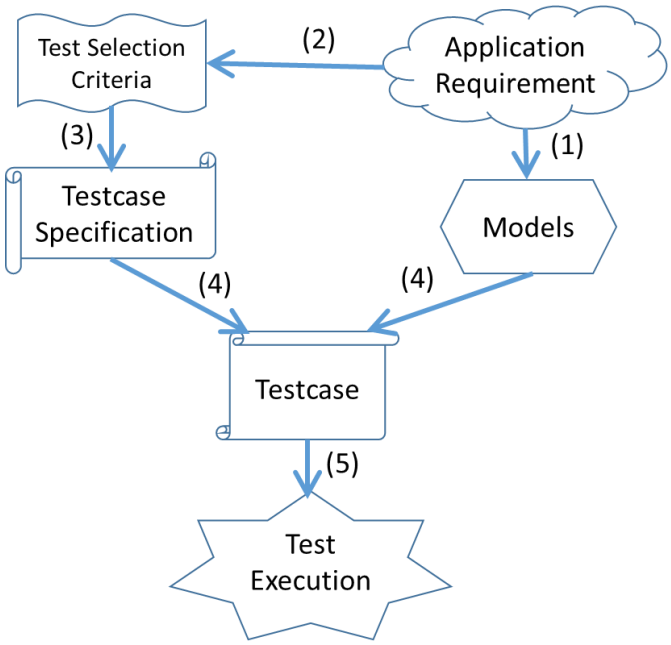


Figure 2.1. Model-Based Testing Process

Unified Modeling Language (UML) is a general-purpose modeling language intended to provide a standard way to visualize the analysis and design of a software system [40]. UML models focus on the definition of system structure and behavior, but provide limited means to describe test objectives and procedures. To make UML specifically for software testing, UML Testing Profile (UTP) [41] provides a generic extension mechanism for the automation of test generation and execution process by using UML Profile [42].

UTP enables the test specification for structural and behavioral aspects of UML models, and is capable to be incorporated with existing test technologies for black box testing. UTP has four testing concepts: (1) test architecture - test structure, test components and test configuration, (2) test data - data and templates used in test procedures, (3) test behaviors - dynamic aspects of test procedures, and (4) test time - time quantified definition of test

procedures. These concepts have been defined by using UML extension mechanisms of stereotypes, constraints or tagged values.

Another example of MBT approaches is Markov Chain Usage model, which is a stochastic process that satisfies the Markov property [43]. The Markov property is used to predict the future of a process based solely on its present state just as well as to know the process's full history. Markov Chain Usage model was developed to statistically analyse the population of all possible uses, generate a sample of tests, and reason about software reliability based on the sample performance. The model allows test input sequences to be generated from multiple probability distributions, making it more general than many existing techniques. Furthermore, the test input sequences generated from the chain are themselves a stochastic model, and can be used to create a second Markov chain to encapsulate the history of a test, including any observed failure information.

The Model Language (TML) is a language for describing Markov Chain Usage model, and it supports development, reuse, and automated testing [44]. TML design goals are: (1) to keep the language simple and syntax consistent, so learning curve can be flat; (2) to separate structural issues from statistical issues, developers can focus on developing the structure; (3) to provide a means to store several distributions in the model for use by tools; and (4) to attach arbitrary information as parts of the model for use by tools. Tools have been developed to parse TML and transform it to and from a number of other sources, including input/output formats used by the tools for graphic layout, numerical analysis, and automated testing.

The last MBT solution we discuss is a transition system to describe the potential behavior of discrete systems. Its diagram consists of states with transitions, labelled with actions. The states model the system state changes and the labelled transitions model the actions that a system can perform. Labelled transition systems constitute a powerful semantic model to reason about processes, such as specifications, implementations and tests. The IOCO (Input/Output COnformance) testing theory provides a sound and well-defined foundation for labelled transition system testing [45]. It is based on the testing equivalences and refusal testing theories. A formal implementation relation IOCO defines the conformance between implementations and specifications.

Most MBT approaches are used for server side application testing, rather than creating a testing environment for SUT integration testing [39]. Furthermore, there are two main problems with using UML to define new modeling languages [46]: one is usually hard to remove parts of UML that are not relevant to a specialized language; another one is that all diagram types have restrictions based on UML semantics. In addition, most existing models are not rich enough to generate signature, protocol and behavior tests like the kinds we want to support.

2.1.4 Approaches for Testing Environment Emulation

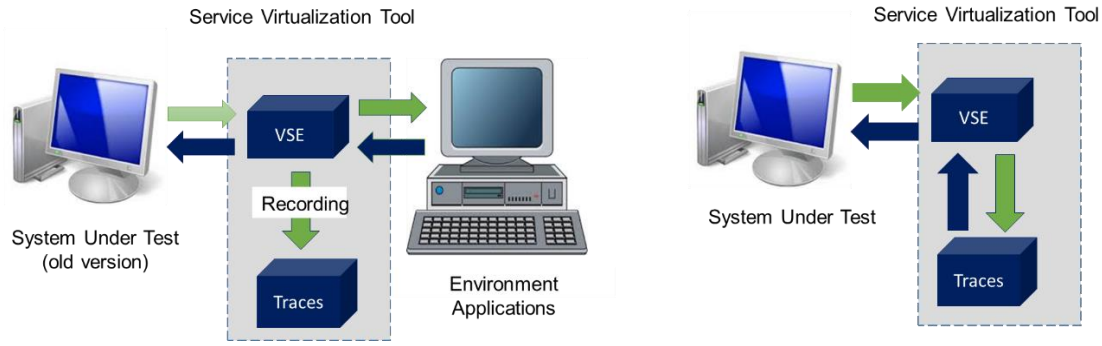
Testing environment emulation is a solution to emulate the behavior of systems in an enterprise environment for application testing [12, 13, 47]. The solution can be used to provide software development and testing teams access to dependent systems that are needed to exercise an application, but are unavailable yet or difficult to access. With the behaviors of the dependent systems virtualized, development and testing of the application can proceed without accessing the actual live systems. Currently, there are two types of TEE approaches: one is interactive trace data record-and-replay (or called interactive tracing), and another one is specification-based to specify external behaviors of an application. In the following, we discuss the implementation details of these two types of TEE approaches.

2.1.4.1 Interactive Tracing Approaches

Aiming to emulate the behaviors of services which a SUT interacts with in its deployment environment, the interactive tracing approaches mimic a response that a real service would return when receiving a request by the SUT. These approaches use a service virtualization tool (e.g. CA LISA [11]) as a proxy to sit between an earlier production version of a SUT and an endpoint application. The pairs of requests and corresponding responses exchanged between the SUT and endpoint are recorded and stored in the tool's database. Later those stored interactive tracing records are used to simulate the endpoint's response for each SUT's request by searching for the close-matching request in the database. Figure 2.2 illustrates how the interactive trace data are recorded (see Figure 2.2a) and virtual services are provided (see Figure 2.2b).

Realising how well the interactive tracing approaches can perform depends on the successfully matching of the new requests with ones stored in the trace record database,

Du et al. proposed a new method of Needleman-Wunsch longest common subsequence alignment to calculate the distances between a new request and those recorded requests [48]. The response that is corresponding to the closest matching request is the best one to be sent back to the SUT. Furthermore, symmetric field substitution is used to modify the sent response so that it is tailored to the new request. By applying this method to two commonly used application layer protocols, the results show that well-formed responses can be generated for all interactions [48].



[a] Interactive trace data recoding [b] Testing system under test

Figure 2.2. Interactive tracing approaches for TEE

On the performance front, Du et al. split service virtualization process into two consecutive stages, a pre-processing stage and a run-time stage [49]. Two popular clustering algorithms of BEA [50] and VAT [51] are utilized to pre-process a large amount of recorded interaction traces. With the obtained clusters, efficient yet well-formed runtime responses can be generated. Experimental results show that by utilising the clustering techniques in the pre-processing step, the response generation time can be reduced by 99% on average compared with existing approaches [49]. However, the cluster based approach can be applied only to those application-level protocols, where their message headers include a formal form of operation or service name. It will generate fewer valid responses for other protocols, such as LDAP.

2.1.4.2 Specification-Based Approaches

Specification-based approaches are used by IT professionals to manually develop simplified versions of applications with the approximate external behaviors as their real counterparts. They perform this using available application knowledge of the underlying message signature and interaction behaviors. Kaluta is the first specification-based

approach developed by Hine et al. [12]. From a high-level architecture point of view, Kaluta emulator contains three main components (see Figure 2.3):

- **Network Interface** -- The component has a service module to allow external SUTs to establish new channels with the emulator, and a conduit module to facilitate the interactions between external SUT and the emulator;
- **Message Process Engine** -- The engine is the host of the virtual endpoint system. Every endpoint is emulated as an instance of a message engine for executing the actual emulation logics specified by the behavior models of a system;
- **Management Component** -- The component is responsible for monitoring the overall system and each endpoint status and configuring all the endpoints in the emulator.

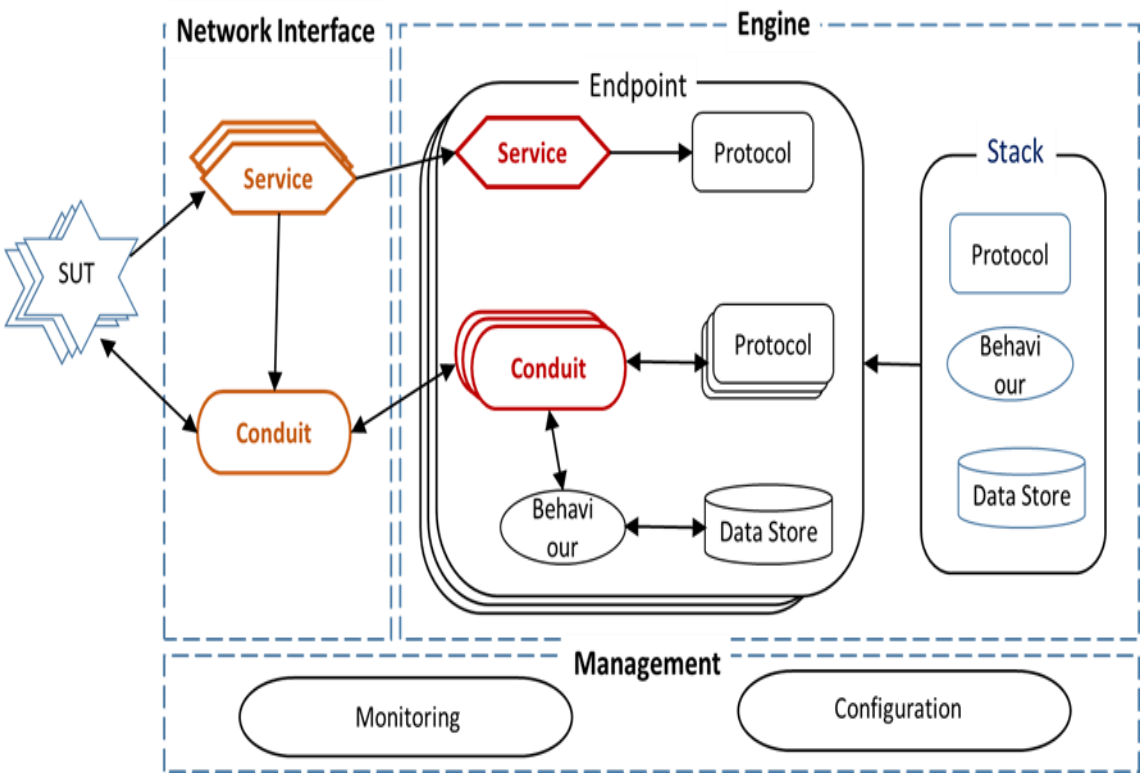


Figure 2.3. The Architecture view of Kaluta testing environment emulator

Coloured Petri Net (CPN) is a mathematical modeling language widely used in distributed system modeling and analysis [52]. Comparing to functional programming languages (e.g. Haskell) to create tasks like message processing engine, CPN has some advantages, such as graphical and executable representation and true concurrency property. For evaluating how well CPN can be used as a modeling language, Yu et al. replicated Kaluta emulator

and replaced its message processing engine written by a functional language Haskell with CPN [13].

To provide a technical comparison, the same experiments were repeated focusing on testing scalability and performance. From the experiment results, we conclude that CPN is also capable of emulating a large number of endpoint instances and provides a better modeling and graphic view than traditional ways of coding. To run the CPN message engine, endpoint abstraction models must be pruned to remove some unessential places and transitions for testing purpose. However, this may ignore some hidden message processing implementations and fail to validate some service requests under certain circumstances.

2.1.4.3 Other Test Bed Development Approaches

Other than the TEE approaches mentioned above, there are also a variety of approaches to testing environment development for some specific testing purposes. We present a performance engineering tool SoftArch/MTE, and a testing DSL Pact for implementing contract based testing.

SoftArch/MTE uses high-level system models to generate executable distributed test-beds [53]. As a similar approach to the predictive early-cycle performance models, SoftArch/MTE is intended to evaluate the performance of different architectures, design and deployment choices for a system while it is being developed. A working implementation of this system is then automatically generated from this high-level architectural description. This implementation is deployed to multiple client and server machines and performance tests are then automatically run for this generated code. Performance test results are recorded, sent back to the SoftArch/MTE environment and are then displayed to the system architect using graphs or by annotating the original high-level architectural diagrams. Essentially, SoftArch/MTE is an automated rapid prototype generation tool whose prototypes can be deployed at scale on real hardware to perform enterprise system performance evaluation.

Pact is an open-source tool, enabling consumer contract driven testing [54]. Pact provides a DSL for users to specify requests and expected responses from a service provider. To emulate a service provider for testing a SUT, the specs for all expected responses according to requests are written to a pact file. When the specs are run, the emulated

service returns the expected responses. The requests in the pact file are later replayed against the provider, and the actual responses are recorded and checked to make sure they match the expected responses.

Both of these approaches have their limitations for developing software testing environments. SoftArch/MTE can be used to develop those testing environments, which are only suitable for conducting performance tests and suffer from the same scaling and modeling problems as virtualisation and other model-based testing approaches. Another shortcoming is its scalability; many physical hosts are required to represent environments with many systems. Pact verifies each interaction in isolation without context maintained from previous interactions. Therefore, Pack is not suitable for performing protocol testing.

2.1.5 Comparison Summary

As a summary, we now provide a qualitative comparison of the existing approaches to emulate enterprise testing environment. This comparison is conducted from two orthogonal dimensions to measure the approaches' ability to develop endpoints and the testing functionality provided by endpoints, respectively. We add some important attributes to these two dimensions for their desirable features. The attributes for the development dimension are: (1) development productivity – the approach has high productivity to develop endpoints, (2) ease of use – the approach is easy to learn and use, and (3) investment – high initial investment is required to develop endpoints. The attributes for the testing functionality dimension are: (1) modality -- the endpoints are able to mimic the behaviors of a client (active), a server (passive) or both (dual), (2) accuracy -- the degree to which the interactive behaviors of an endpoint are accurate with respect to the real application on which the behaviors are based, and (3) information reporting – the endpoints report defect information, such as defect types and causes.

We provide a four-point rating subject to the level of support (None, Low, Medium or High) the approaches provide for each attribute, except for the Modality attribute with a value of active, passive or dual. Table 2.1 summarizes the comparison results. We briefly discuss how these approaches support the attributes and justify our subjective ratings below:

- **Development Productivity** -- Interactive tracing approaches are the highest, as endpoints are generated automatically by recorded interactive trace data. They are followed by replication approaches, where development work involves

Table 2.1. Approaches comparison for testing environment development [13, 14, 25]

Categories	Approaches & Tools	Development			Testing Functionality		
		Development Productivity	Ease of Use	Investment	Modality	Accuracy	Information Reporting
Replication	Physical Replication	High	High	High	Dual	High	None
	System-level Virtualisation [27-29]	High	High	Medium	Dual	High	None
Test Double	Method Stub [34]	High	High	Low	Active	Low	None
	Mock Object [35-37]	Medium	Medium	Low	Active	Medium	None
Model-Based Testing	UML Testing Profile [41]	Medium	Medium	Low	Passive	Medium	High
	The Model Language [44]	Medium	Medium	Low	Passive	Medium	High
	The IOCO-testing Theory [45]	Medium	Medium	Low	Passive	Medium	High
Testing Environment Emulation	Interactive Tracing [47]	High	High	Medium	Active	High	None
	Specification-based [12]	Low	Low	Low	Active	Medium	High
Other Approaches	Pact [54]	Medium	Medium	Low	Passive	Medium	High
	SoftArch/MTE platform [53]	Medium	Medium	Low	Passive	Medium	High

applications' installation and configuration only. Test double approaches are also rated as high, as they only require simple programming. In contrast, specification-based approaches develop endpoints using traditional manual coding, their productivity is the lowest;

- **Ease of Use --** Replication approaches are the easiest, as developers just follow the manual instructions to install endpoint applications. To adopt interactive tracing approaches, a short training course is needed to use the service virtualization tool, but their use is very simple and easy. Therefore, we rate these approaches as high next to replication approaches. Method stub approaches only need to hard code request and response signatures, and they are also considered as easy to use. In contrast, specification-based approaches require users to have both application knowledge and programming skills, they are the most difficult to use;
- **Investment** – No doubt, replication approaches have the highest initial investment cost, as both hardware and software licenses are needed. Interactive tracing approaches need a special service virtualization tool, so they are rated as medium. All others are low, they do not have any special requirement on hardware and software;
- **Modality** – Replication approaches can be used for testing both client and server, and they are dual. Test double and TEE approaches are active, as they provide a testing environment for testing requests from clients. In contrast, all the other approaches are passive, they are used to generate testcases for validating responses from servers;
- **Accuracy –** Obviously, replication approaches have the highest accuracy, their behaviors are almost identical with their real counterparts. The behavior accuracy of interactive tracing approaches depends on finding the closest match in the trace records for each service request, their accuracy is also considered as high. Mock stub approaches are the lowest, as their responses are hard coded;
- **Information Reporting** – Replication approaches do not report defects causes, as they will not be modified for testing purpose. Interactive tracing approaches only tell whether a testcase is passed or failed, but do not report defect information.

Similarly, test double approaches mainly act as APIs to accept requests, they cannot provide defect information. All these kinds of approaches are rated as none.

2.2 Software Interface Abstraction and Modeling

We realize our DSM approach by abstracting software interfaces into different functional layers and attributes, and each of the layers/attributes represents a specific modeling domain. We investigate the current state of the art in this area and discuss the detailed treatments of the existing techniques and solutions to model software interfaces.

2.2.1 Software Interface Abstraction

For software systems to communicate with each other, they must be able to understand the content of the messages being sent over from their communication partners and accept them under current circumstances. Thus, the correctness of message syntax only may not be good enough, and some runtime constraints might jeopardize their communications. To address these static and dynamic issues related to software components interactions, Han proposed a comprehensive interface definition framework to abstract software component interfaces into four logic layers [55]: (1) signature – to characterize the component functionality and form the basis of all other aspects of the component; (2) constraint – to enforce restrictions on the use of the component; (3) configuration – to define the component to be used in different context types and have different roles in a given context; and (4) non-functional properties – to enforce additional constraints on the use of the component, such as security, performance and reliability aspects.

From a service viewpoint, Beugnard et al. defined a four-level software component contract template with increasingly negotiable properties along with the levels [56]. The first basic or syntactic level is required simply to make the component work. The second behavioral level improves the level of confidence in a sequential context. The third synchronization level improves confidence in distributed or concurrency contexts. And the highest Quality-of-Service (QoS) level is usually negotiable with its client. Different from Han's interface framework, a synchronization contract was added to specify the global behaviors of software components in terms of synchronizations between method calls.

Recently, Hine developed a layered interaction model to describe software interface behaviors [25]. The model consists of three functional layers in a top-down manner: (1)

protocol layer – allowable temporal sequence of service requests; (2) behavior layer – service request process and response generation; and (3) data store – persistent data access and manipulation. In order to conduct protocol and behavior study separately, the behavior layer called by Beugnard et al. [56] was split into two different layers. The protocol layer defines the rules for the validity of client service requests. Behavior and data store layers are used to specify the server response for each service request.

By abstracting software interface into layers, these interface description approaches are able to describe not only the static aspects of software interface, but dynamic ones as well. Furthermore, different interface aspects are encapsulated so that modular software development process can be applied. However, Han's interface definition framework and Beugnard's component contract were not specifically designed for creating endpoints, as layered processing sequence is not specified for coming requests explicitly. On the other hand, Hine's interaction model does not include a signature layer and QoS attributes. Thus, service operations, their parameters as well as QoS aspects cannot be specified.

Our DSM approach to TEE consists of three horizontal layers and a number of vertical attributes [26]. In the following, we look into the technical details of existing solutions to model the three horizontal layers of signature, protocol and behavior, and a vertical attribute security as well.

2.2.2 Software Interface Syntax Specification

Software interface syntax defines the name and parameters of a service request and the parameters of the corresponding response. It forms the most basic and lowest level of a software interface specification.

In a distributed computing environment, a service consumer can access and execute services provided by a remote service provider through Remote Procedure Call (RPC) using request-response message passing protocol [23]. An RPC communication is initiated by a client, sending a request message to a known remote server for executing a specified procedure. The remote server sends a response back to the client after processing the request, then continues its process. An RPC can fail because of unpredictable network problems, and the caller (client) is generally responsible for handling such failures without knowing whether the remote procedure was actually invoked.

An Interface Description Language (IDL) is a specification language used to describe software application (or component) programming interfaces [57]. IDLs specify language-independent interfaces to communicate between clients and servers in an RPC. Common Object Request Broker Architecture (CORBA) is an OMG standard designed to facilitate the communication of systems that are deployed in distributed and diversified platforms [17]. CORBA uses an IDL to specify the interfaces that present objects to the outer world and maps the IDL to a specific implementation language like C++ or Java. Java Remote Method Invocation (Java RMI) is a Java API that performs remote method invocation, and it is the object-oriented equivalent of RPC [58]. RMI accesses distributed business objects from another Java Virtual Machine (JVM) by using object serialization to marshal and unmarshal parameters.

SOAP (Simple Object Access Protocol) is a protocol specification for exchanging XML message in the implementation of Web services [59]. It relies on application layer protocols, such as Hypertext Transfer Protocol (HTTP) [60] or Simple Mail Transfer Protocol (SMTP) [61], for message negotiation and transmission. A SOAP message consists of an envelope to define the start and end of the message, a header containing any optional attributes of the message used in processing the message, a body comprising the XML message being sent, and an optional fault element.

REpresentational State Transfer (RESTful) Web services provide interoperability between computer systems on the Internet. They allow requesting systems to access and manipulate textual representations of Web resources using a uniform and predefined set of stateless operations [62]. The operations applying to the resources include the five predefined HTTP verbs of GET, PETCH, POST, PUT and DELETE. Six constraints are used to restrict the ways that a server may process and respond to client requests. By applying these constraints, RESTful Web services have many advantages over other Web services (i.e. SOAP), such as less overhead, less parsing complexity, statelessness, and tighter integration with HTTP.

Web Services Description Language (WSDL) is an XML-based specification schema to describe the details of public interface exposed by a Web service, including what a service does, where it resides, and how to invoke it [22]. It consists of a service interface definition to define specific type of interface provided and a service implementation definition to describe how a particular service interface is implemented. WSDL is often

used with SOAP protocol and XML Schema [63] to provide Web services over Internet. A client program reads the WSDL file to determine what operations are available on the server and what parameters are required to access these operations. The current version 2.0 supports both RPC and RESTful communication styles.

RESTful provides a simple and clean access to a URL resource, and the response is a straight XML document. The key benefits from using RESTful include its performance and scalability. On the other hand, SOAP has the flexibility to specify certain aspects to treat messages by using its header element. This is very helpful for us to specify some essential QoS attributes, such as security. So, WS-Security [64] can be easily implemented to add security aspect to endpoints. WSDL supports the contract first design approach [65] to describes a service contract by defining name, location and operations, inputs and outputs of the service. Apache Axis2 SOAP engine [20] includes a tool wsdl2java to implement WSDL service contract.

2.2.3 Protocol Modeling

Protocol layer defines the allowable temporal sequence of endpoint service operations. The validity of endpoint protocol depends on the current endpoint state and/or some runtime constraints in a stateful communication.

UML state machine diagram (also called Finite State Machine, FSM) shows discrete behaviors of designed systems through finite state transitions [66]. Specifically, its Protocol State Machine (PSM) can be used to express a system usage protocol and its lifecycle. PSM specifies which operations of a system can be called in each state of the system, under which specific conditions, and satisfying some optional post-conditions after the system transiting to a target state. Some key concepts included in UML FSM are: (1) state -- its value defines the current state of the system; (2) event (trigger) -- an occurrence in time that has significance to the system; (3) action -- when an event instance is dispatched, the system responds by performing actions; and (4) transition – an event may trigger state change from one to another. An FSM supplemented with trigger condition variables is called Extended Finite State Machine (EFSM). An EFSM transition is associated with a guard boolean variable, which means that the transition can be triggered only if the guard variable evaluates to TRUE [67]. Figure 2.4 illustrates a basic FSM to represent online shopping account lifecycle.

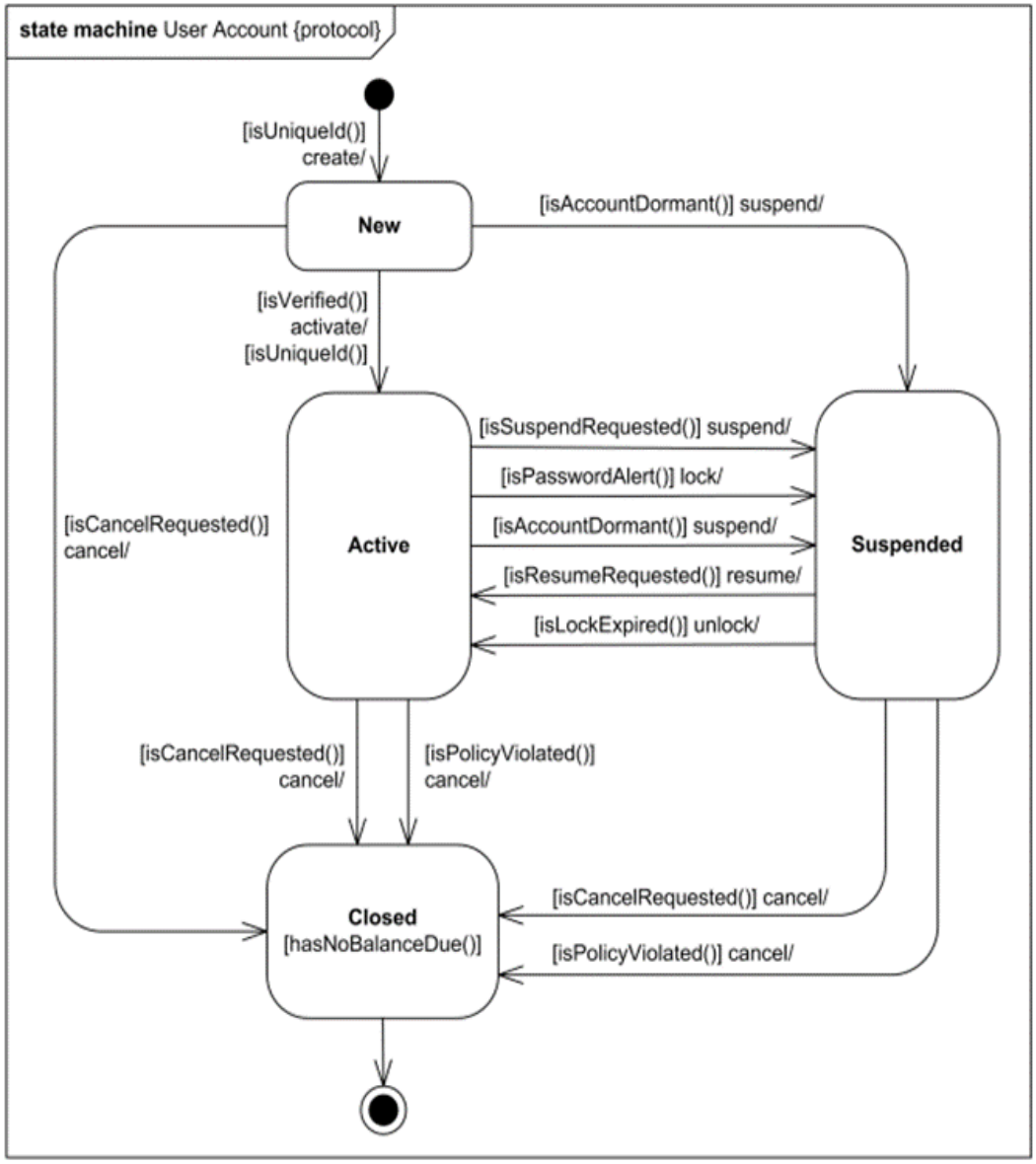


Figure 2.4. An example FSM to represent online shopping account process

As an FSM can be mathematically represented easily and also provide enough expressiveness for modeling endpoint protocol behavior, it has been widely used to validate operation sequence based on endpoint states [68-71]. However, Wehrheim et al. argued that the use of operation name alone may not be sufficient enough to trigger a state transition for a realistic endpoint in the real world [72]. Some runtime aspects are also needed to enrich interfaces with protocol behavioral constraints. To deal with the so-called incomplete protocol specification, they developed an EFSM-based protocol modeling calculus, which allows protocol specification with operation parameters and return values as runtime constraints. Closely related to our modeling approach but targeting on embedded systems, Moffett et al. [73] proposed a Model-Driven

Development (MDD) approach to specify component protocol behavior. This approach uses a PSM to describe allowed message exchanges between two components across a connector.

To specify unambiguous protocol behaviors, some researchers have developed formal protocol specification languages with protocol specific temporal operators，constraint constructs, relationships and other constructs [74-76]. These languages are mainly used to specify the causal and temporal inter-dependencies among service providers and consumers involved in a business process. It is worth to mention the work done by Hine et al. that Process Algebra is used to specify a concise high-level abstract modeling syntax for application-layer protocols in concurrent and asynchronous environments [77]. They defined 14 message evaluation rules to test the validity of a message trace with regards to a given protocol specification.

A typical Web service requires its operations to be invoked in a certain order to progress the interactions of a conversation through to completion. Web Service Conversation Language (WSCL) specifies the allowable sequencing of Web service document exchanges in a standard way [78]. WSCL orchestrates the various message exchanges that occur at each stage of a conversation between the provider and consumer of a service. WSCL and WSDL are highly complementary – WSDL specifies the syntax requirement of a message to be sent to a service and WSCL defines the allowable sequence in which such a message can be sent.

Web Service Choreography Description Language (WS-CDL) is W3C specification of choreography model to describe collaboration protocols of cooperating Web service participants in a business process [79]. The purpose of WS-CDL is to define multi-party contracts, which describe the externally observable behaviors of Web services and their clients. A choreography model describes a collaboration between a collection of services to achieve a common goal. It captures the interactions of which the participating services engage to achieve the common goal and the dependencies between these interactions, including control-flow dependencies, data-flow dependencies, message correlations, time constraints, and transactional dependencies.

Both UML state machine diagram and WSCL are automaton based. They validate service operations by endpoint state, and the endpoint state changes are driven by accepted

operations. However, they can only validate the static endpoint protocol and runtime behaviors cannot be captured. WS-CDL is mainly for collaborating composite services in a business process, and it is not suitable for specifying the temporal sequence of a particular endpoint. On the other hand, formal protocol specification languages are capable of defining both static and runtime protocol behaviors. But their textual form makes them not ease to use, and the textual programming operators are difficult to convert to visual constructs for visual languages. EFSM is a powerful and yet easy to use technique to model endpoint protocol behavior. Its state entity and transition function allow users to specify endpoint static protocol aspects, and constraint entity can be used to specify extra runtime conditions for restricting some endpoint state transitions.

2.2.4 Behavior Modeling

Endpoint behavior involves processing service requests and generating responses. In this research project we assume endpoints are stateful, i.e. persistent data is needed for storing a session information when the endpoints process service requests. To support this, data stores are included in our endpoint behavior model to store endpoint state information.

Behavioral Interface Specification Languages (BISLs) provide formal programming constructs, such as pre/post-conditions, invariants, and assertions for allowing developers to express the intended program behaviors. Java Modeling Language (JML) is a Java BISL implementation to specify classes and interfaces [80]. As a BISL, JML describes two important Java module aspects of interface and behavior. The former consists of the names and static information found in Java declarations; and the latter tells how the module acts when used.

Hatcliff et al. [81] conducted a survey on how behavioral interface specifications are adopted by different languages and how interface behavioral aspects are handled by their specific programming syntaxes. For representing the behavior of a functional module, they introduced the Floyd/Hoare Logic [82, 83] triples of the form:

$$[P]\ C\ [Q],$$

where both the pre-condition P and post-condition Q are boolean formula, and C is a program statement. To enhance the triples of the form's expressive power, some programming constructs are used to specify the pre/post-conditions.

Programming from specifications converts system requirement to executable codes in a step-wise process by complying with certain refinement laws [84]. Its theoretical foundation is the predicate calculus, where a variable *w* can be defined by the formulae:

$$w: [pre\text{-}condition, post\text{-}condition]$$

The necessary infrastructures include a collection of predicate calculus laws and mathematical types. They can be used to build up more advanced programming constructs. It is worth to mention the sequential composition law:

$$w: [pre, post] \in w: [pre, mid]; w: [mid, post],$$

which indicates that one complicated operation can be split into two or more simpler sub-operations.

Business Process Execution Language (BPEL) is a standard to define and manage business process activities and business interaction protocols by collaborating Web services from an orchestration point of view [85]. BPEL uses WSDL to describe the peer-to-peer interactions between services and specify the activities that should take place in a business process. BPEL contains five distinct sections: (1) message flow, (2) control flow, (3) data flow, (4) process orchestration, and (5) fault and exception handling. There are two levels of process description: abstract and executable business processes. The abstract process specifies the external message exchanges between Web services but ignores internal details of the business process. The executable process defines both the external message exchanges and the complete internal details of business logic.

Business Process Model and Notation (BPMN) is a graphical representation for specifying business processes in a business process model [86]. BPMN provides a notation, that is intuitive to business users yet able to represent complex process semantics for IT professionals. BPMN specification defines four groups of grammar constructs: (1) flow object – the basic elements used to create business process diagrams; (2) connecting object -- to connect flow objects through different types of arrows; (3) swimlane -- to group activities into separate categories for different functional capabilities or responsibilities; and (4) artefact -- to display further related information such as processed data or other comments. Figure 2.5 illustrates an example BPMN diagram, showing a

typical HR recruitment process with two swimlanes of HR and Manager of an organization.

Data Flow Programming (DFP) is a programming paradigm that implements dataflow principles and models a program as a directed graph with data flowing in and out processing units (or called "nodes") [87]. When the program begins, special activation nodes place data onto certain key input arcs, triggering the rest of the program to run. Data flow in each node from its input connector, and the node starts to process and convert the data whenever it has the minimum required parameters available. The node then places its execution results onto its output connector for the next nodes in the chain. Except for the nodes and arcs, database operators are needed to retrieve and manipulate persistent data.

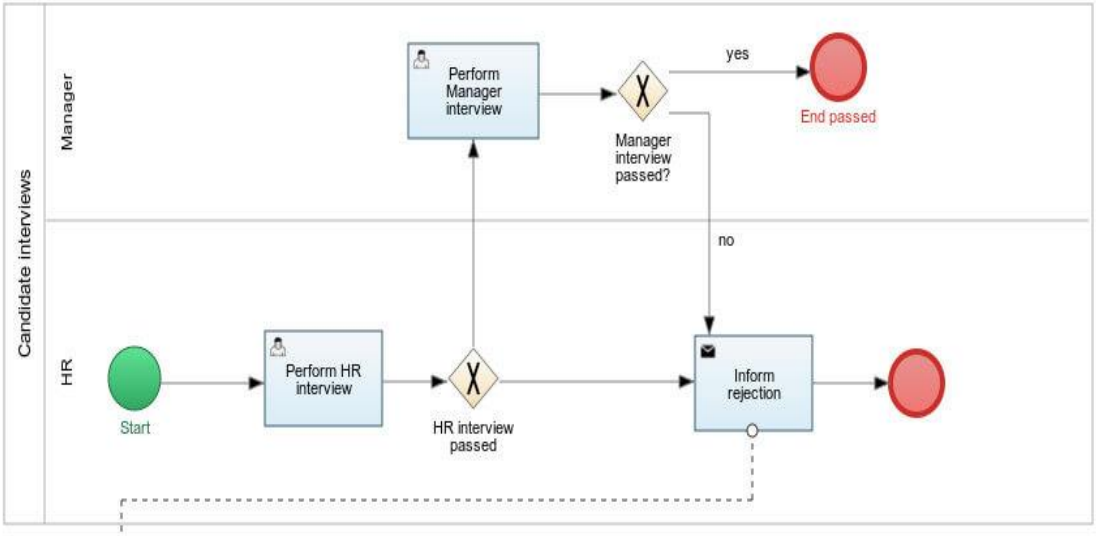


Figure 2.5. An example BPMN diagram for a recruitment process

To handle complicated business logics and better manage the diagrammatic complexity problem, standard DFP approaches can adopt a hierarchical tree structure and other view optimization techniques [88]. Each node may contain several sub nodes, and each of sub nodes executes a specific task. For improving expressive power and simplifying visual presentation, some DFP languages incorporate with object-oriented programming paradigm and provide extensibility and reusability to building blocks. Figure 2.6 depicts a dataflow programming usage example, using circle to represent nodes, arrow line for arcs, open rectangular box for data store operators and rectangular box for external systems.

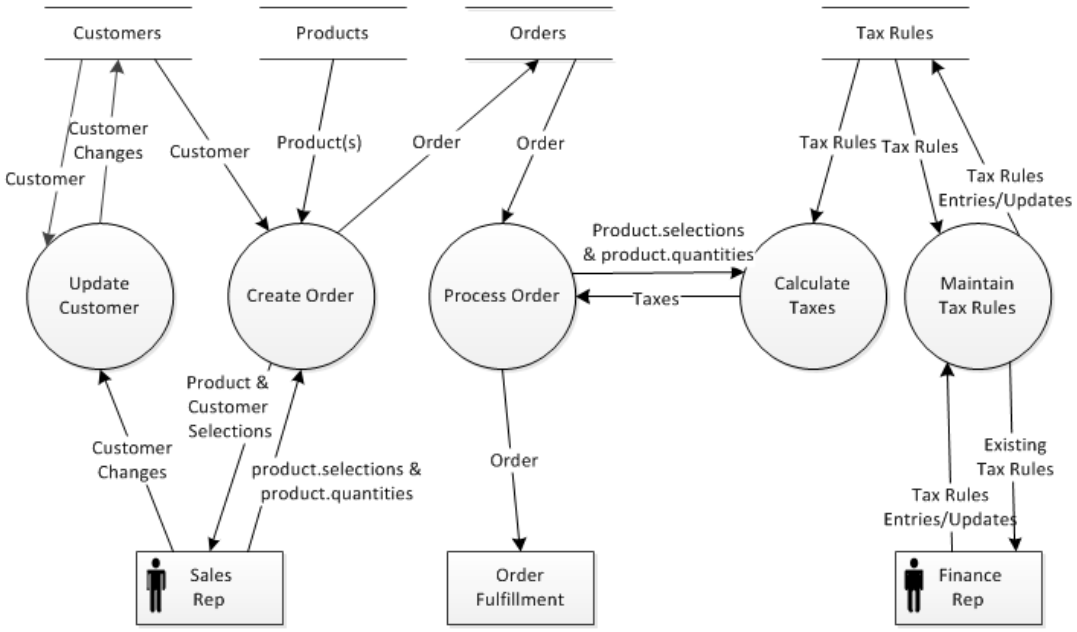


Figure 2.6. An example DFP diagram for a CRM system

LabVIEW [89] is a commercial pure DFP language, specially designed for digital circuit testing domain. Its graphical approach allows non-programmers to build programs by dragging and dropping virtual representations of lab equipment with which they are already familiar with. ProtoHyperFlow (PHF) is a general purpose DFP language with object-oriented programming features [90]. Its building blocks VIPs (Visually Interactive Processes) consist of a mailbox and body. The mailbox holds a discrete data object, and the body is the semantic content (implementation of the semantics). PHF implements higher order functions, allowing functions to be passed to/from functions as data. Similarly, Prograph language is another object-oriented DFP language [91]. It is class-based with single inheritance and dynamic typing. Polymorphism allows each object to respond to a method call with its own method appropriate to its class.

BISLs and programming from specifications model an endpoint behavior from its external interface by specifying pre- and post-conditions. They facilitate the Design by Contract (DbC) programming style [92]. In DbC, a client's obligation is to meet the pre-conditions of the contract when it calls a server operation. On the other end, its server should terminate the operation execution properly and generate an output meeting the post-conditions in the contract terms. On the other hand, BPEL, BPMN and DFP provide graphical notations for specifying internal data processes and flow controls between different nodes inside endpoints. In general, the external approaches, BPEL and BPMN

have rich expressive power to handle complicated business logics, but they require extensive programming and modeling work. Pure DFPs are easy to use by dragging-and-dropping visual symbols, but they are less expressive and only suitable for a narrowed domain. Most object-oriented DFPs are powerful, but they require users to have object-oriented programming skills.

2.2.5 Security Modeling

Security is an endpoint QoS attribute for modeling security rules enforced on service requests. An endpoint should validate the security requirements of a service request first before actually processing it.

Security requirements have become complex in order to deal with diverse and constantly changing threats. Security is also a risky area, as any minor and obscure oversights can lead to serious vulnerabilities. Security modeling is a formal approach to analyse system security, support comparative evaluation, and develop useful insights into design, implementation and deployment decisions. A security model often has three components [93]: (1) a system model -- a clear definition of the system of interest to understand how the system behaves; (2) a threat model -- a clear definition of the vulnerabilities of computational resources and system access; and (3) security properties – a clear definition of the properties to prevent the risks from violating security requirements.

Model-Driven Security (MDS) has emerged as a specialized Model-Driven Engineering (MDE) approach [94] for supporting the development of security-critical systems [95]. MDS models security requirements at a high abstraction level, then transforms them into enforceable security rules with as little human intervention as possible. Using MDS approaches can bring several benefits to security-critical system development [96]. Firstly, security concerns can be modeled explicitly from the very beginning and throughout the whole development lifecycle. By doing so, security requirements can be seamlessly integrated into system's architecture design and delivery code. Secondly, by separating security concerns from business functionality, security experts can therefore focus on security related issues, instead of dealing with other technical problems. And lastly, MDS leverages on the MDS automation provided by model transformations such that human interference, which is naturally error-prone, is reduced.

There are many security control models [97-100] in use to restrict systems and data access to authorised users, covering from basic permission assignment for small companies to comprehensive role-based and attribute-based control mechanisms implemented by large international organizations. In general, the choice of a security control model largely depends on the level of complexity of an organization structure and its business process. In this section, we first introduce a popular and an emerging security model. Then, two UML based MDS approaches are presented as our case studies.

Role-Based Access Control (RBAC) is a modeling approach to restrict system access to authorized users. It is used for security administration and review of enterprise systems. The central notion of RBAC is that system permissions are associated with roles, and users are assigned to appropriate roles [101]. Roles are created for the various job functions in an organization and users are assigned to roles based on their responsibilities. Roles can be granted new permissions when new applications and systems are incorporated, and permissions can be revoked from roles as needed. RBAC supports three well-known security principles: (1) least privilege -- only those permissions required for the tasks conducted by members of the role are assigned to the role; (2) separation of duties -- mutually exclusive roles must be preserved for sensitive tasks; and (3) data abstraction – object permission is specified for the intended use, rather than the read, write, execute permissions typically provided by operating systems.

To unify the different notations from frequently referenced RBAC models, National Institute of Standards and Technology (NIST) proposed a RBAC standard with a set of formalization mechanisms [99]. NIST RBAC model is defined in terms of four model components. The first one is Core RBAC to form the basic RBAC model. The second one is Hierarchical RBAC with additional requirements for supporting role hierarchies. The third one is Static Separation of Duty Relations to prevent a user from gaining authorization for permissions associated with conflicting roles. And the last one is Dynamic Separation of Duty Relations, which contains the constraints on the roles that can be activated within or across a user's sessions.

Figure 2.7 illustrates NIST Core RBAC model concepts [99]. Core RBAC includes five entities called users, roles, objects, operations, and sessions. Their relationships are: (1) resources and operations are part of an application, and resources are assigned to

operations; (2) roles are given permissions to access the application operations; (3) users are assigned to roles; and (4) users and roles are associated with sessions.

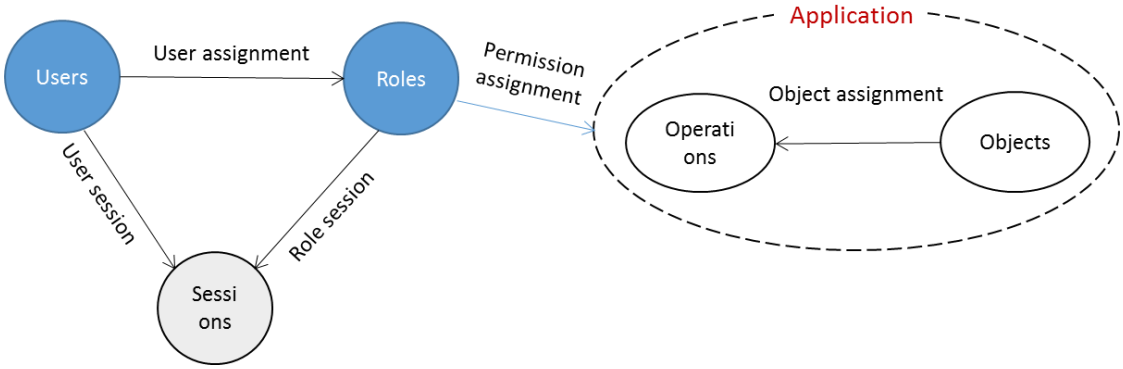


Figure 2.7. NIST Core RBAC model

Attribute-Based Access Control (ABAC) is a new comprehensive security modeling approach to grant access rights to users through the use of security policies, which combine attributes of requester and object, operations, and the environment relevant to a request together [100]. This model supports boolean logic operations, in which rules contain "if, then, else" statements about the requester, resource, and action. Compared to RBAC model, ABAC enables more precise access controls on protected resources by allowing for more discrete inputs into an access control decision. ABAC is considered as the next generation authorization model because it provides dynamic, context-aware and risk-intelligent access control to resources in distributed computing environments.

UMLsec is a pioneering work on MDS domain, adding an extension to UML for integrating security related information in UML specifications [102]. In order to specify security requirements and assumptions on top of standard UML, UML stereotypes are used with annotations. UMLsec combines several UML diagrams for analysing and modeling system's security aspect: class diagram for static structure, state machine for dynamic behavior, interaction diagram for object interactions within distributed systems, and deployment diagram to enforce security in the target platform. UMLsec takes the advantage of the wide spread use of UML as a general-purpose modeling approach.

SecureUML is a modeling language designed for integrating the security specification of access control into application UML models [103]. The language builds on RBAC with support for formalizing authorization constraints specified by UML Object Constraint Language (OCL) [104]. OCL expressions can be applied to all application model types

and allow considerable flexibility in defining authorization constraints. Since OCL is a first-order language, constraints can be incorporated into a formal analysis of UML models. The key benefits from using SecureUML is the support for both role-based and programmatic access control models and adaptable to different security architectures.

Although ABAC provides a more comprehensive protection to systems and data from being accessed illegally, RBAC is still the preferred security model. There are two main reasons behind that. One is that RBAC implementations are far more than any other access control models. Most enterprise software applications are compatible with RBAC, and the majority of medium to large scale companies adopt RBAC model to control their systems and data accesses [105]. Due to the popularity, there are many tools and solutions supporting RBAC, such as SecureUML. Another one is about ABAC, many security attributes are application specific and dependent on individual companies. It is not easy to develop a generic ABAC based security control modeling solution.

As discussed earlier, both Han's interface definition framework [55] and Beugnard's component contract [56] include an abstract layer for defining QoS attributes. However, based on our knowledge all existing TEE approaches [12, 13, 48, 106] do not support QoS modeling, including security attribute.

2.3 Development of Domain-Specific Modeling Languages

DSM development processes and techniques are more varied than those developing software applications, as DSM approach requirements are often vague and unstable [107]. Having selected model-driven domain-specific modeling as our testing environment development approach, we look into the existing processes and related techniques to develop such an approach.

To aid software developers to develop DSM approaches, Mernik et al. conducted a systematic survey to identify the common patterns of existing DSM development approaches [108]. They split a DSM development lifecycle into five sequential phases: (1) decision making – to make decision in favour of a new DSM development; (2) domain analysis – to identify problem domain and gather domain knowledge; (3) DSL design – to design DSL programming constructs and syntax; (4) implementation – to implement code generators and a domain framework, and (5) deployment -- DSM and the applications constructed with them are deployed to a working environment.

Among these phases, the decision making is more from organizational and economic consideration for most commercial DSLs' development [4]. On the other end, the deployment phase is system environment dependent. There are not many generic solutions applicable to all organizations. Therefore, these two phases are excluded from this review. In the followings, we discuss the existing DSM development techniques and solutions for the domain analysis, DSL design and implementation phases.

2.3.1 Domain Analysis

Domain analysis is the DSM term of system analysis in software development. The term of domain analysis was first introduced by Neighbors [109] as "the activity of identifying the objects and operations of a class of similar systems in a particular problem domain." To accomplish this goal, domain analysts must be able to extract, organize, represent and manipulate all domain specific objects and the relationships among them [107].

Inputs for a domain analysis are the various sources of domain knowledge, such as technical documents, business knowledge provided by users and domain experts, existing example applications, and customer surveys. The outputs mainly include the domain scope, domain terminologies, descriptions of domain concepts, and a metamodel describing the commonalities and variabilities of domain concepts and their interdependencies. The information gathered in this phase can be used to design and implement a DSM approach to model all applications in the domain. Variabilities among domain applications should be specified in DSLs, and commonalities are pre-coded to the domain framework of a DSM approach.

The metamodel output from domain analysis phase can be described by a class diagram, which specifies domain entities, their attributes, operations, and the relationships in a specific domain. Figure 2.8 illustrates an example metamodel of a warehouse definition. A container has racks and boxes, and a rack contains boxes. A container also contains elements, and an element has items. And finally, a box has items in it.

In analogy to traditional Waterfall [110] and emerging Agile [111] software development processes, Visser grouped domain analysis methods to deductive and inductive categories [112]. The deductive domain analysis methods follow a sequential (non-iterative) or called top-down process. The target domain is analysed exhaustively, and all domain

[4] As a research project, we make our decision to use a DSM approach from a technical point of view.

aspects are abstracted before progressing to the next development phase. On the other hand, the inductive methods introduce domain abstractions incrementally for allowing a set of domain concepts to be captured in a cycle. These domain abstractions are used to develop an intermediate working DSL with adding functionality on the top of the previous releases.

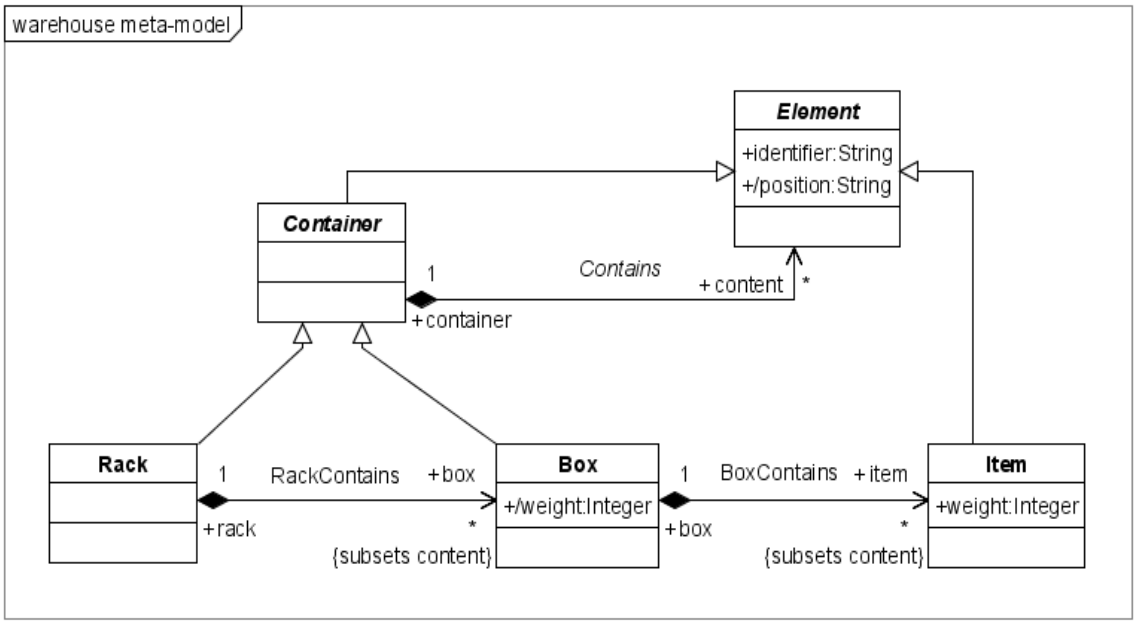


Figure 2.8. An example metamodel of a warehouse definition

The deductive domain analysis methods are based on the Waterfall software development model, which still dominates software development by now [110]. One of the key advantages from using Waterfall model is that the time spent early in the software production cycle can reduce the costs associated with defect fixing at later stages. Surveys have found that a defect found in an early stage is cheaper to fix than the same defect found later on in the process by a factor of 50 to 200 [113]. Another advantage is its structural approach where development process progresses linearly through discrete, easily understandable and controllable phases. However, a key shortcoming for those deductive methods is the risks in terms of schedule delay and mismatch functionality. These can be caused by over design and late understanding of requirements, which could lead to the discovery of requirements and design problems late in the process [111]. We discuss two deductive and one inductive methods and explain the different ways they conduct domain analysis.

In the early 1990's, Prieto-Diaz [114] proposed a Structured Analysis and Design Technique (SADT) based deductive domain analysis method to identify the reusable components from similar applications. The inputs are technical documents, existing implementations, customer surveys, experts' advices, and requirements for current and future systems. Domain experts extract relevant information and knowledge from the inputs, then analyse and abstract them to domain concepts. The process is guided by specified domain analysis techniques and management procedures. The outputs include taxonomies, standards and domain models. A library based domain infrastructure was introduced as an attempt to show how domain analysis could be integrated into a software development process.

Feature-Oriented Domain Analysis (FODA) is another deductive domain analysis method to identify the prominent and distinctive features of software systems in a domain [115]. The method is a three-phase process to conduct domain analysis:

- **Context Analysis**: to develop a context model of the domain for defining the domain scope. Without appropriate scoping, domain analysis outputs can be too diffuse to meet the needs of application developers or omit some areas of the domain;
- **Domain Modeling**: to analysis the commonalities and differences of all the applications in the domain. This phase has three activities: feature analysis, entity-relationship modeling and functional analysis;
- **Architecture Modeling**: to provide a software solution to the problems defined in the domain modeling phase.

To minimize the risk to implement Web application DSL (WebDSL), Visser proposed a risk-driven inductive domain analysis method to capture a set of common programming patterns in software development for Web application domain [112]. Comparing to traditional domain analysis methods, this method has two unique characteristics. The first one is technology-driven, where the best practices are considered in the implementation of systems in the domain, rather than to analyse domain abstractions. After the initial determination of the domain scope, the domain analysis then focuses on exploring what technologies are available and how they can be used to develop the DSL. The second one is the iterative process to break the whole development work into small working increments to minimize the amount of up-front analysis and design. At the end of each

iteration, a working product is demonstrated to stakeholders for collecting their feedbacks. These feedbacks are analysed by domain experts and used as the basis for generating the requirements of the next increment.

2.3.2 DSL Design

DSLs are powerful tools for software application development, because they are tailor-made to a specific problem domain. DSL developers often have larger freedom in designing DSLs than their counterparts in designing applications, and they may face some difficulties to make the right decisions at the early stages. This is mainly because DSLs are supposed to cover their intended domains consistently and at the right abstraction level. To support DSL developers in their designs, some researchers have conducted existing DSLs survey and grouped the design approaches into different design patterns [108, 116]. For each of the design patterns, selection criteria are recommended to DSL developers for determining whether a particular design pattern matches up their DSL. Other researchers have provided guidelines for assisting in language development, making DSL design more a systematic and methodological task and less an intellectual ad-hoc challenge [117, 118].

As a guideline approach, Voelter proposed a framework for describing and characterizing DSL designs [118]. The framework identifies seven design dimensions: expressivity, coverage, semantics, separation of concerns, completeness, language modularization and syntax. These dimensions span the space within which DSLs are designed and provide a vocabulary for describing and comparing the designs of existing DSLs and guiding the designs of new ones. To help DSL developers to make the right decisions, the design alternatives for each dimension are given along with examples from case studies.

To formalize DSL design patterns, Mernik et al. grouped the existing DSL design approaches into two orthogonal dimensions of the relationships between new DSLs and existing languages and the formal nature of the design descriptions [108]. The former includes piggyback (e.g. [119]) to use an existing language partially, specialization (e.g. [120]) to restrict an existing language, extension (e.g. [121]) to extend an existing language, and language invention to develop a new language from scratch (e.g. [122]). The latter specifies the informal approaches (e.g. [119] and [121]) by using natural languages and/or with examples and formal approaches (e.g. [120] and [122]) based on the available semantic definition methods.

To define a pattern language as a building block for a systematic view of DSL development process, Spinellis developed a more comprehensive set of eight DSL design patterns from a detailed survey of exsiting DSLs development approaches [116]. These design patterns include: (1) piggyback, (2) pipeline, (3) lexical processing, (4) language extension, (5) language specialisation, (6) source-to-source transformation, (7) data structure representation, and (8) system front-end. The description of these patterns provides DSL developers with a clear view of the existing DSL realisation strategies and guides them towards the selection of a specific pattern. Therefore, they will have a good understanding of the consequences of a pattern selection, examples of similar use cases and the available implementation alternatives.

DSLs can be broadly grouped into visual, textual and hybrid languages in their representation dimension. There is no fundamental difference between visual and textual languages from the expressivity point of view. A language has programming structures to which meanings are assigned. Viewing and creating these structures can be achieved with a variety of tools, and various representations are interchangeable. However, Moody pointed out a few advantages using visual languages over textual languages [123]. Firstly, visual presentations can convey information more effectively to non-technical people than text. Secondly, people prefer to receive information in visual form and can process it efficiently. Thirdly, diagrams can convey information more concisely and precisely than textual language. And lastly, information represented visually is more like to be remembered than text.

However, most visual languages have a key shortcoming -- diagram complexity for representing complicated implementation. Human mind is limited by working memory capacity to process up to a certain number of elements in a diagram effectively. When the limitation is exceeded, the cognitive overload issue appears rapidly. Therefore, diagram complexity management is one of the most intractable issues in visual languages and a well-known problem is that they do not scale well [124].

To alleviate the diagram complexity issue in business process modeling, Li et al. proposed an Enterprise Modeling Language (EML) to represent complex business systems as tree overlay structure notation supplemented with its support environment (MaramaEML) [125]. EML uses a tree hierarchy to represent organisationally structural services and overlay metaphors to represent process flows, conditions, iteration and exceptions. By

combining these two viewing mechanisms, EML gives users a clear overview of an entire enterprise system with business processes modeled by overlays on the same view. EML incorporates existing business modeling approaches (such as BPMN) to provide additional richer, integrative views for enterprise process modeling.

2.3.3 Implementation

The key objectives of the implementation phase are to create the code generators for DSLs and develop a domain framework for a target environment. After models are created by users, code generators will access and extract information from the models and transform them into outputs in some specific forms. In turn, these outputs will act as inputs for the underlying domain framework to generate executable program for the target environment. Other than default source code, separated code generators are also needed to generate other artefacts, such as development documents, testcases and test data, quality metrics, etc.

A domain framework fills the gap between the generated code and underlying target environment. It fulfils the following objectives: (1) removing duplications from the generated code, (2) providing interface for the generator, (3) integrating with existing code, and (4) hiding the target environment and execution platform. Figure 2.9 illustrates the components of a typical DSM environment and their relationships [107]. Code generators transform models to code, then the code is integrated into the domain framework working in the executable platform.

There is not much difference in development process between a domain framework and a software application. Both have a clear defined requirement at the beginning. From this consideration, our literature review focuses on code generator implementation.

A code generator typically includes three main parts: a parser to read in a model, a code generator proper to transform an abstract syntax representation of the model to the target program representation, and a pretty-printer to format the target program and write it to a text file.

Using the same method as other DSL development phases, Mernik et al. summarised seven implementation patterns on DSL code generators [108]. These patterns include: (1) interpreter, (2) compiler/application generator, (3) pre-processor, (4) embedding, (5)

extensible compiler/interpreter, (6) Commercial Off-The-Shelf (COTS), and (7) hybrid (a combination of the above approaches).

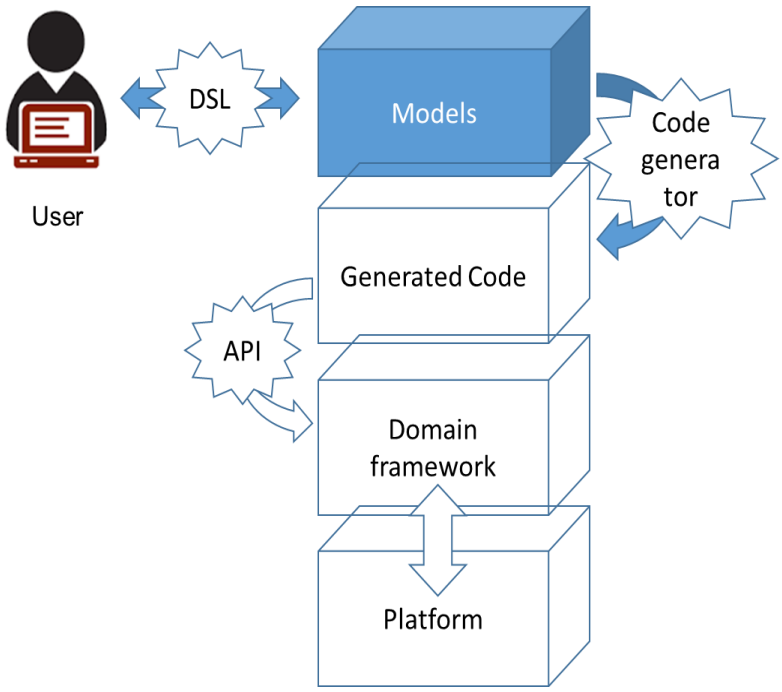


Figure 2.9. A typical domain-specific modeling environment

To help DSL develops to select an appropriate implementation pattern, [108] provides a decision flowchart diagram shown in Figure 2.10. The use of a particular code generator implementation depends on: (1) how the DSL was developed, (2) whether a domain specific notation was strictly obeyed, (3) how large the user community is expected, and (4) whether the DSL was designed using the language exploitation pattern. If the DSL is designed from scratch with no commonality with existing languages, the recommended approach is to implement it by embedding.

To implement code generators, effective languages and tools are crucial. Hemel et al. conducted a case study in code generation by model transformation using Stratego/XT program transformation system [126]. Stratego is a high-level transformation language to be used for model-to-model, model-to-code, and code-to-code transformations [127]. The language provides rewriting rules for basic transformation definition, and programmable strategies for building complex transformations. The use of concrete object syntax creates syntactic correctness of code patterns and enables the subsequent transformation of generated code. To define the concrete syntax of WebDSL, Visser adopted the Syntax Definition Formalism SDF2 [128] to integrate the definition of the lexical and context-

free syntax [112]. As SDF2 is a modular formalism, it is easy to divide a language definition into reusable modules. Another advantage from using SDF2 is to combine definitions for different languages.

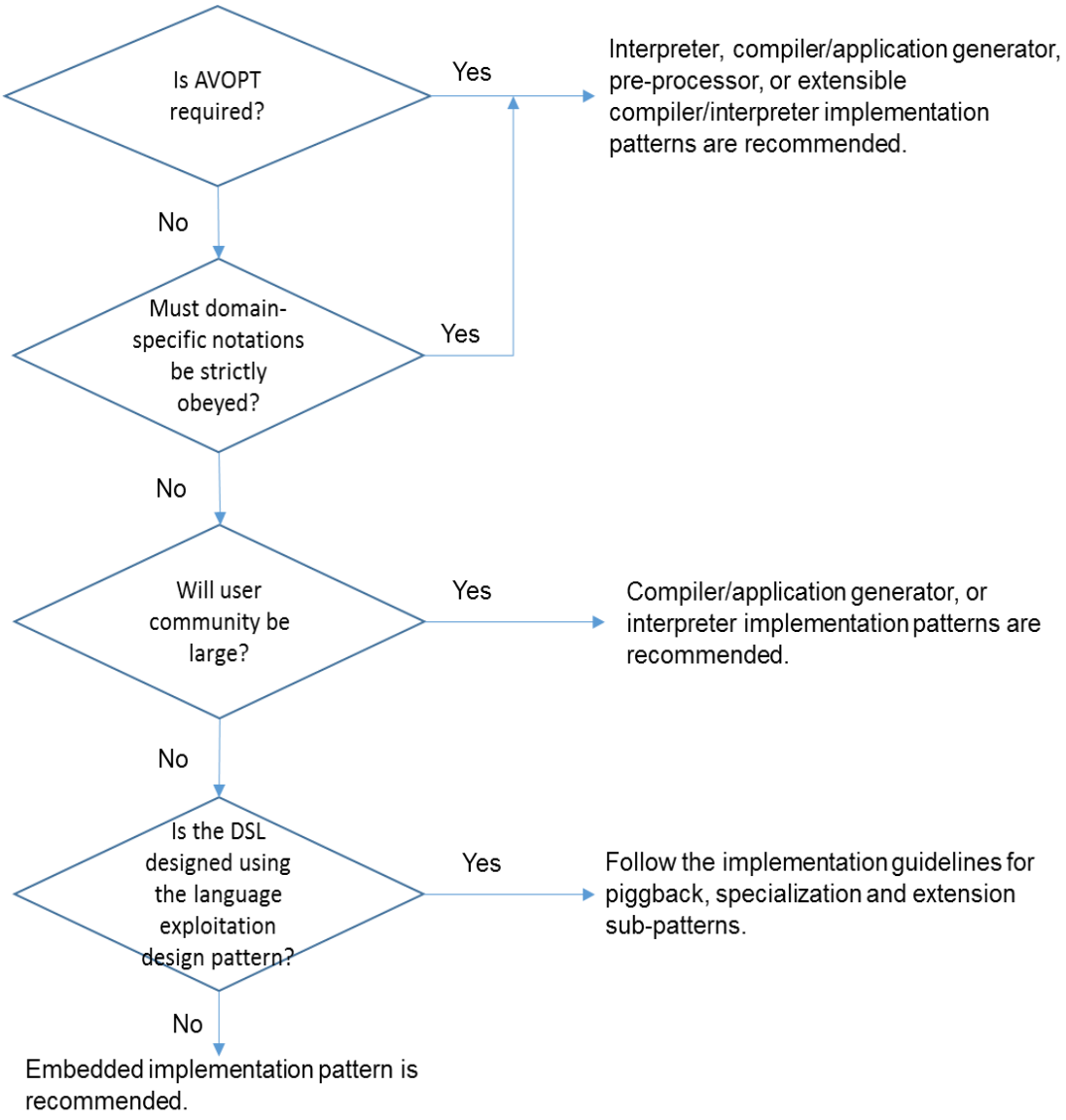


Figure 2.10. Implementation pattern selection guideline

2.4 Domain-Specific Languages and Evaluation Criteria

DSLs sacrifice some flexibility to express any programs in favour of productivity and conciseness of relevant programs in a particular domain. However, the line between DSLs and General-Purpose Languages (GPLs) can be blurred, as a language may have specialized features for a particular domain but be applicable more broadly. Domain specificity is not black-and-white but gradual, a language is more or less domain specific.

To distinguish between these two types of languages subjectively, Voelter characterised DSLs and GPLs by listing 10 language aspects and provided the more likely characteristics to the DSLs and GPLs for each of these aspects in Table 2.2 [118].

Table 2.2. The differences between GPLs and DSLs [118]

Aspects	GPLs	DSLs
Domain size	Large and unbound	Small and limited
Designed by	Guru and committee	A few engineers and domain experts
Language size	Large	Small
Turing-completeness	Almost always	Often not
Execution	Via intermediate GPL	Native
User community	Large, anonymous and widespread	Small, accessible and local
User defined abstractions	Sophisticated	Limited
Lifespan	Years to decades	Months to years
Evolution	Slow, often standardised	Fast-paced
Depreciation/incompatible change	Almost impossible	Feasible

To guide the DSL development process and assess how success the DSL is, Visser proposed 10 DSL engineering evaluation criteria listed in Table 2.3. These criteria include the process to develop DSLs and the desirable features for developed DSLs [112].

Table 2.3. DSL engineering evaluation criteria [112]

Item	Criteria
DSL Development Process	
Productivity	How much will the effort be to develop a new language?
Difficulty	How difficult is it to develop a language? Can it be done by an average programmer or does it require special training? Does it require special infrastructure?
Process	How systematic and predictable is the process?
Maintainable	How well does the process support language evolution? How difficult is it to change the language? Can languages be easily extended with new abstractions?

DSL Desirable Features	
Expressivity	Do the language abstractions support concise expression of applications? What is the effect on the productivity of developing applications using the DSL compared to traditional programming approaches?
Coverage	Are the abstractions of the language adequate for developing applications in the domain? Is it possible to express every application in the domain?
Completeness	Does the language implementation create a complete implementation of the application or is it necessary to write additional code?
Portability	Can the abstractions be implemented on a different platform? Does the language encapsulate implementation knowledge? To what extent do the abstractions leak implementation details of the target platform?
Code Quality	Is the generated code correct and efficient?
Maintainability	How well does the language support evolution? What is the impact of changing a model? What is the impact of changes to the language?

2.5 Summary

The aim of this research project is to develop a new approach to create software testing environments. The approach must be able to address the issues raised by the three key research questions and meet the performance criteria of rich functionality, high development productivity and ease of use, evolved from these research questions.

We first review the current state of the art in the approaches for software testing environment development and propose a new model-driven domain-specific modeling approach. Then, we look into the implementation details of the existing techniques and solutions for endpoint modeling and domain-specific language development. Finally, we list the essential characteristics and differences between DSLs and GPLs and introduced the engineering criteria to evaluate DSLs development.

Starting from the next chapter, we introduce our TeeVML in details and explain how all the issues related to the three research questions are well addressed by our new approach.

CHAPTER 3

Decision Making and Domain Analysis for Functional Layer DSLs

3.1 Introduction

A DSL is "a high-level software implementation language that supports concepts and abstractions that are related to a particular (application) domain" [112]. The high-level means that the DSL abstracts from low-level programming and provides new programming constructs understandable and usable by business users. To use the DSL as an implementation language, we must define its programming constructs, syntax and semantics, and code generators by using a metamodel language. The particular domain is about the scope of computational problems that the DSL attempts to address. This will allow the language to be very expressive and easy to use for the problems that fall in the domain but may be useless for other problems outside the domain.

Our TeeVML approach is developed by going through the typical DSL development life-cycle discussed in the literature review chapter. Such a DSL development life-cycle includes the four phases (excluding the deployment phase) as follows:

- **Decision Making** -- The decision to build a DSM approach is made. The decision making can either be economically based by development productivity gain to offset setup costs, or technically related to address some special problems;
- **Domain Analysis** -- Typical applications in a domain are analysed for identifying the domain concepts and their commonalities and variabilities. From application study, the domain metamodel is drawn for specifying the semantics of all domain concepts and their relationships;
- **DSL Design** – Design and development of the modeling languages for all functional layers and QoS attributes. The semantics of the domain concepts defined in the domain analysis are mapped to the programming constructs of the languages;

- **Implementation** -- Code generators, a domain framework and support tools are generated for automating all the tasks after modeling.

In this chapter, we briefly introduce the DSM development process we used to develop our TeeVML first. We then take endpoint functional layers as examples to describe how we conduct the decision making and domain analysis phases. The last two phases of design and implementation will be discussed in Chapter 4.

3.2 Domain-Specific Modeling Process

Figure 3.1 shows a typical DSM development process overview, which has a DSM approach development environment in the middle and a user modeling environment at the right. To show which MOF level a step is, MOF level is given at the left. A complete DSM process transforms a universe domain metamodel language at the top-level M3 down to a specific domain target at the bottom-level M0.

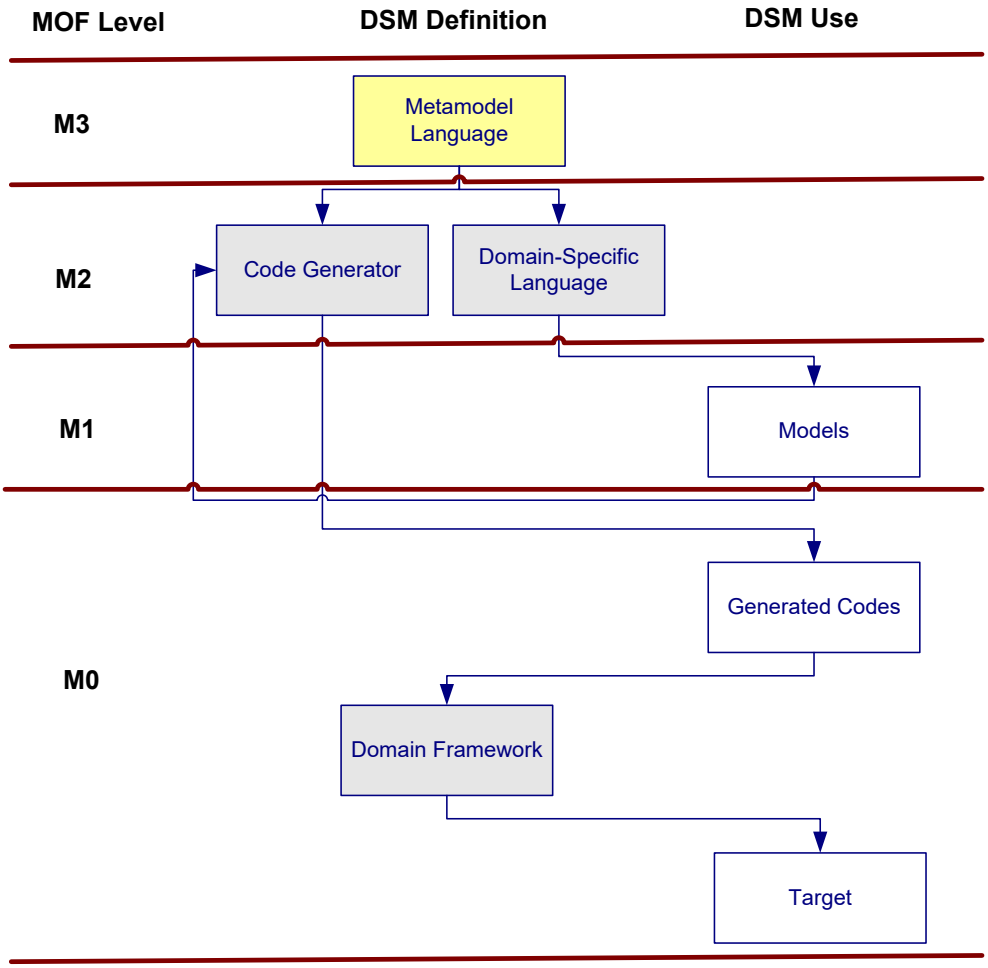


Figure 3.1 A DSM approach development process and application modeling

The three grey boxes are the components in a DSM approach, and they are normally created by experienced software engineers with the domain knowledge. A DSL is created from a metamodel using the metamodel language (the yellow box of Figure 3.1) to specify programming constructs and syntaxes to use these constructs. The models describing an application are instantiated from the DSL. Like a compiler, code generator is used to parse the created models. Domain framework stays between the generated code and target and is used to provide the common features and functions shared by all domain applications. The three white boxes present a model transformation process. Models developed by users are the inputs to the code generator and are converted to codes. The generated codes are integrated to the domain framework to form the executable target.

3.3 Decision Making

From a business point of view, the decision to use a DSM approach is often economic. The investment in DSL development must pay off for itself by more economical software development and maintenance later on. However, we justify the use of a DSM approach from a technical point of view, instead. We want to see whether a DSM approach can meet endpoint development requirement for software testing environment emulation.

Different from software applications, endpoints used in testing environments have some unique characteristics. First, a SUT often interacts with many different types of applications in a testing environment. Therefore, it is desirable that each endpoint development cycle should be short and development approach should have high development productivity. Second, SIT is normally conducted by testing engineers, system analysts or business users. Most of them have rich business knowledge but may lack coding skills. They prefer to model endpoints using problem domain concepts, rather than code them by a textual language. Last, endpoints, as server-side applications to provide testing service, are not necessary to provide accurate results. We may simplify some implementation details, in return for a quick development. Actually, these are the key advantages a DSM approach could provide to its users.

From another angle to investigate the feasibility, we list the potential benefits and possible solutions from a DSM approach to address each of the three key research questions we have formulated for this project:

RQ1 – *Can we emulate a functioning integration testing environment capable of capturing all interface defects of an existing or a non-existing system under test from an abstract service model?*

- Endpoints are modeled based on their interface specifications, rather than generated automatically by interactive trace data or any other means, which are more or less dependent on existing endpoint applications;
- To emulate endpoints from different business domains, we adopt a horizontal DSL development solution, which targets on a specific technical domain -- software interface, not belonging to any specific industrial sector;
- To narrow down applicable scope, software interface is abstracted into different interface layers and attributes. A set of DSLs are developed, each for modeling a specific endpoint interface layer or attribute;
- To capture all interface defects, software interface domain analysis is conducted to analyse all potential interface defects, and the outputs from the domain analysis are converted to DSL development requirements.

RQ2 – *Would our model-based approach improve testing environment development productivity, compared to using third-generation languages (e.g. Java) to implement endpoints?*

- Industrial experiences have consistently shown DSLs to be 5-10 times more productive than traditional software development practices [107]. A DSL raises the level of abstraction and hides the complexities of today's programming languages, in the same way that today's programming languages hide assembler in the early days of computer age;
- The high productivity is applied to endpoint maintenance as well, as any changes to an endpoint are made to its models instead of code;
- With a DSL it is much less likely for users to make errors in the representation of a problem domain than using a GPL, as the DSL imposes some domain-specific constraints and performs model checking that can detect and prevent many human errors early in an application development life-cycle;

- When working with models expressed using a DSL, such models can be validated at the same level of abstraction as the problem space, which means that errors can be detected in an early stage.

RQ3 – *Can we develop a user centric approach, easy to learn and use to specify testing endpoints by domain experts?*

- Working with the concepts used in a specific domain facilitates the understanding of models that represent an application to people who are not experts in software development;
- Having graphical notations that relate directly to a familiar domain not only flattens learning curves but also helps a broad range of subject matter experts to ensure if a software system meets target end user needs;
- Developers use DSLs to model applications using the notations they are familiar with and express their design intent declaratively rather than imperatively.

Therefore, we postulate that a properly designed DSM approach from a software interface domain analysis can meet the requirement of a development tool to emulate testing environments.

3.4 Domain Analysis

Domain analysis is concerned with objects and actions in all systems in a problem domain area. In contrast, the system analysis for an application only considers its specific business requirements and environment. Due to the difference in applicable scope, domain analysts normally use a quite different process, and they must be able to extract, organize, represent and manipulate all domain specific concepts. The outputs of domain analysis include domain-specific terminologies and semantics in abstract forms called metamodels. We divide our domain analysis process into an application study phase and a metamodel development phase.

3.4.1 Applications Study

To define software interface problem domain and identify endpoint functionality, we have set two objectives for the application study. One is to abstract software interface into different interactive aspects, so that a set of DSLs can be developed, each for targeting a

specific aspect. Another one is to find out all potential interface defects. The endpoints that are developed by our approach should be able to capture all these defects and report their causes.

We adopt a bottom-up application study process by choosing typical applications representing different business domains and analyzing their interactive characteristics with other systems. The three applications used for our domain analysis are:

- **PeopleSoft Finance ERP** – the system, introduced in the introduction chapter, represents those workflow applications with multiple interactive steps for completing a full cycle business process;
- **Core Banking System** – a system interacts with a new mobile banking application. This application consists of some typical business operations, focusing on logic processing and persistent data manipulation;
- **Lightweight Directory Access Protocol (LDAP) Server** – LDAP server is a utility application to provide directory services for users and systems in enterprise environments.

Below, we describe the interactive aspects of the core banking system and LDAP server.

The core banking system [129] is currently used by a bank to support its in-house daily banking operations and provides online banking services for its clients. For the sake of its clients' convenience and satisfaction, the bank is planning to introduce a new mobile banking application for allowing its customers to manage their bank accounts through mobile devices. The application operations include login and logout, account balance and transaction query, money transfer between accounts in a user account, bill payment, and money transfer to other user accounts. As the mobile application acts as the front-end and all clients' data are kept in the core banking system, all user requests must be handled by the core banking system. From functional point of view, the core banking system is treated as the endpoint to validate the correctness of operation requests from the mobile banking application as the SUT in our testing environment.

The interactions between the endpoint and SUT are described as follows:

1) after being successfully authenticated by a user's account number and corresponding password, a logon request starts an interactive session for banking operations;
2) in an interactive session, a query operation for account balance or transaction history can be invoked;
3) to transfer money within one user account, the user is required to re-enter password for double authentication;
4) for bill payment or money transfer to another user account, the user is required to enter a One-Time Password (OTP), which is generated and sent to the user through a text message by the endpoint;
5) all operations are synchronous, and the endpoint will reject any operation request when it is in processing a request received earlier. A such business scenario is a client sending a deposit operation request followed by a money transfer. If the endpoint is still handling the former request and the later one should be rejected. Otherwise, the bank account may not have enough balance for the money transfer;
6) some transaction operations, such as money transfer and bill payment, are unsafe -- i.e. not idempotent operations that will produce the same result if executed once or multiple times. If a client communicates with an endpoint over an unreliable network, requests may take unexpected long time to reach the endpoint. However, if the client does not receive an acknowledgment for a request within a time-out period, the request is considered lost and the client will re-send the same request. By doing so, the endpoint will receive the request twice and function wrongly, consequently;
7) a logout request terminates the session automatically, if no operation request is received within a certain period of time.

Figure 3.2 illustrates the core banking system endpoint interactive behaviors using a state transition diagram. The round rectangular boxes are its states and the arrow lines represent the state changes triggered by operations. When users login the system successfully, the endpoint state will transit from Idle state to Active state, and they can execute all query operations. If the users want to do a transaction within their accounts, they must be re-authenticated with password and the endpoint then moves back to the Active state after

the transaction. Users can also do an external transaction, and an OTP is required for their authentication. Once users have completed their banking operations, a logout (or timeout event) command will move the endpoint to Idle state.

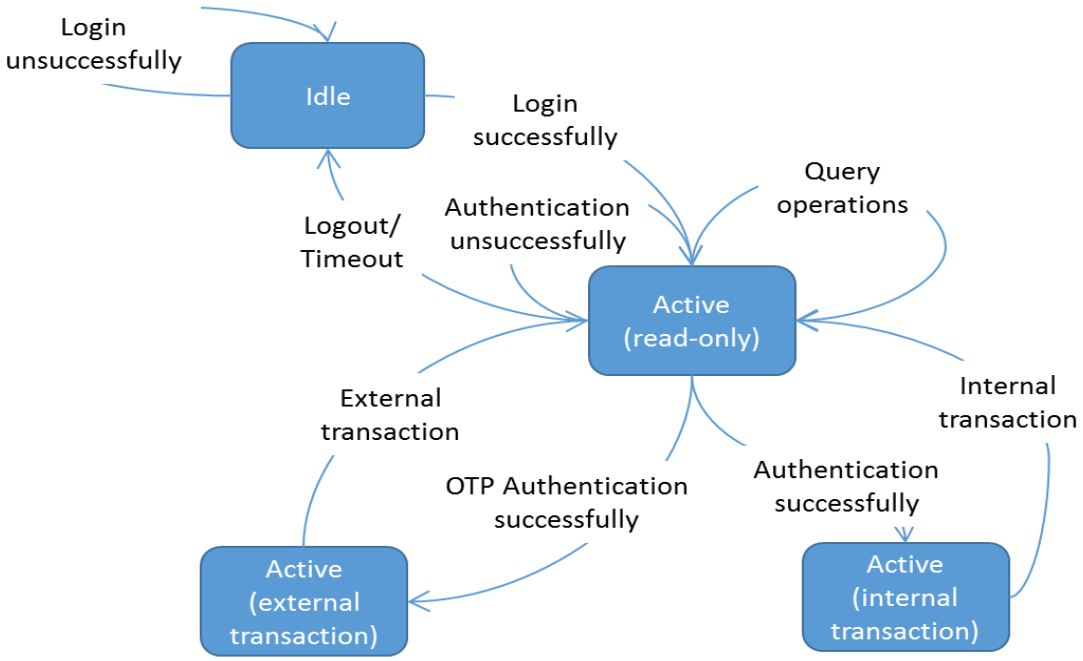


Figure 3.2. The core banking system endpoint state transition diagram

LDAP is an application protocol for accessing and maintaining distributed directory information services over an IP network. Normally, a new application is integrated with an enterprise LDAP server first for directory services, before it can be put in production. LDAPv3 defines eleven server operations from three different categories: authentication, searching and modification [130]. We select eight operations for this domain analysis: ldap_bind and ldap_unbind from the authentication; ldap_search and ldap_next_entry from the searching; and ldap_add, ldap_delete, ldap_compare and ldap_modify from the modification. The LDAP server is treated as the endpoint to be emulated and any applications that access the LDAP server are considered as SUTs.

The LDAP server interacting with its client for these operations is described as follows:

1) ldap_bind operation starts a LDAP service session, either with a registered account or anonymously;

2) if an application binds to the LDAP server with a registered account, it can access all the operations; otherwise, only the searching category operations are accessible;

3) all the operations can be accessed from Home state, except for ldap_next_entry, which is a slave operation of ldap_search;

4) only ldap_modify and ldap_compare can access each other directly, and the others must move back to the home state first;

5) all states have a timeout event, which will transit the LDAP server from the current "from" state to the "to" state automatically;

6) ldap_unbind terminates a LDAP service session.

Similar to the core banking system, we also use a state transition diagram to illustrate the LDAP server endpoint interactive behaviors in Figure 3.3. Whenever the endpoint is at any operation state, it can repeat the same operation until an escape command is issued or a timeout event occurs. Any applications can bind to the endpoint either with a registered account or anonymously. The left-hand side shows the anonymous binding that only one operation ldap_search is allowed at Home state and it has a slave operation ldap_next_entry. To enter the right-hand side states for making LDAP entry changes, user's credential must be authenticated. ldap_compare and ldap_modify operations can be accessed from each other directly, and others must return to Home state first.

3.4.2 Software Interface Description Framework

From the applications study, we propose a new software interface description framework, which is a modification of Han's comprehensive interface definition framework for software components [131]. There are three reasons why we need to have such a framework. First, we need to abstract software interface into different interactive aspects, so corresponding DSLs can be developed with a clearly defined problem domain boundary. Second, we can adopt a modular development architecture to model endpoints from functional layers and QoS attributes. We may also be able to model a few versions of an endpoint type for different SUTs. Third, some of these interface modules may be shared among endpoints, if they have the exact same functionality.

Our framework abstracts software application interface into three horizontal layers and a number of vertical attributes. These horizontal layers are directly related to operation request processing, and they are:

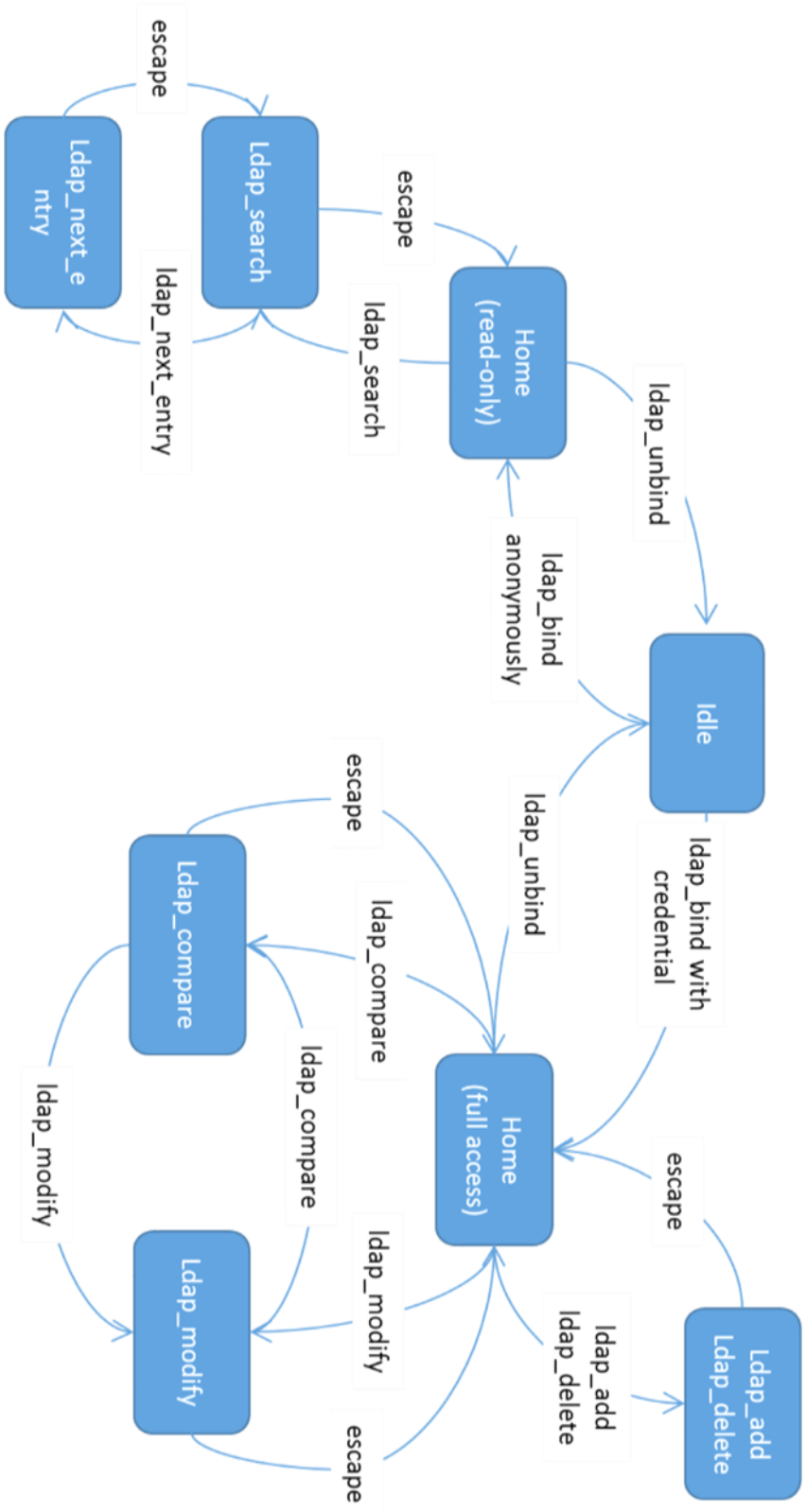


Figure 3.3. The LDAP server state transition diagram

- **Signature** -- following RPC communication style specification, this layer defines a pair of request and response of all endpoint operations, their parameters and properties;
- **Protocol** -- this layer defines the validity of the temporal sequence of operation requests, which is subject to either endpoint state (static protocol behavior) alone or both endpoint state and runtime constraint conditions (dynamic protocol behavior);
- **Behavior** -- this layer abstractly describes endpoint internal operation request process and response generation, and the returned values in response messages are used to capture dynamic protocol defects.

Particularly, the protocol layer is only applied to those applications using stateful communication style. This is because endpoint protocol layer validates the next coming operation using its current state as an input parameter. If an endpoint application is stateless, its state information is retained by neither itself nor its client. The correctness of its client's requests depends on their signatures only. Therefore, both protocol and behavior layers can be skipped. To be applicable to a wide variety of applications in the real world, this research focuses on stateful applications using RPC for service invocation and excludes stateless applications, such as REST and applications using messaging communication protocols.

A SUT operation request is processed horizontally by an endpoint step-by-step from signature, protocol, down to behavior layer. Whenever an error occurs at any layer, the request process will be terminated. The signature and protocol layers act as message pre-processors for checking the correctness of an operation request syntax and temporal sequence. Then, it is handed over to the behavior layer for generating a suitable operation response message.

On the other hand, vertical attributes are related to the QoS aspects of application interactions, and they specify the criteria that are used to assess the operation of an endpoint system. Examples of vertical attributes include: compliance with endpoint security policy; robustness for handling various endpoint faulty conditions, e.g. timeout, no response, wrong message in sequence, wrong message format; performance requirements to support the maximum number of endpoint instances, throughput; etc.

Unlike horizontal layers, different endpoints will have different vertical attributes, and operation requests are validated by endpoints for the correctness of these QoS attributes in arbitrary order[5].

We have introduced our approach architecture in the introduction chapter (refer to Figure 1.4). Our approach has a modeling environment for domain experts to model endpoints at the upper part, and a runtime environment to provide testing services to SUTs at the lower part. The modeling environment contains the DSLs of the functional layers of signature, protocol and behavior and of the QoS attributes of robustness, security, performance, etc. Signature layer is modeled first to define operations and their parameters, and they are used as the inputs for modeling protocol and behavior layers. On the other hand, QoS attributes are independent from each other, and they can be modeled in arbitrary sequence.

Code generators in the middle automatically transform the endpoint models to executable codes for building up testing environment. Testing services to SUTs are provided by packaging the testing environment into Apache Axis 2 Web services interface [20]. The QoS attributes of a SUT request are validated first without a specific order. Then the request is handed over to functional layers for further checking only if the validation of all QoS attributes is passed. Unlike the QoS attributes, the functional layers process the request in sequence, starting from the signature, protocol, and to behavior lastly.

In the followings of this chapter, we describe our domain analysis for the three functional layers only. The development for the QoS attributes will be discussed in Chapter 6.

3.4.3 Service Request Defects

To develop DSLs for modeling endpoint functional layers, we must know all the software interface defect types first. In the introduction chapter, we analysed the possible SUT request defects and grouped them into static and dynamic categories based on their occurrence rate. The static defects will always cause software interactions to fail. In contrast, the dynamic defects are subject to the endpoint current status and they will be rejected by the endpoint under a certain kind of runtime conditions only.

[5] This is the reason why we call QoS aspects as attributes, rather than layers for functional aspects.

In general, signature defects are static and mainly include wrong request names or parameter type mismatching. Protocol defects are dynamic and whether they will cause interactive failures depends on what service request is received by the endpoint and any constraint conditions exist. Due to this nature, software products normally have an interface specification for signature definition only, and we may not be able to capture application protocol defects by going through code reviews. Instead, a SIT must be conduct in a realistic testing environment. This is the reason why most protocol defects can be detected by conducting SIT only. Table 3.1 lists and describes the possible functional layers' defect types that a SUT may have.

Table 3.1. Service request defect types

Type	Description
Signature	
Sig1	An operation request is not an operation provided by endpoint.
Sig2	The parameters in an operation request are not matched with the parameters of the corresponding operation provided by endpoint, in terms of parameters' name, data type and order in the operation request.
Sig3	One or more operation request mandatory parameter(s) is (are) missing.
Sig4	One or more parameters in an operation request is (are) beyond the defined value range of the corresponding endpoint operation.
Protocol	
Pro1	An operation request is invalid for the current endpoint state.
Pro2	An operation request is invalid for the current endpoint state, as one or more parameter(s) violate(s) the defined constraint condition(s).
Pro3	An operation request is invalid for the current endpoint state, as one or more returned value(s) from a previous operation request violate(s) the defined constraint condition(s).
Pro4	An operation request is invalid, due to endpoint state transition driven by some internal event, such as time out.
Pro5	An operation request is invalid, as the endpoint is in processing a synchronous operation request sent earlier.
Pro6	An operation request is invalid, as one such request for an unsafe operation has been received by endpoint.

Table 3.1 does not include any behavior defects. This is because a SUT's obligation is to send correct service requests to its endpoint and the way these requests are to be processed

is defined in the endpoint's internal implementation. The reason why we still model endpoint behavior is that the validity of alternative next operation requests may depend on what values are returned in a response message it has received for a previous operation request (refer to the Pro3 defect type of Table 3.1).

To address our research question RQ1, endpoints that are developed by our approach must be able to capture all of the SUT request defects listed in Table 3.1 and report their causes.

3.4.4 Functional Layer Metamodels

A metamodel is a model that precisely defines the constructs and rules needed for creating a corresponding semantic model. Metamodeling attempts to describe the world of interest for a particular purpose. A model is an abstraction of phenomena in the real world, and a metamodel is yet a higher abstraction, highlighting properties of the model itself. A typical metamodel development process includes these activities: (1) applicable domain scope definition; (2) domain terminology specification, including vocabulary and ontology; (3) domain concept descriptions; and (4) feature model development, describing domain concepts and their interdependencies.

There are two main inputs to our DSL metamodels from the previous application study phase. One is the software interface description framework, which helps us to break down software interface into different problem domains and define their domain boundaries. Another one is the software interface defects listed in Table 3.1. Our DSLs must be able to develop endpoints that are capable of capturing all these defects.

3.4.4.1 Signature Modeling

Endpoint signature layer models endpoint provided operations and their parameters based on RPC communication style, where an operation request from an operation consumer is passed across a network to an operation provider and the returned response is sent back to the consumer [23]. Each parameter has some static properties, such as name, data type, order and mandatory. Some parameter types, such as integer, float or date, may also have upper and lower limits. Our signature metamodel must be able to capture all these concepts and specify their relationships.

Web Service Description Language (WSDL) is a standardised XML-based specification schema to describe the details of the public interface exposed by a Web operation,

including what an operation does, where it resides, and how to invoke it [22]. It consists of an operation interface definition to define the specific type of interface provided and an operation implementation definition to describe how a particular operation interface is implemented.

To define operation parameters, WSDL employs W3C XML Schema type system component [63] to declare elements and define types in a formal manner. The benefits from using WSDL specification as our Signature DSVL include:

- WSDL defines the necessary entities for users to construct a service provider, and provides well-documented interfaces for both internal logic implementation and external operation invocation;
- A testing runtime environment can be generated automatically by transforming a signature WSDL file to Axis2 SOAP engine using Axis2 wsdl2java utility [20]. By doing so, Axis2 provides the messaging protocol layer of a Web service protocol stack. So, users can concentrate on endpoint modeling at application layer;
- All static signature defects can be detected by the transformed Axis2 SOAP engine. It can save our development effort on validating these signature defects.

Figure 3.4 illustrates our endpoint signature metamodel that our Signature DSVL is based on. The metamodel adopts a three-level architecture design. The top-level DSVL (see Figure 3.4a) uses WSDL 1.1 specification as its metamodel. It consists of a root Definition entity and other four entity types: Service, Port, Binding and Porttype. These entities are briefly described below:

- Definition – a collection of basic definitions to define the provided Web service;
- Service -- a collection of related endpoints;
- Port -- a single endpoint defined as the combination of a binding and network address;
- Binding -- a concrete protocol and data format specification for a particular port type;
- Porttype -- an abstract set of operations supported by one or more endpoints.

The instances of these entities are related with each other by use of two relationships: Composition and Association. A Composition relationship (a filled diamond shape line) connects contained entities to a containing entity; and an Association relationship (a straight line) associates two interrelated entities by one or more attribute(s) of the two entities. As we restrict one service definition per signature model, a Composition relationship is used to connect a Definition instance to a Service instance; and two Composition relationships connect a Port instance to the Service instance and a Porttype instance to the Definition instance, respectively. Two Association relationships are used to associate a Binding instance with a Port and Porttype instances.

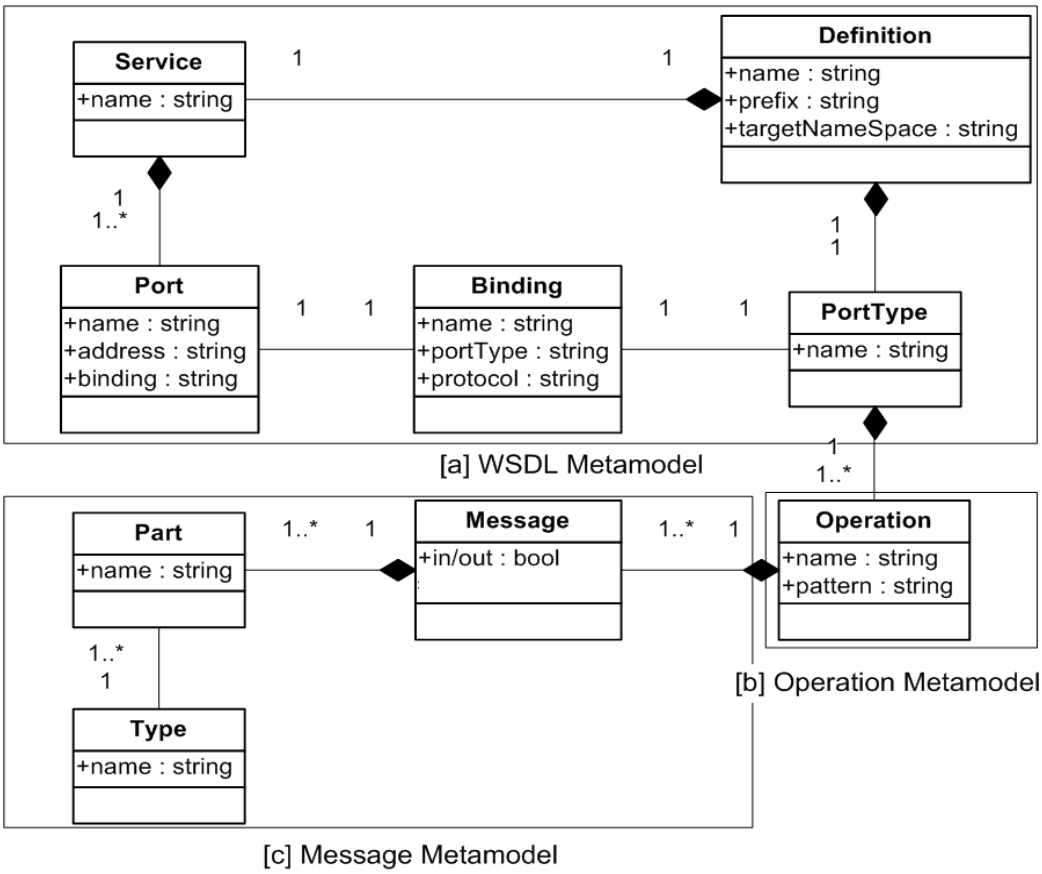


Figure 3.4. Endpoint signature metamodel

The middle-level Operation DSVL (see Figure 3.4b) is for modeling endpoint operations. One or more Operation instances are connected to a Porttype instance by a Composition relationship. Operation DSVL models an operation by providing the operation name and pattern properties. The pattern determines whether it contains a request message only, a response message only or both request and response messages.

The bottom-level Message DSVL (see Figure 3.4c) is for modeling messages and their parameters. A Message instance is a decomposition of an Operation instance; and the message in/out property determines that it is a request or a response. Part entity is used to define message parameters, and their data types are specified by Type entity based on W3C XML Schema 1.1.

By using the multi-level modeling approach, lower level models can be reused by higher level models. Message element's reusability is particularly important for signature modeling, as an endpoint may have a large number of message parameters to define and many of them have the exact same properties but are in different operations. This is the reason why we add an additional operation level sub-DSVL to our Signature DSVL for separating operation specific request and response messages from reusable message elements.

There are some open-source or commercial WSDL tools available, such as Eclipse WTP Plugin [132] and XMLSpy [133]. The motivation for developing our own WSDL tool is to increase the consistency among different parts of TeeVML. Behavior model imports operations and their messages and parameters from corresponding signature model and Message DSVL is reused to define data store model.

The signature defects Sig1 to Sig3 in Table 3.1 can be detected by Axis2 SOAP engine itself. To specify the upper and lower limits of a number or date element (refer to Sig4 defect type in Table 3.1), we add two properties to element type to detect any invalid request parameters beyond defined value limits at runtime.

3.4.4.2 Protocol Modeling

Finite State Machine (FSM) has been widely used to model communication protocols [12]. However, such an FSM can only validate endpoint static protocol behavior, and an endpoint state transition depends on its received operation request only. As given in our motivating case study in the introduction chapter, there are some dynamic protocol behaviors: (1) some constraint conditions may prevent an endpoint from changing its state, even after the endpoint has received an operation request; (2) some internal events may change an endpoint state automatically, such as a timeout event; and (3) sometimes an endpoint may not be able to process all or a certain type of operation requests, such as the endpoint in processing a synchronous operation.

To deal with the incomplete protocol specification problems and capture runtime constraints, we design an Extended Finite State Machine (EFSM) to enrich our protocol modeling capability with dynamic protocol aspects (refer to Figure 3.5). Its core part is a traditional FSM with three entities of State, Transition and Operation. State entity represents endpoint state changes from its initial state, different active states to terminative state. Transition entity drives State changes, when the endpoint receives an Operation instance. On the other hand, the endpoint validates a coming Operation instance based on its current State.

To handle dynamic protocol aspects, we add one entity type and two entity properties to the FSM (the items are marked yellow in Figure 3.5). The entity type is InternalEvent, which is used to define state transitions triggered by time event. One of the entity properties is StateTransitionConstraint of Transition entity, and it is for specifying either static or dynamic constraints on state transition function. Another one is StateTimeProperty of State entity, which allows users to simulate synchronous and unsafe operations. As endpoint protocol modeling is relatively simpler than other two functional layers, we use a flat view presentation structure.

All protocol defects listed in Table 3.1 can be detected by an endpoint, developed by a modeling tool based on our EFSM model: (1) Pro1 – the operation-driven state transition FSM; (2) Pro2 and Pro3 – StateTransitionConstraint property; (3) Pro4 – InternalEvent entity; and (4) Pro5 and Pro6 – StateTimeProperty.

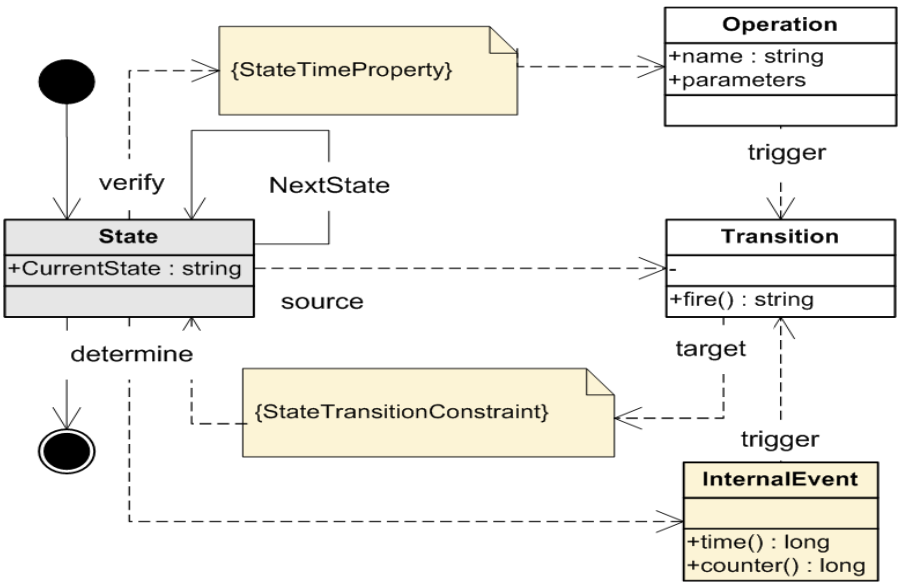


Figure 3.5. Endpoint protocol metamodel

3.4.4.3 Behavior Modeling

Our endpoint Behavior DSVL is designed based on DataFlow Diagram (DFD) programming paradigm [87]. DFD is the core to most visual programming languages and claims to provide end-user modeling capability of application behaviors in some way. We choose this metaphor as it is capable of modeling complex specification of behavior models and understandable by a wide range of software stakeholders as well. Furthermore, it allows abstract endpoint behaviors to be modeled quickly and effectively across a variety of application domains.

DFD programming execution model is represented by a directed graph; nodes of the graph are data processing units, and directed arcs between nodes or visual constructs in a node represent data dependency and flows. Each node is an executable block that has data input, performs data transformations and then forwards computational results to the next node in the computation chain. A node starts to process and convert data, when it has the minimum required input parameters available and all its dependent predecessors have been executed. The node may execute successfully or exceptionally, and this will determine the alternative process flows after the node. Therefore, nodes have two exit ports for connecting different nodes to be executed next, depending on their execution status.

To be able to model any assigned tasks, our Behavior DSVL provides a node with some key computational entities as follows:

- Input/Output connectors to hold input parameters and output results and mark the starting and ending execution points of a node;
- Arc to link a “from” entity instance from its “out” port to the “in” port of a “to” entity instance;
- Variable/Variable array to hold intermediate result(s);
- Evaluator to perform an arithmetic operation on input values and assign result to a variable;
- Conditional operator to test two input parameters for determining alternative process flows;
- Iterator to execute a block of entity instances for a number of times.

Conceptually, data flows along arcs as tokens and behaves like unbounded first-in, first-out (FIFO) queues. Arcs that flow toward a node are input arcs to that node, while those that flow away are output arcs from that node. When a program begins, a special activation node places data onto certain input arcs, triggering the rest of the program to run. The program ends, when the last node, which does not have any other nodes connected to its exit ports through arcs, has finished its execution.

To handle complicated business logic, we design our Behavior DSVL using a hierarchical tree structure, an approach we have successfully used on earlier business process modeling problems [88]. The benefits of using the hierarchical structure are two-fold: first, we can reuse some of the nodes, if they perform exactly the same task but are located in different components. Second, it can help us managing diagrammatic complexity problem. At the bottom level, each node consists of some primitive programming constructs for a specific data processing operation. At the top level of node tree structure, discrete service nodes are used to represent the operations provided by an endpoint. To prevent the data inconsistencies between behavior model and signature model, each of the service nodes imports its request and response parameters from the same endpoint signature model.

The behavior layer often needs to access and manipulate persistent records for processing business logics. As discussed in the follow chapter, we use MySQL relational database management system [134] to store these records. Signature Message DSVL is reused to create database tables. To handle Create, Read, Update and Delete (CRUD) persistent storage operations, we create a JDBC operator to access a JDBC class domain framework.

3.5 Summary

In this chapter, we use endpoint three functional layers of signature, protocol and behavior as examples to illustrate how the early phases of endpoint DSL development are conducted. Unlike traditional cost-driven decision-making process, we have decided to use a DSM approach from a technical point of view. We analyse the characteristics of endpoints in testing environments and postulate the potential solutions from such an approach to answer our three research questions.

Our domain analysis is split into two steps of domain application study and metamodel development. From the application study, we develop a software interface description framework and identify interface defect types to be detected by endpoints. From the

application study outputs, we create the DSL metamodels for endpoint three functional layers. Signature layer uses WSDL specification as its metamodel to define endpoint operations and their parameters. Protocol metamodel is based an Extended Finite State Machine to capture both static and dynamic endpoint protocol aspects. Behavior layer is modeled using DataFlow Diagram programming with nodes as data processing units and arcs to specify the inter-dependences among these nodes.

In the next chapter, we further describe the last two phases of our functional DSL development -- the design of DSL visual constructs and implementation of code generators and a domain framework.

CHAPTER 4

Design and Implementation for Functional Layer DSLs

4.1 Introduction

We described the first two phases of our TeeVML development for endpoint functional layers in the last chapter. In this chapter, we continue the discussion to cover the last two phases of the design of domain-specific languages and the implementation of code generators and a domain framework. This chapter structure is organized as follows: first, DSL design principles are given to guide our TeeVML development in this chapter. Second, our development process of DSVLs is introduced. Third, the design details of DSVLs' visual constructs are discussed. And fourth, the implementation of code generators and a domain framework are described. Before the end of the chapter, we describe how to select a metamodeling language and introduce our metamodel tool MetaEdit+ 5.1.

4.1.1 DSL Development Principles

DSL development is hard, its developers need to have the expertise on both application domain and language development [135]. DSL development techniques are more diverse than GPLs, requiring careful consideration of the various factors involved. To help IT professionals to develop their new DSLs from scratch, Kelly et al. proposed a set of high-level language definition guidelines [107]:

- **Follow Established Naming Conventions** -- To define domain concepts, it is preferred to use exactly the same names and naming conventions for the language concepts already in use;
- **Keep the Language Simple and Minimal** -- It suggests to stick with the identified needs and support them first. The language should be extended later if needed;

- **Try to Minimize Modeling Work** – Users model application concepts and fill their properties, which are varied in the domain. All the common aspects and similarities should be produced by code generators or provided by a domain framework;
- **Define Modeling Concept Precisely** – To provide unambiguous definitions for all domain concepts semantically;
- **Consider Language Extension Possibilities** -- If the domain is new or it is unclear whether the defined language provides the needed modeling capabilities, the language needs to add special extension concepts;
- **The Language Does Not Need to Include Every Domain Concept** -- Some relevant domain concepts can be "composed" by combining existing concepts.

To develop ease of use and high development productivity DSLs for TEE, we followed these guidelines during our development process.

4.1.2 Domain Specific Rules

Along with modeling concepts, a DSL should also enforce various domain specific rules, constraints and consistencies. These domain specific rules may include the followings (not exclusively) [107]:

- **Naming Conventions** – An example is that a value must start with a capital letter or must not include certain characters;
- **Uniqueness** -- There cannot be another entity instance with the same property value;
- **Mandatory** – An entity property must have a value;
- **Default Values** – The best choice of values under most circumstances;
- **Occurrence** – An entity can only have a certain number of instances in a model;
- **Binding Rules** -- What kinds of entities can be linked together?
- **Connectivity Rules** -- How many times may an object have a certain kind of connections?

- **Reuse Rules** -- A user can choose a certain value or refer to another model element, rather than create a new one;
- **N-ary Relationship Rules** -- How many objects can a single relationship connect to?
- **Integrating Models** – Can a value be shared with another entity, possibly in another model?
- **Model Structuring Rules** – Can a model be decomposed to lower level models or be referred to libraries?

Having specific syntaxes and business rules in a language makes it domain-specific and brings many benefits to users. The key benefits include: (1) human mistakes and errors can be prevented at early stages and illegal or unwanted modeling cannot be made; (2) developers are guided toward preferable design patterns; (3) model completeness can be checked by reporting any missing parts; (4) modeling work can be reduced by applying conventions and default values; and (5) modeling consistencies among different parts can be kept. Domain rules are best defined after having decided on the main modeling concepts. We describe what domain rules are implemented when introducing our DSVLs.

4.2 Domain-Specific Visual Languages

A Visual Programming Language (VPL) lets users create programs by manipulating programming constructs graphically rather than by specifying them textually. It allows programming with visual expressions by spatially arranging graphic symbols with complementing texts. Although there is no fundamental difference in expressivity, visual languages are generally easier to learn and use than textual languages [136]. This is because visual languages use models to represent developers' design intents. Models are considered easier to learn, comprehend, and navigate than textual programs. Furthermore, a VPL normally has less programming constructs than a textual language. This will let us predefine some syntactic constraints on the use of these programming constructs easier than textual languages. Considering all these advantages, we have made our decision to use Domain-Specific Visual Languages (DSVLs) for our DSM approach to TEE.

A visual language (also called visual notation) consists of a set of visual symbols (also called visual constructs), visual grammars to define a set of compositional rules and visual

semantics to describe meaning of each symbol. Visual symbols and visual grammars together form visual syntax. Visual symbols are used to symbolize semantic concepts, typically defined by a metamodel. The meanings of visual symbols are defined by mapping them to the concepts they represent. A valid expression in a visual language is called a visual sentence or diagram. Diagrams are composed of visual symbols, arranged according to the rules of visual syntax [123].

4.2.1 Visual Symbol Design

Visual symbols have a profound effect on the usability and effectiveness of a visual language. Visual symbols play a critical role in communicating with business users and customers as they are believed to convey information more effectively to non-technical people than text. Research in diagrammatic reasoning shows that the form of representations has equal (if not greater) influence on communication effectiveness as their contents [137]. Particularly, a DSVL is usually "small" in terms of its user community. It will not be a surprise to see that most users are new to the language. To flatten their learning curve, a DSVL must use visual symbols that are easy to be comprehended.

Visual symbols are a kind of human thought representation for facilitating communications and problem solving among individuals. To be most effective in doing this, they need to be optimized for processing by human mind. For this reason to evaluate the "goodness" of a visual symbol, Larkin et al. defined the term cognitive effectiveness as "the speed, ease, and accuracy with which a representation can be processed by the human mind" [138]. Cognitive effectiveness determines the ability of visual symbols to both communicate with a wide range of software stakeholders and support design and problem solving by software engineers.

Cognitive Dimensions (CDs) of Notations framework is a popular, psychologically based heuristic technique to quickly evaluate a visual language in terms of the cognitive effectiveness of its visual constructs [139]. CDs consists of a small vocabulary of terms (or dimensions) designed to capture the cognitively relevant aspects of visual symbols. To maximize overall cognitive effectiveness, designers need to trade off these dimensions against each other. CDs does not intend to provide a rigorous guidance for designing visual symbols, but instead gives designers a rough idea of the human factor issues inherent in visual languages. Therefore, CDs allows designers to get a broad-brush feel

for the characteristics of a visual language before or instead of running an expensive usability study. Although, CDs framework has played a valuable role in advancing visual symbol designs beyond the level of intuition. But, Moody pointed out that it does not provide a scientific basis for evaluating and designing visual symbols, mainly because of its poor dimension definitions [140].

To establish a scientific foundation for visual symbol designs, Moody proposed Physics of Notations (PoN) and defined a set of principles to evaluate, compare, and construct visual symbols [123]. These principles were developed using a synthesis approach based on theory and empirical evidence about the cognitive effectiveness of visual symbols. Some of these principles are related to a visual language as a whole, such as Complexity Management, Cognitive Integration and Graphic Economy. While others focus on individual visual symbol's properties, such as Semiotic Clarity, Visual Expressiveness and Perceptual Discriminability.

To maximize the cognitive effectiveness of our DSLs, we apply eight out of the nine PoN principles to the designs of our visual symbols. The last principle Cognitive Fit is irrelevant to this research, as we do not expect to have different visual dialects for different tasks or users. Table 4.1 lists these PoN principles, definitions, descriptions, and our corresponding rules guiding our designs for visual constructs. Among these eight principles, we would put emphasis on some of them subject to DSVL's characteristics. If multiple entities are to be used, Perceptual Discriminability principle will be our primary design consideration. This principle is assessed by the visual distance between symbols, measured by the number of visual variables on which they differ and the size of these differences. In contrast, there is no meaning to consider visual distance, if a DSVL contains only one entity. Instead, we would focus on Semantic Transparency principle. Our TeeVML's usability was actually assessed by a user study described in Chapter 7.

In addition to visual symbols, there are also some other factors to be considered when designing DSVLs, such as reusability for this research. To maximize the reusability, we should make models simple enough to be reused directly or easily assembled with others as a reusable component. This is the main reason why we have designed some single entity sub-DSVLs. But this may contradict Cognitive Integration principle with too many lower level diagrams, and they cannot be seen at higher-level diagrams.

Table 4.1. PoN principles [123] and our visual symbol design rules

PoN Principle	Definition	Description	Visual Symbol Design Rule
Semiotic Clarity	There should be 1:1 correspondence between semantic constructs and visual symbols.	A diagram should not have symbol redundancy, overload, excess and deficit.	All visual symbols should have 1:1 correspondence to their referred concepts.
Perceptual Discriminability	The ease and accuracy with which visual symbols can be differentiated from each other.	Discriminability should be primarily determined by the visual distance between symbols. This is measured by the number of visual variables on which they differ and the size of these differences.	All symbols should use different shapes as their main visual variable, plus redundant coding such as colour and/or textural annotation.
Semantic Transparency	Visual representations whose appearance suggests their meaning.	The extent to which the meaning of a symbol should be inferred from its appearance.	We should use icons to represent visual symbols and minimise the use of abstract geometrical shapes.
Complexity Management	Complexity management refers to the ability of a visual notation to represent information without overloading human mind.	A diagram should have as few visual elements as possible to reduce its diagrammatic complexity.	We should use hierarchical view representation and allow user to hide visual construct details for complex diagrams.
Cognitive Integration	Systems using multiple diagrams place additional cognitive demands on users to mentally integrate information from different diagrams and keep track of where they are.	Information from separate diagrams should be assembled into a coherent mental representation of a system; and it should be as simple as possible to navigate between diagrams.	All the relationships between diagrams should be in hierarchical tree structure, and child diagrams should be opened only from their parent diagram.
Visual Expressiveness	It is defined as the number of visual variables used in a notation to measure	A range of visual variables should be used, resulting in a perceptually enriched representation that exploits multiple visual	We should use various visual variables, such as shape, colour, orientation, texture, etc. when designing visual symbols.

	visual variation across entire visual vocabulary.	communication channels and maximizes computational offloading.	
Dual Coding	Use text to complement graphics.	Textual encoding should be used, as it is most effective when used in a supporting role -- to supplement rather than to substitute for graphics.	All visual symbols should have a textual annotation.
Graphic Economy	The number of visual symbols in a notation - the size of its visual vocabulary.	The number of different visual symbols should be cognitively manageable.	We should use as few visual symbols as possible in a DSVL.

4.3 Design of TeeVML Domain-Specific Visual Languages

We present our DSVLs' metamodels for endpoint three functional layers in the last chapter. The domain concepts in the metamodels must be mapped to the modeling concepts in DSLs. The main concepts are considered as objects existing more or less independently from others. We then add more details, and our focus generally moves from objects to other kinds of language concepts such as their properties, connections in terms of relationships, roles, or ports the objects may have in different connections or sub-models.

To model an endpoint, we need to use the visual constructs of the DSVL we have developed as building blocks to draw a model diagram to process the service requests from its SUT. These visual constructs must be instantiated to instances first by filling in their properties before they can be put into the diagram. In the followings of the thesis, we will assign a name with initial capital letter followed by lower-case letters (for example: Service Node) to each visual construct of our DSVLs. For their instances, we use the corresponding visual construct's name plus "instance" (for example: Service Node instance) or simply all lower-case letters (for example: service node).

4.3.1 Signature Domain-Specific Visual Language

Signature DSVL is based on the endpoint signature metamodel, described in Section 3.4.4.1 (refer to Figure 3.4). The metamodel uses WSDL 1.1 specification with a three-level hierarchical structure design to specify operation, request and response messages and their parameters. Hence, we design our Signature DSVL as three sub-DSVLs in hierarchical style, called WSDL, Operation and Message.

4.3.1.1 WSDL sub-DSVL

Signature top level WSDL sub-DSVL consists of a root Definition and other five entities: Service, Port, Binding, Porttype and Operation, and two relationships Composition and Association to link them together (refer to Figure 3.4a). We use a dialog box (see Figure 4.1) to specify Definition, instead of a visual construct. This is because the information specified by Definition applies to entire endpoint signature modeling. Definition properties mainly include endpoint name, service address and database access information. The other five entities and two relationships are listed in Table 4.2, providing

a detailed description of these visual constructs, such as their visual symbols, properties and inter-relationships.

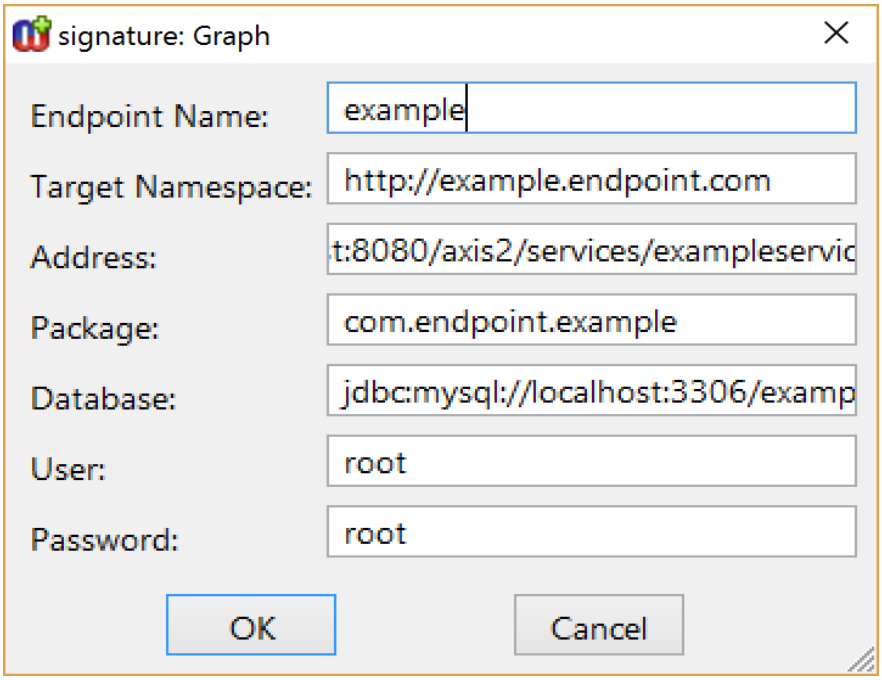
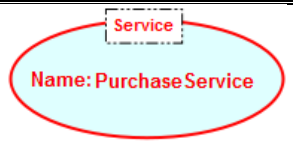
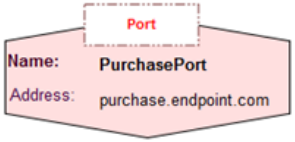
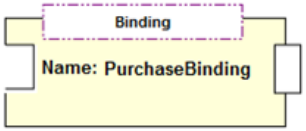
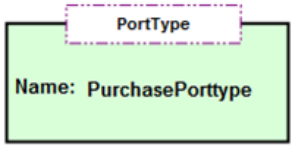
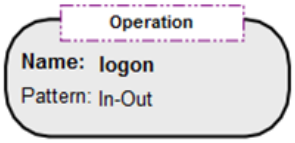


Figure 4.1. Signature WSDL sub-DSVL dialog box

To maximize the cognitive effectiveness of WSDL sub-DSVL, we have applied six PoN principles to our visual constructs' design as follows:

- Semiotic Clarity – the five entities and two relationships have their unique semantic concept in a signature model;
- Perceptual Discriminability -- shape and colour are the primary visual variables to distinguish among the five entities and two relationships;
- Complexity Management – Operation entity most likely has multiple instances in a graph, and we design it with hierarchical structure to manage the viewing complexity of endpoint signature models. Furthermore, all entities only show the essential information for modeling and hide the details;
- Cognitive Integration – To make navigations between operation instances and their child diagrams easily, child diagrams are opened by mouse clicking their parent visual construct;
- Dual Coding – the five entities have a textual annotation to supplement their visual variables;

Table 4.2. WSDL sub-DSVL visual constructs

Visual Constructs	Visual Symbol	Description	Property	Inter-Relationship
Service		A set of system operations that are exposed to Web-based protocols.	Name: A Service instance name.	A Service instance consists of one or more Port instances.
Port		Address or connection point to a Web Service. It is typically represented by a simple HTTP URL string.	Name: A Port instance name. Address: The network address at which the Service is offered.	A Port instance is associated with one Binding instance.
Binding		The Binding entity specifies interface, SOAP binding style and transport protocol.	Name: A Binding instance name. Type: To identify the kind of binding details contained in a Binding entity instance.	A Binding instance is associated with a Porttype instance.
Porttype		The PortType entity defines a Web Service, operations that can be performed, and the messages that are used to perform the operation.	Name: A Porttype instance name. Extends: A lists of Porttype entities that this Porttype derives from. StyleDefault: To construct the properties of all operations contained within the Porttype.	A Porttype instance consists of one or more Operation instances.
Operation		A Web Service SOAP action and the way a message is encoded. An operation is like a method or	Name: Operation instance name. Pattern: A template for the exchange of one or more messages.	

		function call in a traditional programming language.		
Composition relationship		To link an entity instance to an instance of its component entity.	N/A	
Association relationship		To link an entity instance to an instance of an associated entity.	N/A	

- Graphic Economy – there are only seven visual vocabularies used in the sub-DSVL, including five entities and two relationships.

We apply some domain rules to these five WSDL entities and list them in Table 4.3. These rules mainly include allowable instances of these entities in a WSDL model, valid inter-relationships among them, and specific restrictions on their properties.

Table 4.3. WSDL sub-DSVL domain rules

Visual Construct	Rule
Service	• The occurrence of its instance(s) must be equal or greater than one in a signature model; • Its name property cannot be null; • Its name property must be unique in a signature model; • An instance must be linked with one or more Port instance(s) by (a) Composition relationship(s).
Port	• The occurrence of its instance(s) must be equal or greater than one in a signature model; • Its name and address properties cannot be null; • Its name property must be unique in a signature model; • An instance must be linked with exact one Service instance by a Composition relationship.
Binding	• The occurrence of its instance(s) must be equal or greater than one in a signature model; • Its name property cannot be null; • Its name property must be unique in a signature model; • An instance must be linked with exact one Port instance and one Porttype instance by an Association relationship.
Porttype	• The occurrence of its instance(s) must be equal or greater than one in a signature model; • Its name property cannot be null; • Its name property must be unique in a signature model; • An instance must be linked with exact one Binding instance by an Association relationship; • An instance must be linked with one or more Operation instance(s) by (a) Composition relationship(s).
Operation	• The occurrence of its instance(s) must be equal or greater than one in a signature model;

	• Its name property cannot be null; • Its pattern property must be in/out, out/in, in-only or out-only: • Its name property must be unique in a signature model; • An instance must be linked with exact one Porttype instance by a Composition relationship.

Figure 4.2 shows an example endpoint signature model of an online banking application. A Service instance Exampleservice at the right defines the testing service to be provided to the endpoint SUTs, and its address is provided through a Port instance ExamplePort. A Binding instance ExampleBinding associates the Service and Port instances with a Porttype instance ExamplePortType. The Porttype instance is associated with six Operation instances, each has a sub-diagram to define its message details. Based on our knowledge, a typical endpoint application may have 10 to 20 instances of Operation entity, and one instance for each of the other four entities. Modeling such an endpoint will not cause diagram complexity problem.

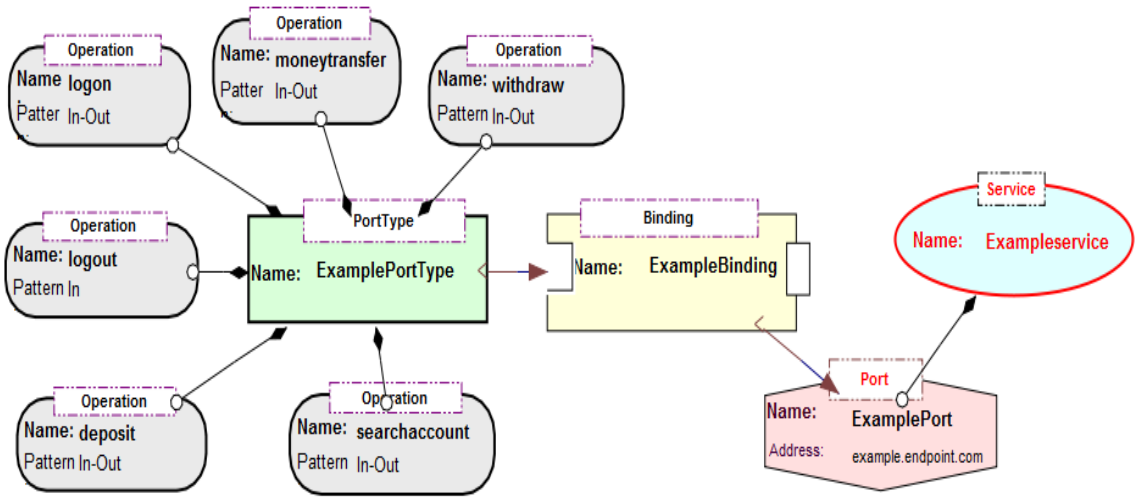


Figure 4.2. An example endpoint signature WSDL model

4.3.1.2 Operation sub-DSVL

Operation sub-DSVL is composed of only one visual construct – Message to define request and/or response message(s) in an Operation (see Figure 3.4b). The visual construct is designed by applying to Semantic Transparency principle, and an iconic envelope symbol is used for showing a message to be sent. Message visual construct has two properties:

- **Name** – Instance name of Message. The property cannot be null and must be unique in an Operation model;
- **Label** – A fixed list with values "In" and "Out" to represent whether the Message is a request or response. The pattern property of an Operation instance in signature WSDL model determines whether its corresponding operation model has an "In" Message instance only, an "Out" Message instance only, or both "In" and "Out" Message instances.

Figure 4.3 shows an Operation instance, containing both a request at the left and response at the right. It can be easily opened by doubly clicking the corresponding Operation instance in WSDL model. Obviously, this hierarchical structure helps to manage WSDL model complexity. Otherwise, signature WSDL model will be overcrowded with too many Message symbols.

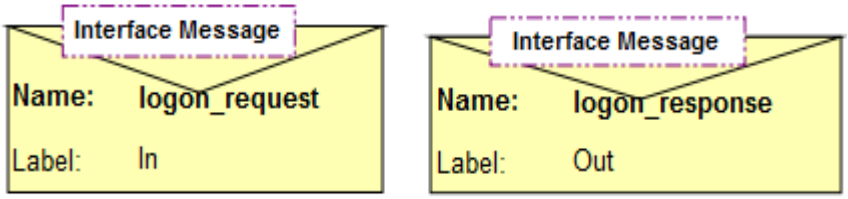


Figure 4.3. An example Operation instance

4.3.1.3 Message sub-DSVL

Message sub-DSVL also has only one visual construct -- Complex Element to define one or more parameter(s) in a Message instance (see Figure 3.4c). To apply Semantic Transparency principle, we design the sub-DSVL visual construct as an iconic element symbol with a textual annotation. It has seven properties to specify a Message instance. We provide the description and use constraints for these properties in Table 4.4.

Table 4.4. Complex Element visual construct properties

Property	Description	Constraint
ID	The parameter position in a message.	It is mandatory and must be in alphabetic sequence, starting from letter "a".
Name	The name of the parameter.	It is mandatory and must be unique in a message model.

Type	The type of the parameter.	It is a fixed list property with values of "int", "float", "String", "date" and "undefined"; When the "undefined" is chosen, a composite parameter can be defined.
Mandatory	It is boolean type to define whether the parameter is mandatory.	It must be a value of "T" or "F" to represent true or false selection.
Default	The default value of the parameter.	It is nullable.
Minimum	The minimum allowable value of the parameter.	Its type must be integer, float or date; It is only applicable, if the parameter type is "int", "float" or "date".
Maximum	The maximum allowable value of the parameter.	Its type must be integer, float or date; It is only applicable, if the parameter type is "int", "float" or "date".

Figure 4.4 shows the three elements of an example Message instance. The first element name is "inputuserid". It is a mandatory integer with five digits. The last two elements are "inputusername" and "inputpassword". They are optional, and their data types are string. As we would expect many elements could be reused within an endpoint or even across endpoints, reusability is the primary consideration to design Message sub-DSVL.

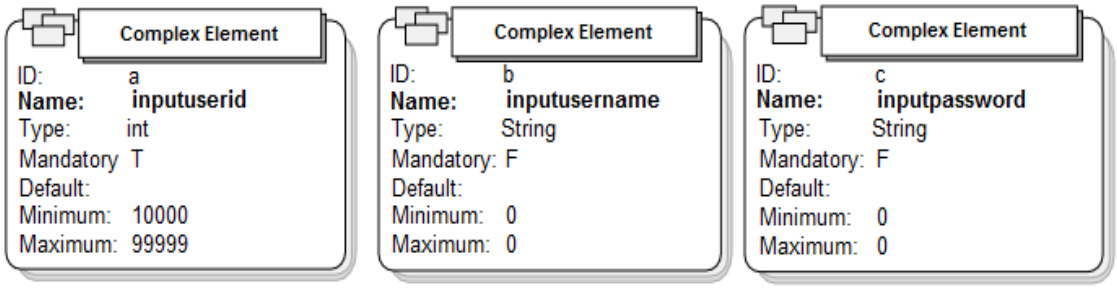


Figure 4.4. An example Message instance

4.3.2 Protocol Domain-Specific Visual Language

Our Protocol DSVL implementation is based on the metamodel depicted in Figure 3.5 of Chapter 3. The metamodel uses an Extended Finite State Machine (EFSM) to capture static and dynamic endpoint protocol aspects. The EFSM mainly includes an endpoint state entity with an idle, a home and different working states and a transition function triggered by operations and internal events. We create three state visual constructs for

representing endpoint's different states and four relationships for managing endpoint state transitions.

To discriminate the instances of different states, we use two visual variables of shape and colour, textual annotation, plus a symbolic icon at the top-left conner. There are always an idle state and a home state instances, and one or more working states in any endpoint protocol models. The instances of different states can be easily identified with sufficient visual distance. Except for the name, working state entity also has four properties to simulate some protocol scenarios. We will discuss how to setup these properties in Chapter 5.

The visual variables for the relationships include shapes at both ends, colour, line type and textual annotation. There are four relationships of Transition, ConstraintTransition, Timeout and Loop used in protocol model. The properties of Transition, Timeout and Loop relationships can be seen from protocol model diagram directly. To see and define ConstraintTransition condition instance, we must open its dialog box shown by Figure 4.5. Comparing with other two functional layer models, protocol layer models are relatively simple. For this reason, we use a flat modeling structure.

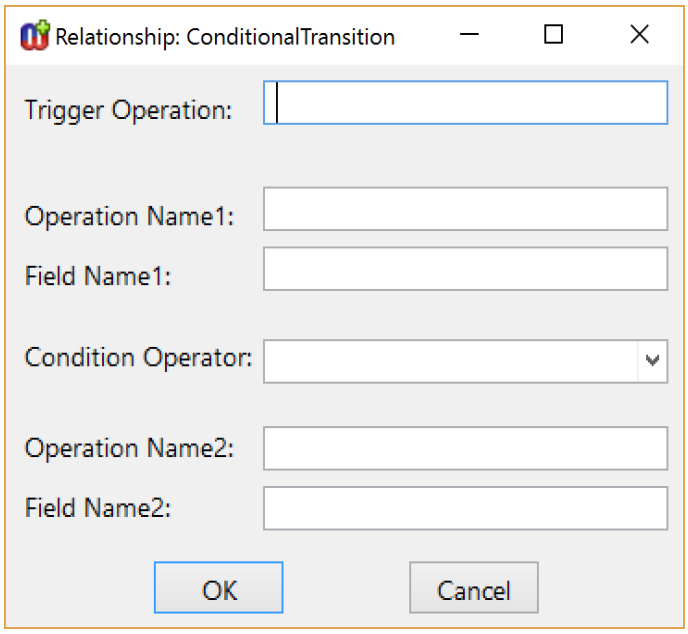


Figure 4.5. ConstraintTransition relationship dialog box

We give the design details of our Protocol DSVL visual constructs in Table 4.5. The table includes all the essential information for users to use these visual constructs to model

endpoint protocol layer. The visual symbols provide the building blocks for intuitionistic endpoint protocol models for both developers and end users. The descriptions describe the semantics and explain the use of these visual constructs. The lost column lists the properties of each visual construct and discusses how they can be filled.

We use a simplied banking system to show how an endpoint protocol layer can be modeled in Figure 4.6. A “logon” operation moves the endpoint from its idle state to home state, ready to provide testing services to its SUTs. In the reverse direction, a “logout” operation or “timeout” event deactivates the endpoint. There are three states to move when the endpoint is at its home state, triggering by different operations. Among them, the transitions to “deposit” and “withdraw” states are conditional, subject to the success of user authentification. “moneytransfer” is a slave transition, it can only be reached when the endpoint is at its “searchaccount” state.

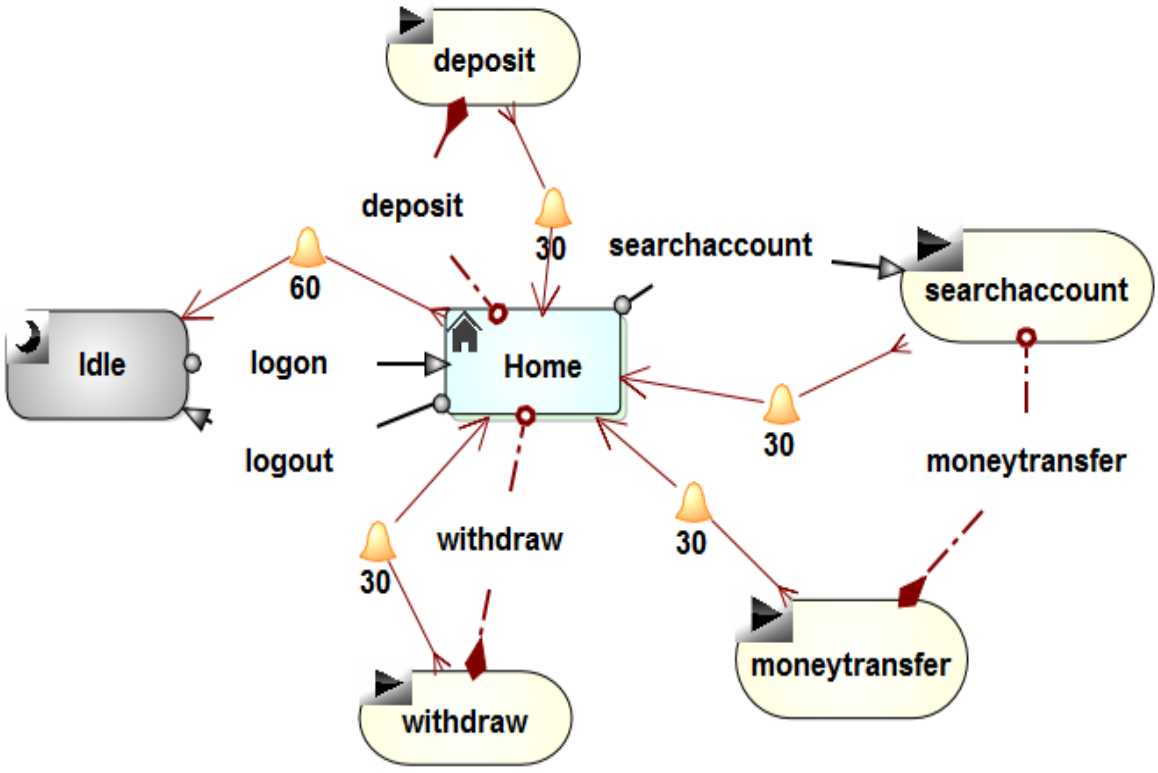
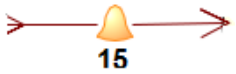
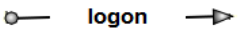
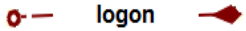


Figure 4.6. An example endpoint protocol model

Table 4.5. Protocol DSVL visual constructs

Visual Construct	Visual Symbol	Description	Property
Working State		It presents an endpoint state, which normally uses operation as its default name.	Name: State instance name; Synchronous Operation: Is the state operation in synchronous mode? Processing Time: Simulated operation processing time in seconds; Safe Operation: Is the state operation safe? Transmission Time: Simulated operation request transmission time in seconds.
Home State		It is a special endpoint state, representing endpoint in active status.	N/A
Idle State		It is a special endpoint state, representing endpoint in inactive status.	N/A
Timeout Relationship		It links a “from” state to a “to” state for representing endpoint state transition; The transition will happen, if no valid operation request is received within a defined timeout period.	Time: The time in seconds for an automatic state transition.
Transition Relationship		It links a “from” state to a “to” state for representing a state transition.	OperationName: The operation triggers the state transition.
Constraint Transition Relationship		It links a “from” state to a “to” state for representing a state transition;	Trigger Operation: The operation triggers the state transition;

		The transition is subject to a constraint condition, defined by its dialog box (see Figure 4.5).	Operation Name1 + Field Name1: They defines the first state transition condition; Condition Operator: It is used to compare the two conditions; Operation Name2 + Field Name2: They defines the second state transition condition.
Loop		It defines a repeat state transition from a “from” state to a “to” state.	Loopnumber: The state transition will repeat for the number of times.

We apply some domain rules to the three endpoint states and four state transition relationships, and they are listed in Table 4.6. Specifically, the occurrence of home state and idle state must be exactly one and working state must be at least one. ConstraintTransition is specified by comparing two operation fields, which must have been defined in the endpoint signature model.

Table 4.6. Protocol DSVL domain rules

Visual Construct	Rule
Working State	• The occurrence of its instance(s) must be equal or greater than one in a protocol model; • Its name property cannot be null; • Its name property must be defined in the signature model as an operation; • Its name property must be unique in a graph.
Home State	• The occurrence of its instance must be one in a protocol graph; • An instance must be linked with exact one idle state instance in a protocol model.
Idle State	• The occurrence of its instance must be one in a protocol model; • An instance must be linked with exact one home state instance.
Timeout relationship	• Its time property must be an integer data type, and must be equal or greater than zero; • It must link two endpoint states but cannot start from idle state instance.
Constraint transition relationship	• It must link two endpoint states. • Its trigger operation property must be defined in the signature model as an operation; • Its operation name1 property must be defined in the signature model as an operation; • Its field name1 property must be a parameter of the operation; • Its operation name2 property must be defined in the signature model as an operation; • Its field name2 property must be a parameter of the operation; • Its condition operator must be valid.
Transition relationship	• It must link two endpoint states. • Its operation name property must be defined in signature model as an operation.
Loop	• It must link two endpoint states.

	• Its loop number property must be an integer data type, and must be greater than zero.

4.3.3 Behavior Domain-Specific Visual Language

Our Behavior DSVL is based on a hierarchical structure design and dataflow metaphor. Its top level contains service nodes for defining operations provided by an endpoint and data stores for specifying persistent data tables. A service node may contain a number of nodes (also called methods) to handle one or more specific tasks. These nodes can be further decomposed into more specific sub-nodes if needed. At the bottom level, nodes contain only primitive programming constructs to perform operations on data and control process flows. Here, we introduce the key visual constructs used in our Behavior DSVL:

- **Service Node** -- is used to process operation requests and generate corresponding responses;
- **Node** – is a component of a service node for handling a specific task. Depending on how complicated the task is, the node can be further decomposed into more sub-nodes or use primitive programming constructs to model the task;
- **Arc** – connects nodes, primitive visual constructs and entrance and exit bars within a service node or a node to control data and process flows;
- **Entrance and Exit Bars** – Every service node or node has a pair of Entrance and Exit bars. All visual constructs are placed between these two bars;
- **Data Store** – is used to create data tables for storing persistent data;
- **JDBC Operator** – provides a graphic user interface for users to specify persistent data operations. It invokes a JDBC module in the domain framework to access and manipulate persistent data;
- **Evaluator --** performs arithmetic operations on input variables and assigns the result to an output variable;
- **Conditional Operator** -- tests two input parameters for determining alternative process flows;
- **Loop** -- includes a set of visual constructs to be executed repeatedly for pre-defined times or condition;

- **Variable** -- represents a variable with a specified data type;
- **Variable Array** -- stores multiple variables with a same data type.

In general, we apply four PoN principles when designing our behavior DSVL. To perceptually discriminate the visual constructs in a behavior model, we use shape as the main visual variable, plus textual annotation. To manage diagram complexity, only key information is shown on the appearance of visual constructs, and details are hidden in their dialog box. For users to guess the meanings from their appearances, Semantic Transparency principle is applied to some visual constructs. The last one is about Semiotic Clarity, each visual construct has exact one correspondence semantic concept in behavior modeling.

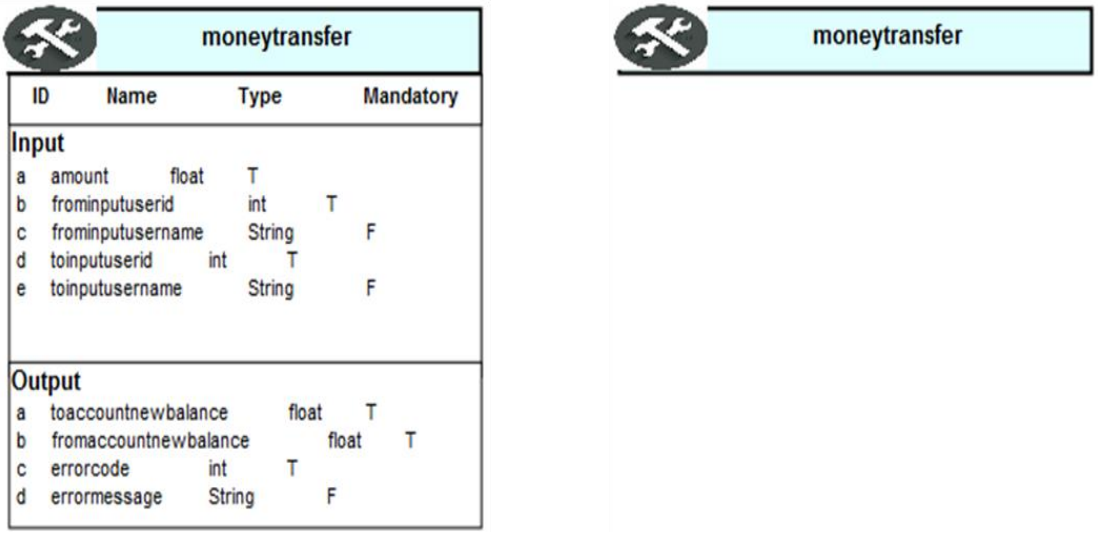
4.3.3.1 Service Node

A Service Node instance is created by entering an operation name, which must have been defined in the endpoint signature model. Once a matching operation is found, the parameters in both the request and response messages are imported from the operation definition.

Figure 4.7a shows a Service Node instance. To help users to model operation behavior, we design the visual construct to show all request and response parameters and their properties. We add a symbolic operation icon at the top-left to differentiate service nodes from nodes. As the root visual constructs of endpoint behavior models, Service Node instances do not have input and output ports for receiving inputs and sending results from/to other instances of Service Node or Node. To manage behavior model view complexity, service nodes can be collapsed to hide parameters for reducing their symbol size (see Figure 4.7b).

There are several service node definition rules:

- The instance name cannot be null and must have been defined in the endpoint signature model;
- Each operation defined in the endpoint signature model must have only one corresponding instance of Service Node;
- A Service Node instance must contain one and only one pair of entrance and exit bars.



[a] A Service Node instance definition [b] A collapsed Service Node instance

Figure 4.7. An example Service Node instance

4.3.3.2 Node

A Node instance is located inside a Service Node or its parent Node instance as a data processing unit for performing one or more specific tasks. Normally, a service node contains a number of nodes, connecting by arcs to form a data processing chain from the "out" port of a node to the "in" port of the next one. Occasionally, some nodes may not be able to execute successfully, subject to their input data. To handle this abnormal situation, there are two alternative "out" ports to determine the next node to be executed, depending on whether the current execution is successful or failed. When a service node is activated, the first node takes the input parameters from a request to process and the generated results are forwarded to the next in the chain. By going through the chain, the last node places the final results to the response.

Figure 4.8 shows a Node instance, which is a thick line round rectangle filled with grey colour. There is a small hollow circle on the top as the "in" port for data flowing in the node. The normal "out" port is a black circle and the "exceptional out" port is a yellow circle. Both the ports are located at the bottom of Node main construct at the left and right.

There are several node definition rules:

- A Node instance must be defined in a Service Node instance;
- Node instance name cannot be null and must be unique in a behavior model;

- Each Node instance must contain a pair of entrance and exit bars.

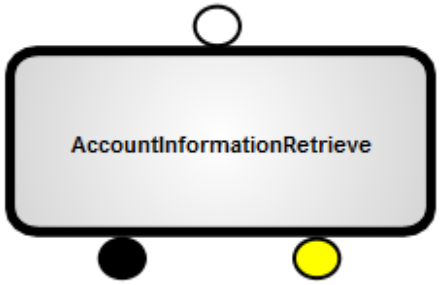


Figure 4.8. An example Node instance

4.3.3.3 Arc

An Arc instance is used to link two nodes in a service node or two primary visual constructs in a node. It links a "from" entity from its "out" port to a "to" entity "in" port. By doing so, the arc passes the "from" entity results to the "to" entity as its input parameters and hands over the execution. Some entities may require input data from more than one entities, and they have multiple arcs pointing to their "in" port. Whether these entities are ready to run are also subject to their internal mechanism to ensure the availability of all required parameters.

Arc visual construct is a black arrow line, pointing to the "in" port of the next entity to run.

There is one rule for the use of Arc: an Arc instance must start from an "out" (or "exceptional out") port and end at an "in" port.

4.3.3.4 Entrance and Exit Bars

All service nodes and nodes have a pair of Entrance and Exit bars to specify their input and output parameters and define where execution starts and ends.

Figure 4.9 shows these two visual constructs, which are in trapezoidal shape with shorter edge facing inside. The entrance bar has one "out" port underneath, and the exit bar has a normal "in" and an "exceptional in" ports on it. The parameters for both bars can be displayed or hidden by users, depending on whether they need to know these parameters. The properties of the parameters include name, data type, mandatory (T) or optional (F), and default value. The entrance and exit bars' definition are different for a service node

and node. The parameters of the service node's entrance and exit bars are imported from the endpoint signature model. While the parameters for the node's one need to be specified by users.

There are several entrance and exit bars' definition rules:

- A service node and node must have exact one pair of entrance and exit bars;
- Parameter property definitions must follow the rules specified in Table 4.4;
- An entrance bar "out" port must have one or more linked arc(s) to specify the execution starting point;
- An exit bar "in" port must have one or more arc(s) linked to it for specifying the normal execution ending point.

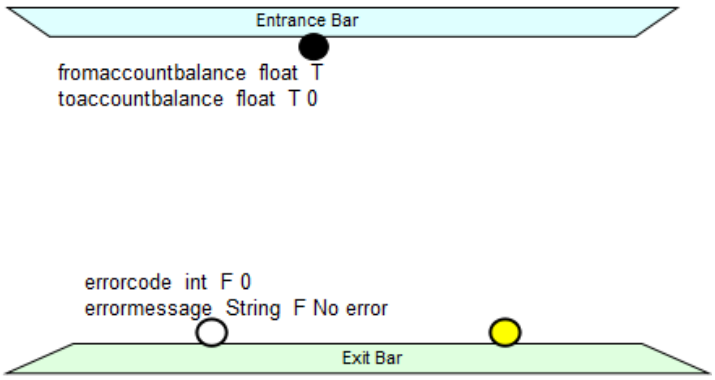


Figure 4.9. An example of Entrance and Exit bars

4.3.3.5 Data Store

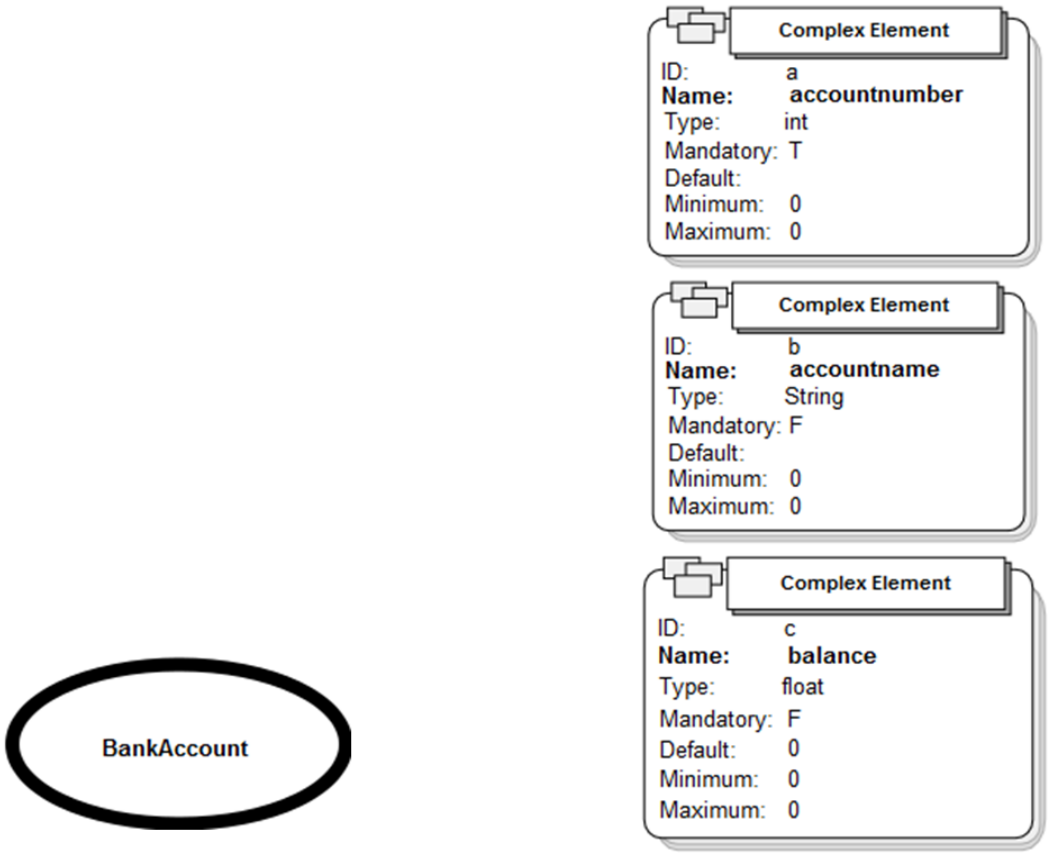
We reuse signature Message sub-DSVL as our Data Store visual construct to create data tables in MySQL database. Same as message element definition, we have to specify the properties of each table field. In addition, tables must have a primary key, and the first field of a data store is considered as the primary key. To handle more complex data structures, data store can define a slave table by specifying its foreign key field as "undefined" data type in its master table. By doing this, a slave table can be defined inside a data store.

Figure 4.10a shows a Data Store instance BankAccount of an endpoint behavior model. We use a thick line oval symbol to represent Data Store visual construct. Figure 4.10b

illustrates the three fields of the table, defined by Message sub-DSVL. Among them, the first field “accountnumber” is the primary key of the table.

There are two data store definition rules:

- Table name cannot be null and must be unique in the model;
- Properties of a table field must follow the rules specified in Table 4.4.



[a] A Data Store instance [b] Table field definition

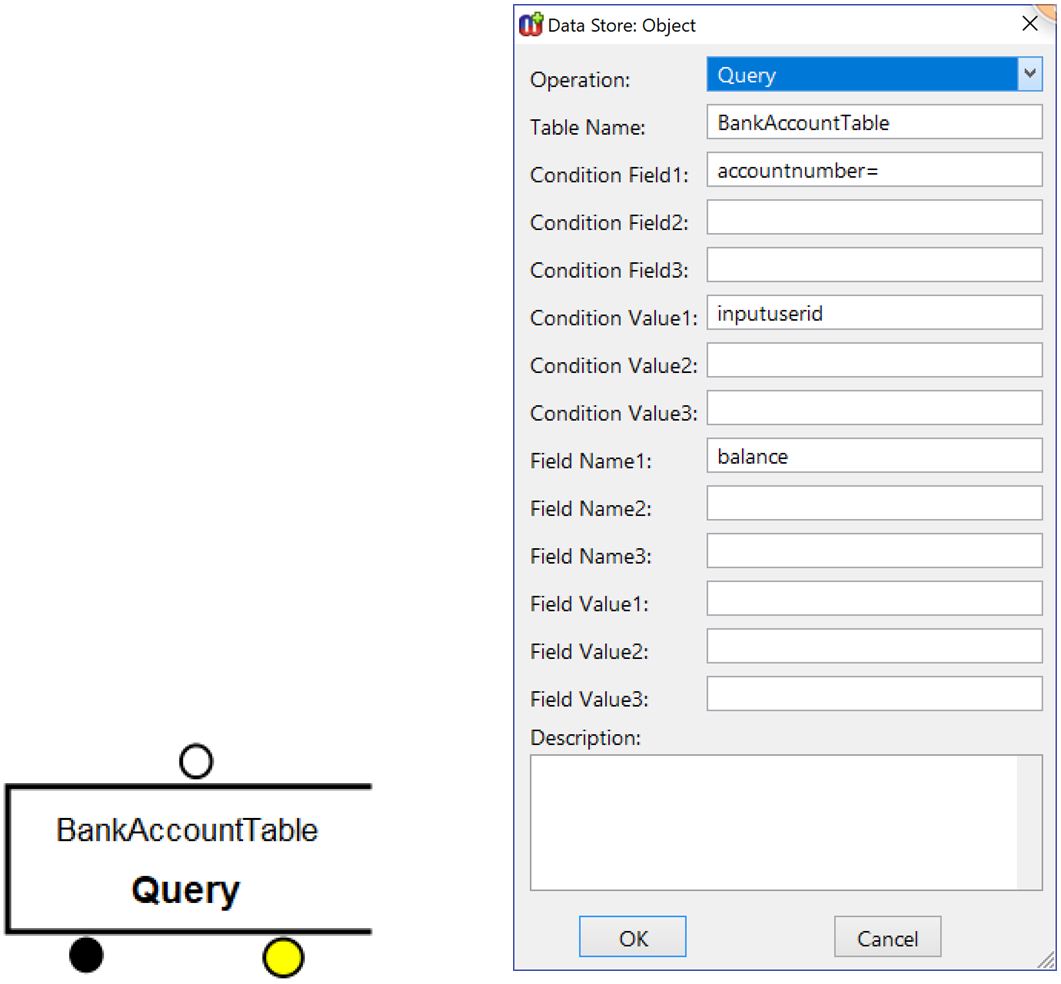
Figure 4.10. An example Data Store instance

4.3.3.6 JDBC Operator

JDBC Operator allows users to specify Select, Insert, Update and Delete SQL commands through use of a graphic user interface. This kind of SQL commands normally require users to provide table name, records selection criteria, and values assignment to the selected table fields.

We design its visual construct as an open rectangle symbol with table name and SQL command (refer to Figure 4.11[a]), as we believe they are the key information to be

known in a behavior model. The SQL command information is specified using the visual construct dialog box, and all the dialog box fields are described in detail by Table 4.7. As an example, Figure 4.11[b] illustrates how a query statement is defined of Figure 4.11[a] by filling the dialog box. The query is conducted on table BankAccountTable; record section criterion is "accountnumber" equal to "inputuserid"; and table field "balance" is retrieved for this query.



[a] JDBC Operator visual construct [b] JDBC Operator dialog box

Figure 4.11. An example JDBC Operator instance

Table 4.7. JDBC Operator properties

Name	Description	Rule
Operation	A valid operation list for users to select.	It must be Select, Insert, Update or Delete.

Table Name	The table to be accessed.	It cannot be null and has been created by data store construct.
Condition Field1	The first table field for searching table records.	If not null, it must be a field of the table.
Condition Field2	The second table field for searching table records.	If not null, it must be a field of the table.
Condition Field3	The third table field for searching table records.	If not null, it must be a field of the table.
Condition Value1	The test value for the first record search field.	It must have the same data type as the Condition Field1.
Condition Value2	The test value for the second record search field.	It must have the same data type as the Condition Field2.
Condition Value3	The test value for the third record search field.	It must have the same data type as the Condition Field3.
Field Name1	The first table field to be retrieved, created or updated.	The field is applicable, only if the operation is Select, Insert or Update; It must be a field of the table.
Field Name2	The second table field to be retrieved, created or updated.	The field is applicable, only if the operation is Select, Insert or Update; It must be a field of the table.
Field Name3	The third table field to be retrieved, created or updated.	The field is applicable, only if the operation is Select, Insert or Update; It must be a field of the table.
Field Value1	The assigned value to the first table field.	The field is applicable, only if the operation is Insert or Update; It must have the same data type as the Field Name1.
Field Value2	The assigned value to the second table field.	The field is applicable, only if the operation is Insert or Update; It must have the same data type as the Field Name2.
Field Value3	The assigned value to the third table field.	The field is applicable, only if the operation is Insert or Update; It must have the same data type as the Field Name3.

4.3.3.7 Evaluator

Evaluator is used for performing arithmetic operations on input parameters and assigning computational result to a variable. The valid arithmetic operations include addition (+),

subtraction (-), multiplication (*), division (/), exponentiation (**), logarithmic functions (log), and trigonometric functions (sin, cos, tan and cot). These operations are performed on integers and real numbers, and their processing order follows PEMDAS (Parentheses, Exponents, Multiplication/Division, Addition/Subtraction) precedence order.

Figure 4.12 shows an Evaluator instance, which is a rectangle filled with blue colour. The "in" port on the top takes input parameters from a single or multiple entities executed beforehand, depending on how many arcs are connected to it. Evaluator visual construct has three lines for specifying a formula. The first line defines the result variable to be assigned after the execution. The second line lists all parameters to be used by the evaluator, and they are separated by commas. The last line is the arithmetic formula with parameters in a "P" array. The order of the array elements follows the sequence of the parameters in the second line. The use of simple array elements will have a much more concise formula representation, comparing with long parameter names. Certainly, this will help to manage graphic complexity.

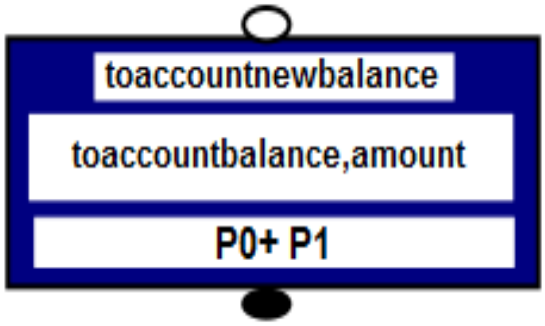


Figure 4.12. An example Evaluator instance

There are several evaluator definition rules:

- The first line variable name cannot be null;
- All arithmetic operation parameters must be defined, and their values are assigned beforehand;
- The number of the P array elements must be equal to the number of the parameters.

4.3.3.8 Conditional Operator

Conditional Operator tests two input parameters with either numeric data type or string based on a user defined comparator, and the result determines alternative process flows.

The comparators for numerical parameters include "=", ">", "<", "<=" and ">=". Unlike numeric data comparison, only "equal" is used to compare two strings.

Figure 4.13 shows a Conditional Operator instance, which is in a diamond (rhombus) shape with two key properties of data type and comparator displayed. We design our Conditional Operator visual construct by referring to the decision symbol of flowchart diagram. So that, most IT professionals should be familiar with it. Two testing parameters flow in the conditional operator from its two "in" ports on the top. If the testing result is true, the next node or programming construct to be executed will follow the arc from the normal "out" port underneath. Otherwise, the "exceptional out" port at the right will be followed.

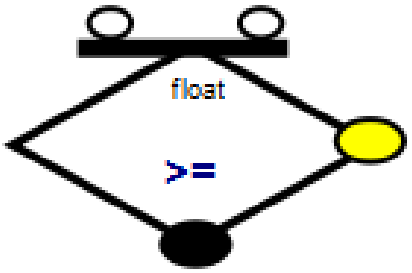


Figure 4.13. An example Conditional Operator instance

There are several conditional operator definition rules:

- Two parameters must match with their data type. Otherwise, the testing result is "false";
- Variable(s) used must be defined beforehand;
- If one or both parameters are missing, the testing result will be "false".

4.3.3.9 Loop

Like most popular third-generation languages, we define two loop types of For-loop and While-loop to execute a block of programming constructs repeatedly. Similar to a node, a loop contains some primary visual constructs to process business logics, and arcs link these visual constructs to control process sequence and direct data flows. On the other hand, a loop does not take any input parameters and generate results as a node does. So, there are no entrance and exit bars within a loop body. Instead of using entrance and exit bars to mark the starting and ending points, the visual construct to be executed first is the

one without an arc pointing to its "in" port. At the other end, the visual construct without an arc from its "out" port is the last to be executed.

For-loop tests an iterator at the start of a loop to determine whether the loop body is to be executed. The iterator is assigned an initial number when the loop starts, and it increases (or decreases) by a pre-defined increment (or decrement) for each iteration. The loop is terminated when the iterator reaches a final number. Instead of using an iterator, While-loop has a boolean expression at the end of a loop body. The boolean expression has a variable with a new value assignment for each iteration and a user defined invariable. The testing result determines to continue for the next iteration or exit the loop body.

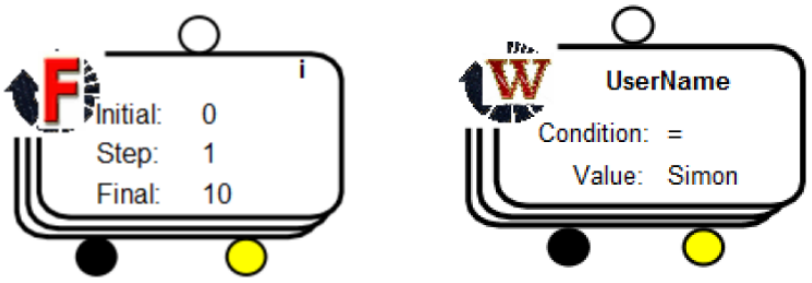
We design For-loop visual construct as a round rectangle with iterative shapes behind the rectangle to represent a repeating construct (refer to Figure 4.14a). A loop symbol with "F" icon at the top-left corner provides redundant coding to reduce errors and counteract noise. For users to quick identify loop information, the iterator "i" is located at the top-right corner. The iterator of the figure example has an initial value 0 before the loop starts, increases by 1 for each iteration, and terminates the loop when reaching the final value 10.

A While-loop example is shown by Figure 4.14b. We use a similar visual design as For-loop, since both represent the iteration of a block of programming constructs. To discriminate it from For-loop visual construct, we use "W" loop symbol and different textual messages shown on the construct. The figure example is a loop of SQL statements to search a record in a user database. The loop variable is UserName, and the process flow will exit the loop if the user record with user name "Simon" is found.

There are several loop definition rules:

- For For-loop, the initial, step, final and loop counter must be integer data type;
- For For-loop, if the initial value is greater than the final value, its body will not be executed;
- For While-loop, the variable and pre-defined value must have the same data type. Otherwise, it will not be executed;
- For While-loop, the variable must be assigned a new value for each iteration;

- For While-loop, endless iteration will occur, if the variable is never equal to the pre-defined value.



[a] For-loop visual construct [b] While-loop visual construct

Figure 4.14. Examples of For-loop and While-loop instances

4.3.3.10 Variable and Variable Array

A Variable instance is used to hold an intermediate result to be used by any visual constructs later on. It can be either assigned by a primary visual construct executed earlier or a constant. A variable is not necessary unique in a behavior model, and its value can be overwritten by the same name variable executed later.

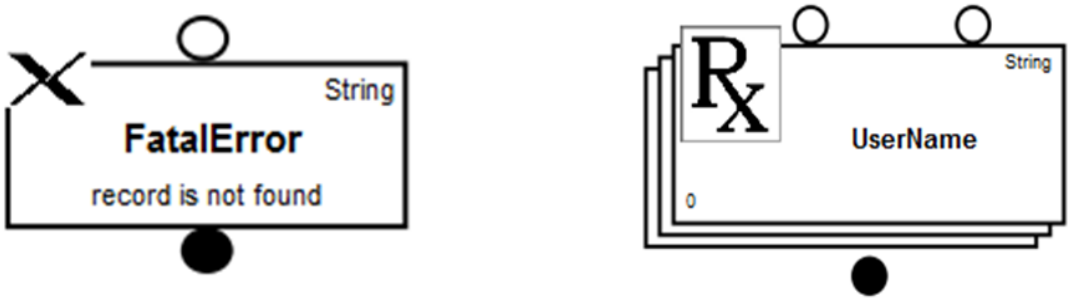
Figure 4.15a shows an example Variable instance. We design it in a rectangle shape with a "X" icon at the top-left corner.

A Variable Array instance holds multiple variables of same data type. It has two "in" ports, the left one is for assigning a value to a specific array element and the right one provides the index for the element. Figure 4.15b shows an example Variable Array instance. Its visual construct is a rectangle with iterative shapes behind the rectangle to represent a repeating construct, and a "RX" icon is at the top-left corner.

There are several variable and variable array definition rules:

- The name property cannot be null for both variable and variable array;
- The data type must be "String", "Date", "int" or "float" for both variable and variable array;
- The data type of an assigned value must be the defined data type;
- The assigned index of a variable array must be equal or greater than zero;

- The assigned index of a variable array must not be greater than the defined array size.



[a] Variable visual construct [b] Variable Array visual construct

Figure 4.15. Examples of Variable and Variable Array instances

4.3.3.11 A Behavior Model Example

Here we take a simplified online banking endpoint as an example to illustrate how a behavior model looks like. Figure 4.16 shows a very clean and well-presented top view of the example behavior model. It consists of six service nodes and two data stores, and they can be easily identified from their appearances. To balance the information displayed and diagram complexity, users can selectively show the parameters of some service nodes and hide others. In practise, only those to be modeled are expanded to display input and output parameters.

Figure 4.17 illustrates how a service node "moneytransfer" is modeled. The service node is decomposed into three nodes to retrieve a bank account balance, calculate the new amount, and update database, respectively (refer to Figure 4.17a). These three nodes are placed between a pair of entrance and exit bars with input and output parameters shown (see Figure 4.17a). The service requests for the operation are processed by these nodes in sequence along with those arcs from normal "out" ports. Whenever an error occurs, the process will be terminated, and an error message will be passed to the exceptional "out" port of the exit bar.

We then look into the first node to see how primary visual constructs are used for performing a specific task (refer to Figure 4.17b). The first two JDBC operators retrieve the "from" and "to" bank account balances by their user ids and names, and their results are assigned to two variables of "fromaccountbalance" and "toaccountbalance". If the "from" account is not found from the first JDBC operator, an error message will be

generated. Unlike the "from" account, a new "to" account record will be inserted by the third JDBC operator if the "to" account does not exist.

From these two diagrams, we can see that it will not be difficult to model such an endpoint behavior and the model can also be comprehended easily by domain experts.

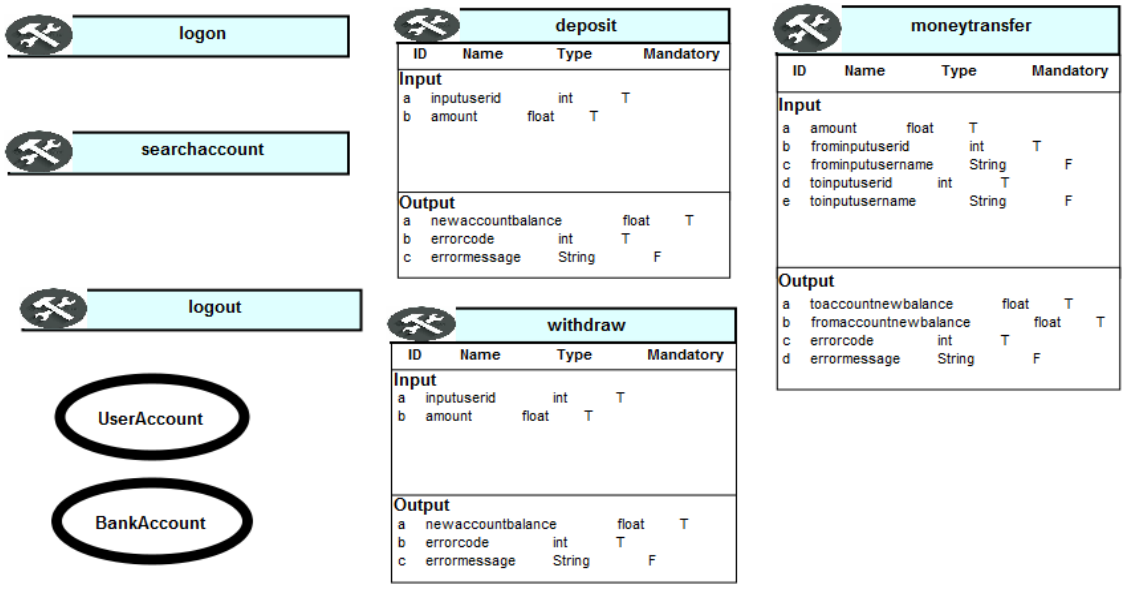
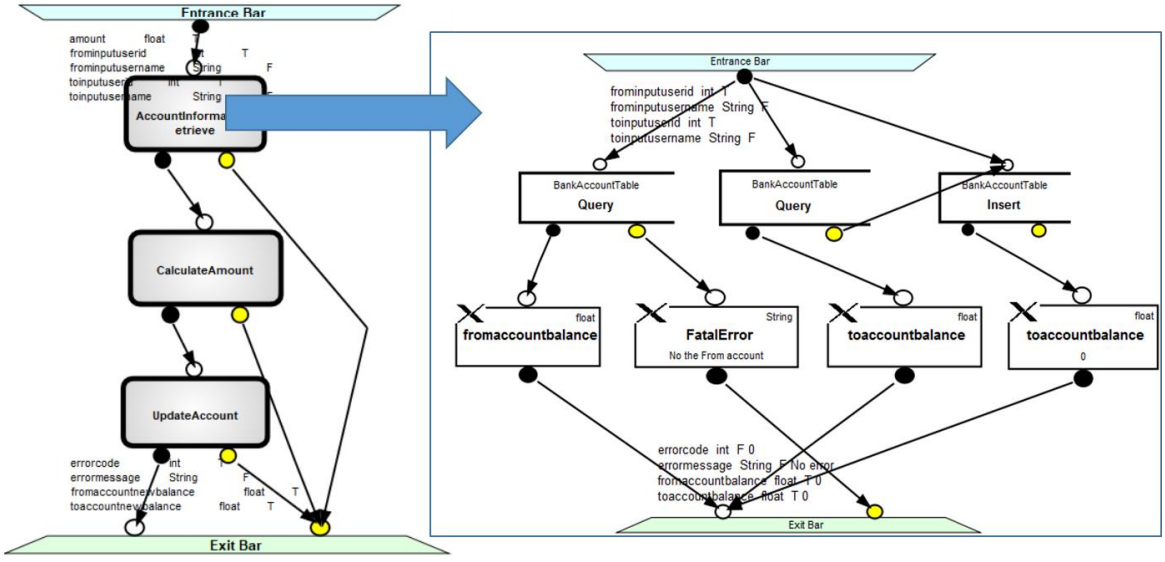


Figure 4.16. The top view of an example behavior model



[a] A Service Node instance decomposition [b] A Node instance definition

Figure 4.17. A Service Node instance definition

4.4 Implementation of Code Generators and a Domain Framework

In our DSM approach, modeling is the main task for users to develop an operational endpoint. However, there are also some additional works to be done after the modeling. These works are not trivial, and users have to pay special attention to provide the right parameters for their configurations. We list these works below:

- **Development of a Domain Framework** – the WSDL file generated from an endpoint signature model is transformed to Axis2 SOAP engine as the domain framework by using Axis2's wsdl2java tool;
- **Model Transformation** – the code generators transform endpoint models to Java classes for logic processing and SQL scripts for creating tables and storing persistent data;
- **Code Compilation** – transformed codes need to be configured and compiled with the domain framework;
- **Package Service** -- to provide testing service, all Java classes and libraries are packaged to a Tomcat service and the .aar service archive file is loaded to the Tomcat webapps folder.

After restarting the Tomcat application server, the endpoint is ready to provide its testing service to its SUT through http application protocol.

A good DSVL should let its users focus on application modeling and release them from unnecessary overheads for model transformation, code compilation and endpoint deployment by means of task automation. Particularly, our target users are non-technical background domain experts. They may not be good at IT technologies and cannot configure endpoint generation properly. From a development productivity point of view, it is also desirable to have these activities done automatically. For these reasons, we have developed a supporting toolset to automate operational endpoint generation from models.

4.4.1 Code Generators

A DSL code generator works in a similar way as a compiler, which translates programming code written in a third-generation language to a lower-level programming code, such as assembler. A code generator accesses models, extracts information from them, and transforms the models into output in a specific form. This process is guided by

the concepts, semantics and rules of the modeling language. It is also subject to the input syntax required by domain framework and target environment. A generator normally has some conditional statements for generating different codes depending on the values in a visual construct, the relationships it has with other visual constructs, or other information in a model. The generated code must be complete and in production quality. After generation, the code does not need rewriting, inspection or additions manually.

In general, a code generator fulfils three main tasks:

- **Accessing Model** – The code generator starts to navigate a model from the root element and goes through all other model elements based on their relationships. It seeks for certain object types, dependencies on the various relationships and connection types the model has;
- **Extracting Model Data** – The code generator extracts data by analysing combinations of model elements, such as the relationships connecting them, the sub-models an element has, or other linkages between model elements;
- **Transforming Model to Output Code** – With the extracted model data the code generator adds additional information for output as well as integrating with domain framework code or making calls to the underlying target environment and its libraries.

A DSL may have more than one code generators to transform a model into different forms. In our case, for example, an endpoint signature model is transformed to a WSDL file for creating an endpoint domain framework and a Java class for validating dynamic aspects of the endpoint signature layer.

4.4.1.1 Signature DSVL Code Generator

We use WSDL specification to describe endpoint signature, and our code generator must be able to convert endpoint signature models to WSDL documents. WSDL document structure consists of eight parts in sequence -- Definition, Type, Message, Operation, Porttype, Binding, Port and Service. Among them, the first six parts are defined in the top level of signature models, and they can be easily accessed by navigating from the root through either Association or Composition relationships. In contrast, we need to iterate over the decomposition of each operation to reach message instances in a signature model,

as messages are defined within operations as subgraphs. Similarly, each message can be further decomposed into types.

To explain how our signature code generator works, we select a typical code snippet, which has a two-level decomposition to navigate to type instances by going through all the operations and messages (refer to Figure 4.18). The code snippet starts with a type part definition based on WSDL specification. Then, iteration over operations and decomposition to messages are followed. As a same message instance may be in multiple operations, message duplications must be avoided. To do this, we define a string variable $found to store all new messages and use the variable to test each message being used before writing it to the WSDL document.

```
/* Type definition */
'<wsdl:types>' newline;          /* type definition header */
'<xs:schema attributeFormDefault="qualified" elementFormDefault="qualified"
        targetNamespace="' :Target Namespace;'">' newline;
variable 'messageoid' write '.' close;  /* a variable to store new messages */
   to '%dotToNL
   . \
' endto
foreach .Operation {             /* iteration over operations */
   do decompositions {           /* decomposition over operation */
     foreach .InterfaceMessage {        /* iteration over messages */
        $found = 'F'             /* check if it is a new message */
        do $messageoid %dotToNL {if id = :Name; then
             $found = 'T'
             endif
        }
        if $found = 'F' then
          do decompositions {    /* decomposition over message */
            '<xs:element name="'; :Name '">'; newline
            '<xs:complexType>' newline
            '<xs:sequence>' newline

            foreach .ComplexType; orderby :ID {
              '<xs:element name="' :Name; '" type="xs:':Type Name
              if :Mandatory = 'F' then
                '" minOccurs="0"/>'; newline
              else
                '"/>'; newline
              endif
            }
           ' </xs:sequence>' newline
           ' </xs:complexType>' newline
           ' </xs:element>' newline
          /* append the message to the variable */
           variable 'messageoid' append '.' :Name; close
        }
      endif
    }
  }
}
'</xs:schema>' newline
'</wsdl:types>' newline newline
```

Figure 4.18. A code snippet of Signature DSVL code generator

Figure 4.19 shows the type definition part of a signature model WSDL file, transformed from the code generator snippet listed in Figure 4.18. It consists of a deposit request and response messages and defines three and two type instances, respectively. The properties of these types include name, data type and mandatory.

```
<wsdl:types>
   <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified"
                                targetNamespace="http://example.endpoint.com">
        <xs:element name="deposit_response">
           <xs:complexType>
               <xs:sequence>
                  <xs:element name="newaccountbalance" type="xs:float"/>
                  <xs:element name="errorcode" type="xs:int"/>
                  <xs:element name="errormessage" type="xs:string" minOccurs="0"/>
               </xs:sequence>
           </xs:complexType>
        </xs:element>
        <xs:element name="deposit_request">
           <xs:complexType>
              <xs:sequence>
                 <xs:element name="inputuserid" type="xs:int"/>
                 <xs:element name="amount" type="xs:float"/>
              </xs:sequence>
            </xs:complexType>
         </xs:element>
   </xs:schema>
</wsdl:types>
```

Figure 4.19. Type definition part of an example WSDL file

For numeric and date data types, they normally have a valid range defined by their minimum and maximum properties. To handle this type of endpoint signature validation, we design another signature code generator to navigate through endpoint signature models. It searches for messages with these data types and stores their lower and upper values through some SQL statements in Java language.

4.4.1.2 Protocol DSVL Code Generator

Endpoint static protocol in a specific state depends on coming operations. After receiving a valid operation, the endpoint will move to the operation state. To capture the relationships among states and operations, a protocol database is created to store endpoint current state, valid operations, destination state, etc.

Dynamic protocol aspects include constraint state transitions, time-out events and process/transmission time simulations. A constraint condition adds a restriction on an endpoint state transition after receiving a valid operation, and it is defined by comparing two operation fields. Time-out events are represented by linking a “from” state to a “to”

state using a time-out relationship and giving a time in seconds. To simulate synchronous and unsafe operation, we have to search for the states with the corresponding properties selected and store the time in the database together with the state name.

Figure 4.20 is a code snippet of the protocol code generator to insert condition state transition records to a state transition table. A constraint condition transition is modeled using a ConditionalTransition relationship to link a state as FromStateCondition role to another state as ToStateCondition role. The code generator starts to iterate over operations, then navigates along the FromStateCondition role of the operations to the corresponding ConditionalTransition relationships. The fromstate name field is from the operation name, and the state transition condition is defined by taking the properties of the ConditionalTransition relationship. To catch the tostate name, we navigate along the FromStateCondition role of each ConditionalTransition relationship to the ToStateCondition role into the object at the other end.

```
foreach .Operation {
  do ~FromStateCondition>ConditionalTransition {
    statetransition.add(fromstate:"' :Name; 1 '", '
    ' tostate:"'
        do ~FromStateCondition>()~ToStateCondition .Operation {:Name '", '}
    ' transitionoperation:"' :Trigger Operation '", '
    ' conditionaltransition: true,'
    ' operationname1:"' :Operation Name1; '",'
    ' fieldname1:"' :Field Name1; '",'
    ' testvalue1:"' :TestValue1; '",'
    ' conditionoperator:"' :Condition Operator; '",'
    ' operationname2:"' :Operation Name2; '",'
    ' fieldname2:"' :Field Name2; '",'
    ' testvalue2:"' :TestValue2; '",'
    ')' newline
  }
}
```

Figure 4.20. A code snippet of Protocol DSVL code generator

4.4.1.3 Behavior DSVL Code Generator

To run a behavior model, Behavior DSVL code generator must navigate through all the primitive visual constructs in the model and translate them to corresponding Java language statements. The sequence of statements is significant for a procedure language like Java, and the code generator places the code lines of visual constructs following the flows of the directed arcs between them. The first statements are from those visual constructs connected to the entrance bar directly.

In general, Behavior DSVL code generator mainly serves two purposes. One is to define the interdependences among nodes and primitive visual constructs; and another one is to transform the primitive constructs to corresponding Java statements. A visual construct usually has one or more predecessors, and the visual construct can be executed only after all its predecessors have completed their own executions. In addition, the visual construct must have all mandatory input parameters available beforehand. To determine whether a visual construct is ready to run, the code generator must verify these two conditions. Code generators for primitive constructs simply write Java statements or library calls for performing the assigned tasks to them.

We use a boolean function Parameters to explain how Behavior DSVL code generator works. The function verifies the availability of all mandatory input parameters of a visual construct before it can run (see Figure 4.21). The inputs to the function include the arrays of the global variables with assigned values AvailableVariables and the needed parameters NeedParameters to run the visual construct, as well as their sizes. Within the function body, there is a nested loop to check whether each needed parameter can be found in AvailableVariables array by comparing with all the elements of NeedParameters array. If it is found, "true" value is assigned to the corresponding element of a boolean array ReadyArray for holding parameters' assignment status. At the end, Parameters function will return "true", if all ReadyArray elements have been assigned "true" value.

```
'public boolean Parameters(String[] AvailableVariables,
        int AvailableSize,String[] NeedParameters,int NeedSize) { ' newline
'  boolean Ready = true;' newline
'  boolean[] ReadyArray = new boolean[20];' newline newline

'  for (i=0; i<NeedSize; i++) {' newline
'    ReadyArray[i] = false;' newline
'    for (j=0; j<AvailableSize; j++) {' newline
'      if (NeedParameters[i].equals(AvailableVariables[j]))' newline
'        ReadyArray[i] = true;' newline
'      }' newline
'   }' newline newline

'  for (i=0; i<NeedSize; i++) {' newline
'    if (ReadyArray[i] = false)' newline
'      Ready = false;' newline
'  }' newline newline
'  return Ready;' newline
'}' newline newline
```

Figure 4.21. A code generator function to check the availability of input parameters

4.4.2 A Domain Framework and Target Environment

A domain framework acts as an interface between the code transformed from application specific models and the target environment on which the application will run on. A domain framework often serves for four purposes [107]:

- **To Remove Duplications from Generated Code** -- Applications tend to have similar components within a specific domain and yet are not provided by their target environment. To reduce modeling and generator development effort, they can be inserted into the domain framework code;
- **To Provide an Interface for a Code Generator** – A domain framework defines the expected format for code generation output, so that generated code can be seamlessly integrated with the domain framework;
- **To Integrate with Existing Code** -- A domain framework may be used to integrate with existing code, rather than directly calling library services and its interfaces of target environment;
- **To Hide Target Environment and Execution Platform** – A domain framework can be used to support different implementation platforms. Models and generated code can then be the same and the choice of domain frameworks decides the execution platform.

Different from other DSLs, our domain framework also plays another important role in TEE – to provide a network infrastructure to facilitate low-level message exchanges between endpoints and SUTs. Given that it is not our research focus, we have not developed our own but used Apache Axis2 Web service engine instead. Apache Axis2 is one of the most stable and commonly used Web service frameworks. Its core architecture comprises of an XML processing model, a SOAP processing model, a messaging framework and abstractions to implement other aspects like transports and deployment. We use Axis2 Eclipse Plugin to convert an endpoint WSDL file to Web service and package Java code transformed from the endpoint models to provide SIT services to its SUTs.

The use of Axis2 brings some benefits to our endpoint modeling approach: (1) Axis2 facilitates Design by Contract (DbC) programming style [141]. The implementations on both endpoint and SUT sides are bound to a service contract defined by the endpoint

signature WSDL file; (2) Web service can be generated automatically by converting the endpoint WSDL file by using some tools, such as Axis2 Eclipse Plugin; (3) some Axis2 tools allow users to modify its SOAP message headers by adding some QoS attributes, so that we can simulate a variety of business scenarios; and (4) Axis2 is a popular open-source tool, many IT professionals familiar with it.

To implement DbC programming, Axis2 generates linkage codes for both service provider and service client from a signature definition WSDL file. The service provider linkage code takes the form of a service specific implementation skeleton, along with a message receiver class that implements org.apache.axis2.engine.MessageReceiver interface. The service client linkage code is in the form of a stub class, which always extends the Axis2 org.apache.axis2.client.Stub class. Both the service provider skeleton class and client stub class are generated by wsdl2java tool.

The skeleton class defines parameters and data types for the request and/or response messages of all operations provided by a service provider. It acts as the interface for integrating business logic processing classes to receive requests and generate responses from/to its clients. The downside of adding code directly to this class is that if the service interface changes, the skeleton class will be regenerated and all the changes will be overwritten. To avoid this, we have developed a separate implementation class that extends the generated skeleton class, allowing skeleton methods to be overridden without altering the generated code. To make this work, we need to change the generated services.xml service description, replacing the skeleton class name with the implementation class name.

Figure 4.22 lists the skeleton implementation class code, which extends skeleton class PurchaseServiceSkeleton. It re-defines all the eight operations of the endpoint and their parameters, and calls PurchaseServer class to actually process each operation request and generates corresponding response. Figure 4.23 shows how PurchaseServer class works by using Paymentrequest operation method as an example. The method validates the operation request signature by invoking Paymentrequest method of Signature class and checking the returned error code. To validate the protocol correctness, OperationValidation method of Protocol class is invoked by providing the operation name Paymentrequest. After successfully validating both signature and protocol layers, the response parameters of Amount, ErrorCode and ErrorMessage are assigned by calling the

corresponding behavior class PaymentRequest. On the other hand, whenever an error is found during the validation, the request processing will be terminated and an error code and error message will be generated for reporting the request defect.

```
package com.endpoint.purchase;
import com.endpoint.purchase.types.*;

public class PurchaseServiceImpl extends PurchaseServiceSkeleton {
    private PurchaseServer myServer;
    public PurchaseServiceImpl() {
        myServer = new PurchaseServer();
    }
    public ResponseType supplierpo(int supplierpono, String category,
                String item, int quantity) {
        return myServer.Supplierpo(supplierpono, category, item, quantity);
    }
    public ResponseType porequest(int pono, String clientName,
                String category, String item, int quantity) {
        return myServer.Porequest(pono, clientName, category, item, quantity);
    }
    public ResponseType inventorycheck(String category, String item) {
        return myServer.Inventorycheck(category, item);
    }
    public ResponseType approvalnotification(String approver, int supplierpono) {
        return myServer.Approvalnotification(approver, supplierpono);
    }
    public ResponseType supplierpoapproval(int supplierpono) {
        return myServer.Supplierpoapproval(supplierpono);
    }
    public ResponseType supplierdelivery(int supplierpono) {
        return myServer.Supplierdelivery(supplierpono);
    }
    public ResponseType paymentrequest(int pono) {
        return myServer.Paymentrequest(pono);
    }
    public ResponseType deliveryrequest(int pono) {
        return myServer.Deliveryrequest(pono);
    }
}
```

Figure 4.22. An example endpoint skeleton implementation class

Axis2 also provides a stub class for allowing client to access the server operations. We have created a Java API class for each operation to integrate the stub class from endpoint signature model with a SUT. So, our testing environment is suitable for testing all Java applications. Figure 4.24 lists the code of a SUT API class, where the parameters for accessing testing service must be provided from an Ant build file. The required input parameters are checked first. Then, the stub class and response message type are declared and Paymentrequest method is invoked. The response parameters are assigned from the returned values of the stub class.

```
    public synchronized ResponseType Paymentrequest(int pono)
    {
        ResponseType resp = new ResponseType();

        ProtocolError = Protocol.OperationValidation("paymentrequest");

        if (Signature.paymentrequest(pono) == 0)
           if (ProtocolError.ErrorCode == 10)  {
               resp.setErrormessage(ProtocolError.ErrorMessage);
               resp.setErrorcode(ProtocolError.ErrorCode);
           }
           else {
               PaymentrequestResult =
                       MyPaymentrequest.paymentrequest(pono);
               resp.setAmount(PaymentrequestResult.amount);
               resp.setErrorcode(PaymentrequestResult.errorcode);
               resp.setErrormessage(PaymentrequestResult.errormessage);
           }
        else {
           resp.setErrorcode(1);
           resp.setErrormessage("signature parameter value range error.");
        }
        return resp;
    }
```

Figure 4.23. An example operation method of PurchaseServer class

```
package com.endpoint.purchase;
import java.io.IOException;

public class PaymentRequestClient
{
    public static void main(String[] args)
            throws IOException, XMLStreamException {
         // check for required command line parameters
        if (args.length < 4) {
            System.out.println("Usage:\n  java " +
                "com.endpoint.purchase.WebServiceClient protocol host port path");
            System.exit(1);
        }
        // create the client stub
        String target = args[0] + "://" + args[1] + ":" + args[2] + args[3];
        System.out.println("Connecting to " + target);
        PurchaseServiceStub stub = new PurchaseServiceStub(target);
        ResponseType PaymentRequestResponse = new ResponseType();
        PaymentRequestResponse = stub.paymentrequest(40000);
        System.out.println("supplier po error code: " +
                PaymentRequestResponse.getErrorcode() +
                " error message: " + PaymentRequestResponse.getErrormessage() +
                " status: " + PaymentRequestResponse.getDone() +
                " amount: " + PaymentRequestResponse.getAmount());
    }
}
```

Figure 4.24. The code of an operation SUT API class

We use Apache Tomcat 7.0 [21] server as our target environment, and put our Axis2 Web Service engine into it for providing testing service to its SUT. Tomcat implements several Java EE specifications including Java Servlet, JavaServer Pages (JSP), Java EL and WebSocket. It organizes all these parts into a single directory structure and provides a "pure Java" HTTP web server environment in which Java code can run. An API class specifies Tomcat application server URI for a SUT to access the endpoint testing operations through SOAP over HTTP communication protocol.

To automate endpoint generation process, we create an Apache Ant build file. The Ant tool executes the following tasks in sequence automatically: (1) to execute Axis2 wsdl2java command to generate server linkage code from the endpoint WSDL file, (2) to replace implementation class in the deployment descriptor, so that the message receiver will load an instance of our class rather than the generated skeleton, (3) to compile server code for deployment, (4) to package server code as .aar file, and (5) to load the .aar file to Tomcat webapps folder and restart Tomcat.

Figure 4.25 shows a code snippet of the build file to automatically deploy a new Tomcat Web service. This is done in four main steps: (1) to copy all the files located in src folder to bin folder, (2) to package PurchaseService.aar Tomcat service file and create services.xml to specify the Web service configuration, (3) to copy PurchaseService.aar to Tomcat webapps folder, and (4) to re-start Tomcat service for putting the new service in operation.

Figure 4.26 illustrates a deployment view on how an endpoint provides SIT service to its SUT. The left-hand side is the emulated endpoint hosted in a Tomcat application server, its protocol and behavior classes are integrated into Axis2 skeleton class for performing the SUT operation requests validation. The grey areas at the bottom of both sides are Axis2 Web service engine for encoding and decoding SOAP messages exchanged between the endpoint and the SUT. The SUT is located on the right-hand side at the top, communicating with Axis2 Stub class through an API class. The SUT invokes the endpoint service through accessing Tomcat Axis2 service URL using SOAP over HTTP application protocol.

```
<!-- Package server code as .aar file -->
<target name="package">
    <delete quiet="true" file="${service-archive-name}.aar"/>
    <copy todir="${build-server}/bin">
        <fileset dir="${build-server}/src" includes="service.*"/>
    </copy>
    <jar jarfile="${service-archive-name}.aar" basedir="${build-server}/bin">
        <metainf file="resources/services.xml"/>
    </jar>
    <delete quiet="true" file=
        "/java/tomcat 7.0/webapps/axis2/WEB-INF/services/PurchaseService.aar"/>
    <copyfile src="PurchaseService.aar" dest=
        "/java/tomcat 7.0/webapps/axis2/WEB-INF/services/PurchaseService.aar"/>
    <exec executable="cmd" dir="C:/java/Tomcat 7.0/bin">
        <arg value="/C"/>
        <arg value="shutdown.bat"/>
    </exec>
    <exec executable="cmd" dir="C:/java/Tomcat 7.0/bin">
        <arg value="/C"/>
        <arg value="startup.bat"/>
    </exec>
</target>
```

Figure 4.25. A code snippet of Ant build file for deploying Tomcat Web service

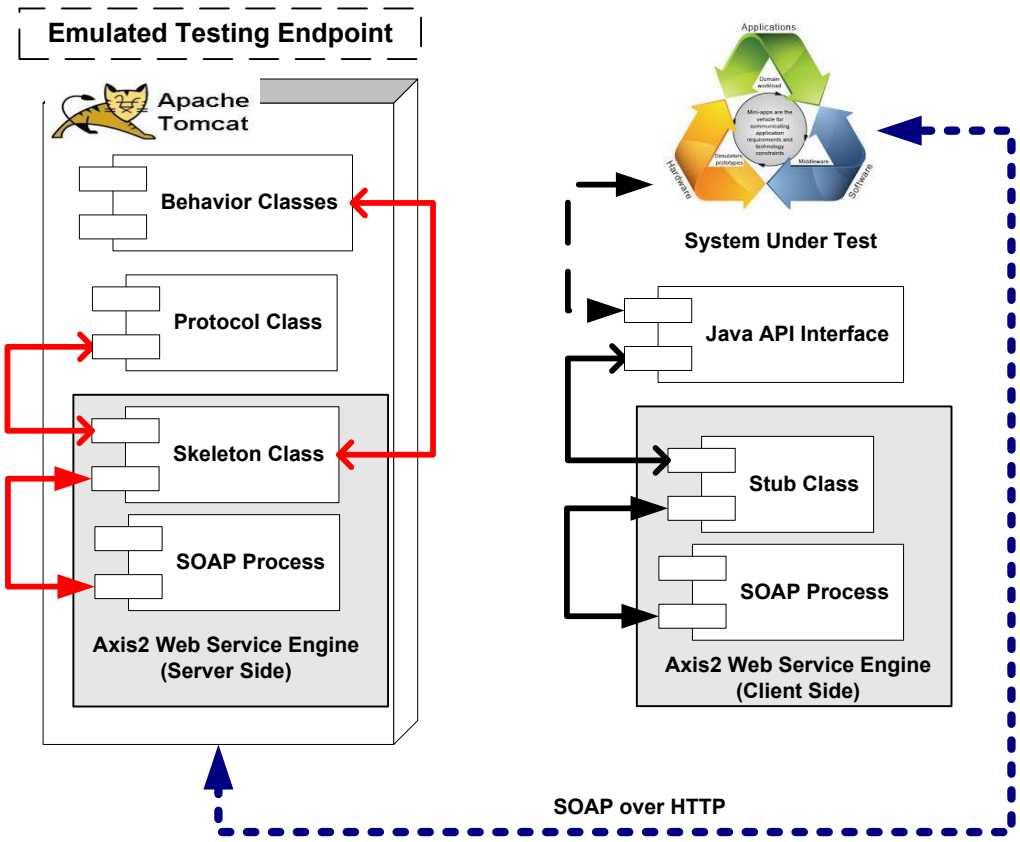


Figure 4.26. The deployment view of an endpoint and its SUT

4.5 Metamodeling Language

A metamodeling language is used to design the programming constructs and specify the syntax of domain-specific languages. It also provides a supporting toolset for users to create code generators to transform models. A good metamodeling language should guide its users during language definition and let them focus on language definition by hiding implementation details.

4.5.1 Metamodeling Language Selection

There are many DSL development tools either commercial products or open-source freeware, and all of them promise significant development productivity gain and deliverable quality improvement. However, it is difficult for most DSL developers to compare them and know under which conditions a tool is more suitable than others. To help DSL developers out of the difficulty, Amyot et al. proposed a systemic comparison framework with six evaluation criteria [142] as defined below:

- **Graphical Completeness** – the tool is able to represent all the notation elements used in DSLs;
- **Editor Usability** -- the editor supports undo/redo, load/save, simple manipulation of notation elements and properties, etc.;
- **Effort** – the tool is easy to learn and use to develop DSLs;
- **Language Evolution** – the tool is able to support older models developed before the tool evolves;
- **Integration with Other Languages** -- the tool is able to support additional languages or integrate with other tools;
- **Analysis Capabilities** – the models created by the graphical editor can be easily analysed or transformed?

They applied these criteria to evaluate five popular metamodeling tools:

- **Generic Modeling Environment (GME)** [143] -- a configurable framework developed at Vanderbilt University and used to create domain-specific modeling environments;

- **Telelogic Tau G2** [144] -- a model-driven development environment that supports UML 2.0;
- **Rational Software Architect (RSA)** [145] -- a UML 2.0 compliant integrated software development environment, built on the top of Eclipse platform;
- **Xactium XMF-Mosaic** [146] -- an integrated, Eclipse-based and extensible development environment for domain-specific language development;
- **Eclipse Modeling Framework (EMF)** [19] -- a framework and code generation facility for building tools and other applications based on a structured data model.

Other than these tools, we add one more popular metamodeling tool MetaEdit+ to this evaluation [18]. The six metamodeling tools are evaluated against the six evaluation criteria, and the summary results are given in Table 4.8.

From the evaluation results, we can see that the best choice is MetaEdit+ and followed by EMF. In addition to the superiorities on the technical aspects listed in the table, MetaCase provides exceptional support for whatever problems you may meet during your DSL development. The minor drawback is a small amount license fee for use the tool for academic purpose.

4.5.2 Metamodeling Language and Toolset Chosen -- MetaEdit+

MetaEdit+ is one of the world's leading DSM tools for automating full cycle DSM software development process. MetaEdit+ uses GOPPRR (Graph, Object, Property, Port, Role and Relationship) metamodeling framework for allowing developers to create DSL solutions for any business domains. MetaEdit+ CASE toolset includes a diagram editor, object and graph browsers, and property dialogs for users to create visual constructs. Using these tools, language concepts, their properties, relationships between concepts, associated rules and symbols can be defined easily.

MetaEdit+ graphical tool provides functionality to access, view and modify modeling language specifications at graph level. Figure 4.27 shows a screenshot of MetaEdit+ graph tool. The graph tool contains five tabs for specifying different aspects of a graph type (i.e. a DSL): (1) Basics for entering the basic information about the graph type itself, (2) Types for specifying the object, relationship and role types of the graph type, (3) Bindings for specifying how relationships, roles, ports and objects are connected to each

Table 4.8. Evaluation summary results of metamodeling tools[6]

Evaluation Criteria	**GME** [143]	**Tau G2** [144]	**RSA** [145]	**XMF-Mosaic** [146]	**EMF** [19]	**MetaEdit+** [18]
Graphical Completeness	Medium	Low	Very Low	Low	High	High
Editor Usability	Medium	Medium	Low	Low	Very High	Very High
Effortlessness	Medium	Low	High	Low	Very Low	High
Language Evolution	High	?	?	?	Medium	High
Integration	Low	High	High	Low	High	Medium
Analysis/Transformation	Medium	Medium	Low	High	Medium	Medium

[6] The first five tools' ratings were given by Amyot et al., and we provided the last tool ratings based on our use experience through this research project.

other, (4) Subgraphs for defining the decomposition and explosion links for the graph type, and (5) Constraints for defining the constraints on relationship, role, port and object combinations.

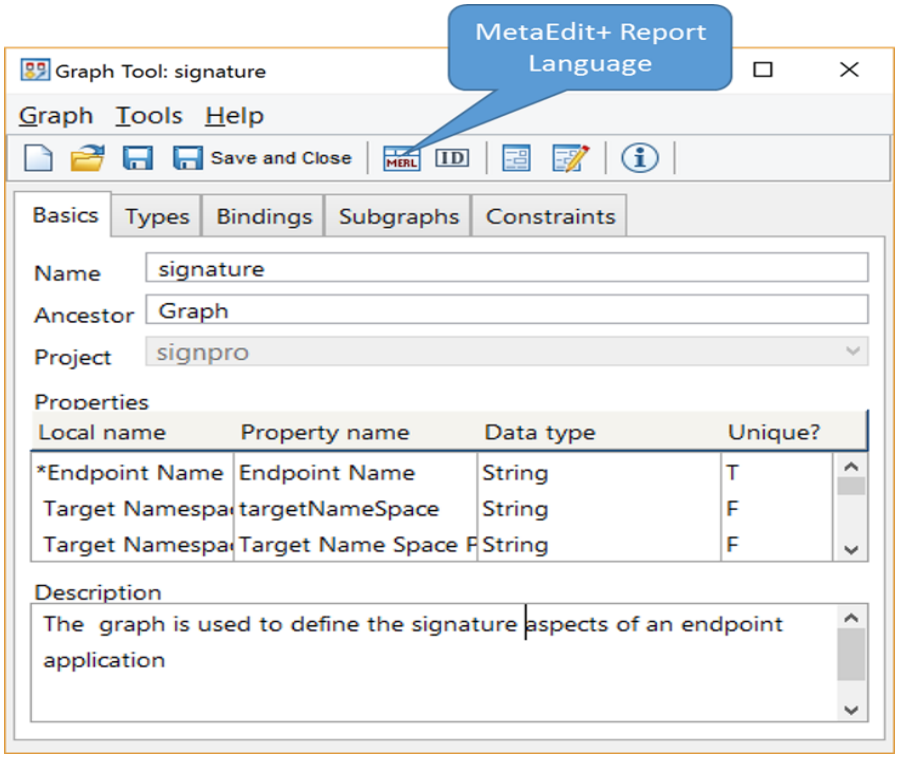


Figure 4.27. MetaEdit+ 5.1 graph tool

To design a symbol's visual representation, MetaEdit+ provides a symbol editor drawing tool (see Figure 4.28) for creating or modifying the graphical symbol of an object, role or relationship type as displayed in diagrams (also called models). The symbols are made using variety of shapes, colours and textual fields. Each graphical symbol can have a condition attached to it. The condition depends on the value of a property or the output of a generator and determines whether to draw the symbol. Object and relationship symbols can specify connectable areas that specifies how and where role lines are attached to them.

MetaEdit+ provides a code generator facility called MetaEdit+ Report Language (MERL), which uses a DSL to specify how to walk through models and output their contents along with other text. Code generators, developed by MERL, can produce code or configuration information, create documentation and data dictionaries, check the consistency of models, analyse model linkages, and export models to other programs. MERL editor (see Figure 4.29) includes four main components: (1) Generator Box at the left below menu bar lists the available generators for a graph type; (2) Concept Box in the middle allows users to

select what to view in the right list box; (3) Choice Box at the right provides a list of entities based on the choice in Concept Box; and (4) Main Editing Area at the lower part shows the currently selected generator for users to view and edit.

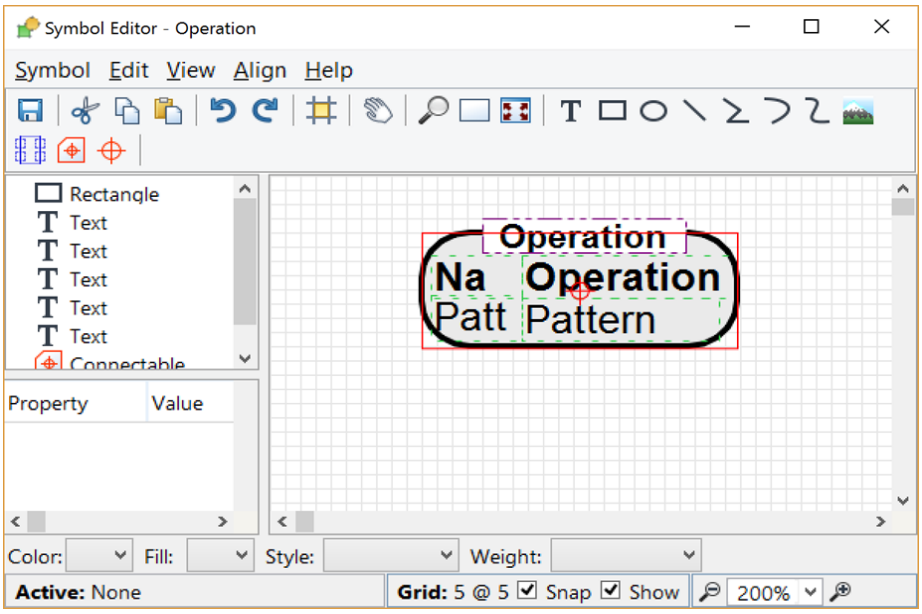


Figure 4.28. MetaEdit+ 5.1 symbol editor

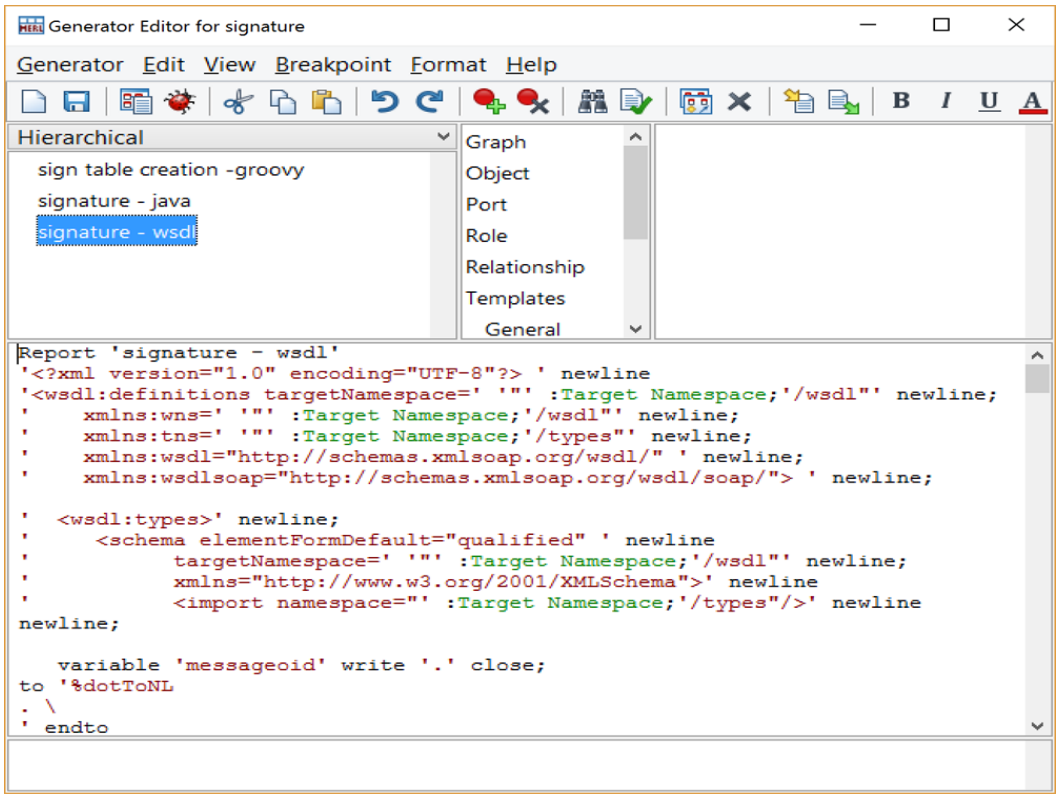


Figure 4.29. MetaEdit+ 5.1 code generator editor

4.6 Summary

Domain-specific language development is hard. Developers must have the expertise on both business domain knowledge and language development skills. The scope and requirements for a new DSL are often vague and unstable. To meet these DSL development challenges, we have laid down a development guideline, complied with a well-established development process and selected the best suitable meta-modeling tool. Visual languages let users create programs by manipulating programming constructs graphically rather than by specifying them textually. They are generally easier to learn and use than textual languages without much difference in expressivity. Targeting domain experts, we have decided to use visual languages to model endpoints.

The visual representations of a visual language have a profound effect on its usability and effectiveness. With a clear design goal in mind to maximize cognitive effectiveness, we use a systematic approach to evaluate, compare, construct and optimize our DSVL's visual constructs. Our design approach is based on Moody's Physics of Notations and eight out of the nine principles are applied. To prevent modeling errors from users' mistakes, we define many domain rules to restrict some kinds of illegal use of visual constructs, their properties and the relationships among these visual constructs.

To make a DSM solution easy to use and have a high development productivity, models are the only artefacts to be manipulated by users and other tasks should be implemented automatically. We have achieved the full code generation from models, the code does not need inspection, manual rewriting or additions after generation. Building operational endpoint from generated code is a tedious work and requires users to have a certain level of technical skills. It may not be able to handle by non-technical background domain experts. Therefore, we create a supporting toolset for generating endpoints automatically.

In the next chapter, we use our TeeVML to emulate the functional layers of a typical business application example. Specifically, we provide a stepwise demonstration to model the endpoint and show how well the three research questions raised in Chapter 1 have been addressed by our approach.

CHAPTER 5

Case Study - Functional Layer Modeling

In this chapter, we use the motivating example introduced in Chapter 1 as a case study to demonstrate how our endpoint functional layers can be modeled by using our TeeVML. We also describe the steps to convert these functional layer models to executable forms and integrate them to our selected domain framework and target environment. The purposes of this case study are two-fold. First, a stepwise instruction is provided for the use of our TeeVML and supporting toolset. Second, our endpoint modeling process is demonstrated by going through a typical endpoint example.

5.1 Case Study

In this chapter, we use an endpoint application called PeopleSoft Finance ERP system, which supports a new public cloud CRM salesforce.com application as its system under test introduced in Chapter 1. The endpoint protocol behavior is described in the activity sequence diagram of Figure 1.1, details of which were provided in Chapter 1. Here we focus on the other two functional layers – signature and behavior.

5.1.1 Example Signature Layer

Our endpoint signature is based on Remote Procedure Call (RPC) communication style. RPC is a request and response communication protocol, and communication is initiated by a client sending an operation request to a known remote server for executing a specified operation with supplied parameters [23]. If the operation has an “in-out” pattern, the remote server will send a response message back to the client after processing the operation request. The request and response parameters can be of string, integer, float, boolean or date data types; and they may be either mandatory or optional. There are total of 10 operations provided by the ERP endpoint system for the CRM application. Table 5.1 lists the details of these operations, including their names, parameters, parameter data types and value ranges. The “No” column represents the operation sequence for the

typical sales process specified in Figure 1.1. To simplify the endpoint modeling, we use a standardised response message format with four elements.

Table 5.1. The example signature definition

No	Operation	Direction	Parameter	Data Type	Value Range
1	logon	Request	inputuserid	Integer	10000 to 99999
			inputusername	String	
			inputpassword	String	
2	porequest	Request	pono	Integer	10000 to 99999
			clientname	String	
			category	String	
			item	String	
			quantity	Integer	0 to 99999
3	Inventorycheck	Request	category	String	
			item	String	
4	supplierpo	Request	supplierpono	Integer	10000 to 99999
			category	String	
			item	String	
			quantity	Integer	0 to 99999
			requiredlevel	Integer	0 to 10
5	supplierpoapproval	Request	supplierpono	Integer	10000 to 99999
6	approvalnotification	Request	approver	String	
			supplierpono	Integer	10000 to 99999
7	supplierdelivery	Request	supplierpono	Integer	10000 to 99999
8	paymentrequest	Request	pono	Integer	10000 to 99999
9	deliveryrequest	Request	pono	Integer	10000 to 99999
10	logout	Request	inputuserid	Integer	10000 to 99999
			inputusername	String	
			inputpassword	String	
	All operations	Response	errorcode	Integer	
			errormessage	String	
			done	Boolean	

			amount	Float	

5.1.2 Example Behavior Layer

The operation request from a SUT is firstly validated for its correctness of syntax and temporal sequence by endpoint signature and protocol layers. Then, it is handed over to behavior layer for logic processing and response generation. Not only does behavior layer modeling need to process the parameters of operation's request, but it also accesses persistent data. For handling a purchase process, the ERP endpoint system needs to have six tables to store user, product, purchase order, client and supplier information. These tables and their relationships are depicted by an entity relationship diagram in Figure 5.1. Every staff has a user account, and he/she can place multiple purchase orders. A purchase order is for a client and may contain multiple products. A supplier delivers one or more products of a purchase order.

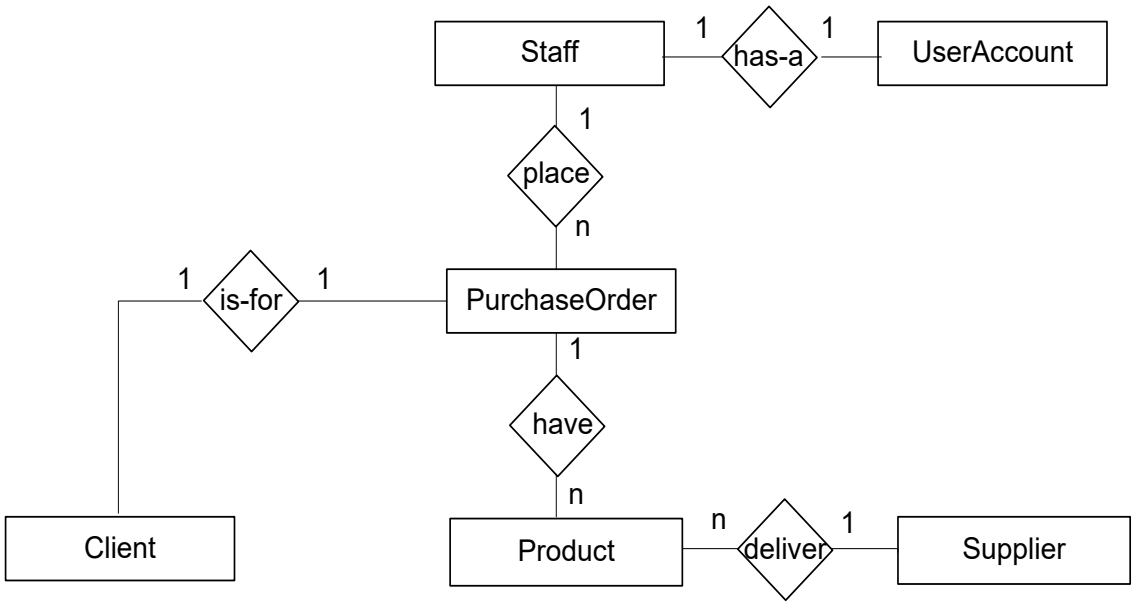


Figure 5.1. The entity relationship diagram of the ERP endpoint persistent data tables

The details of the tables are listed in Table 5.2, including all the fields and their properties. Specifically, Staff table is a slave table of UserAccount, since it has a reference key "username", which is a field of UserAccount table.

After defining the ERP system signature layer and persistent data tables, we briefly describe its behaviors to process operation requests.

Table 5.2. The ERP endpoint persistent data tables and fields

Table	Field	Data Type	Mandatory	Default	Key
Client	clientname	varchar(20)	True		Primary key
	contact	varchar(20)	True		
	address	text	True		
	email	varchar(40)	False		
	discount	float	False	0	
Supplier	suppliername	varchar(20)	True		Primary key
	contact	varchar(20)	True		
	address	text	False		
	email	varchar(40)	False		
	product	varchar(20)	True		
	item	varchar(20)	True		
Product	category	varchar(20)	True		Primary key
	item	varchar(20)	True		Primary key
	inventory	int	True	0	
	unitprice	float	False	0	
	wholesaleprice	float	False	0	
UserAccount	userid	int	True		Primary key
	username	varchar(20)	True		
	password	varchar(20)	True		
	insession	int	False	0	
	failures	int	False	0	
Purchase Order	pono	int	True		Primary key
	category	varchar(20)	True		
	item	varchar(20)	True		
	quantity	int	True	0	
	status	varchar(20)	True	“open”	
	client	varchar(20)	False		
	supplier	varchar(20)	False		
	type	varchar(20)	True	“clientpo”	
	requiredlevel	int	False	0	

	approvedlevel	int	False	0	
Staff	username	varchar(20)	True		Reference key
	department	varchar(20)	True		
	role	varchar(20)	True		
	email	varchar(40)	False		
	approvallimit	float	False	0	

To start an interactive session, the CRM sends a “logon” request to the ERP. The user initiating the request will be authenticated, provided that a match for the combination of “userid”, “username” and “password” parameters is found in UserAccount table. A purchase order (PO) is placed by sending a “porequest” request. The request is only valid and a record will be inserted to PurchaseOrder table, if its “category” and “item” parameters are found in Product table and “clientname” is found in Client table. Following the “porequest” request, an “inventorycheck” request is sent to the ERP for checking the current stock level of the PO product by the “category” and “item” parameters. If a matching record is found in Product table, the product inventory will be retrieved and returned to the CRM in the corresponding response message.

After the CRM receives the “inventorycheck” operation response, the next operation request will be decided depending on whether the current product inventory is enough to fulfil the PO. If the inventory is not enough, the CRM must send a “supplierpo” request to the ERP for purchasing the missing quantity of the PO product from a supplier. The supplier PO must get approval internally first, and a “supplierpoapproval” request is sent to the ERP. The ERP determines who need to approve the supplier PO based on the requester’s department and role in Staff table. Therefore, an “approvalnotification” request is sent to ask the requester’s manager to give his/her approval. Sometimes, such “approvalnotification” request may need to be sent to more than one managers from lower level to higher level until all the required approvals are obtained. The last step of supplier purchase is to inform the supplier to deliver the purchased product by sending a “supplierdelivery” request.

After conforming enough product inventory for the PO, a “paymentrequest” request is sent to the ERP for calculating the PO amount. It is calculated by multiplying the PO quantity of the “porequest” and “unitprice” in Product table, deducting “discount” in

Client table, and adding a standard 10% tax. Following the "paymentrequest", a "deliveryrequest" request is sent to inform the PO delivery to the client. The last step of the PO process is a "logout" request, which marks the completion of the PO process by setting the "status" field of PurchaseOrder table to "close' and the "insession" field of UserAccount table to "0". Figure 5.2 illustrates the supplier purchase process using activity diagram.

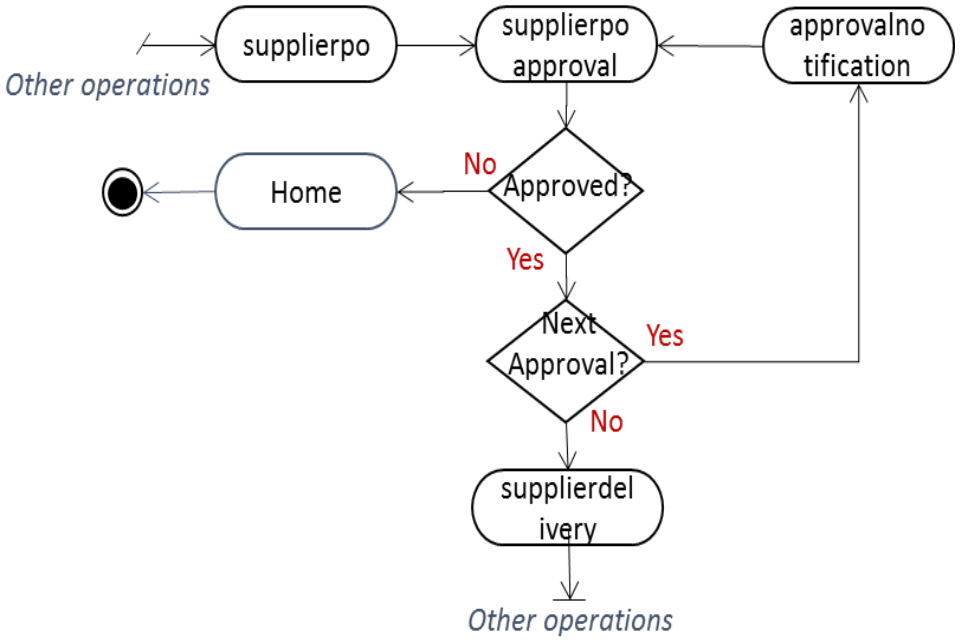


Figure 5.2. The activity diagram of supplier purchase process

5.2 Endpoint Modeling

We use the diagram editor of MetaEdit+ to model the ERP endpoint. The diagram editor is a tool for creating, managing and maintaining models as diagrams. Using the diagram editor, we can view and edit models as well as make or view explosions and decompositions between models at different levels. The main components of the diagram editor include a main diagram drawing area at the centre, a visual construct icon bar at the upper left-hand corner, a sidebar tree view below the visual construct icon bar, and a sidebar property sheet at the lower left-hand corner. The visual constructs of a selective DSVL we have developed are all available at the visual construct icon bar. Figure 5.3 shows the diagram editor.

A diagram is drawn by selecting visual constructs at the icon bar and dragging-and-dropping them to the main drawing area. Normally, a visual construct has a pop-up dialog box for users to provide property information to instantiate the entity.

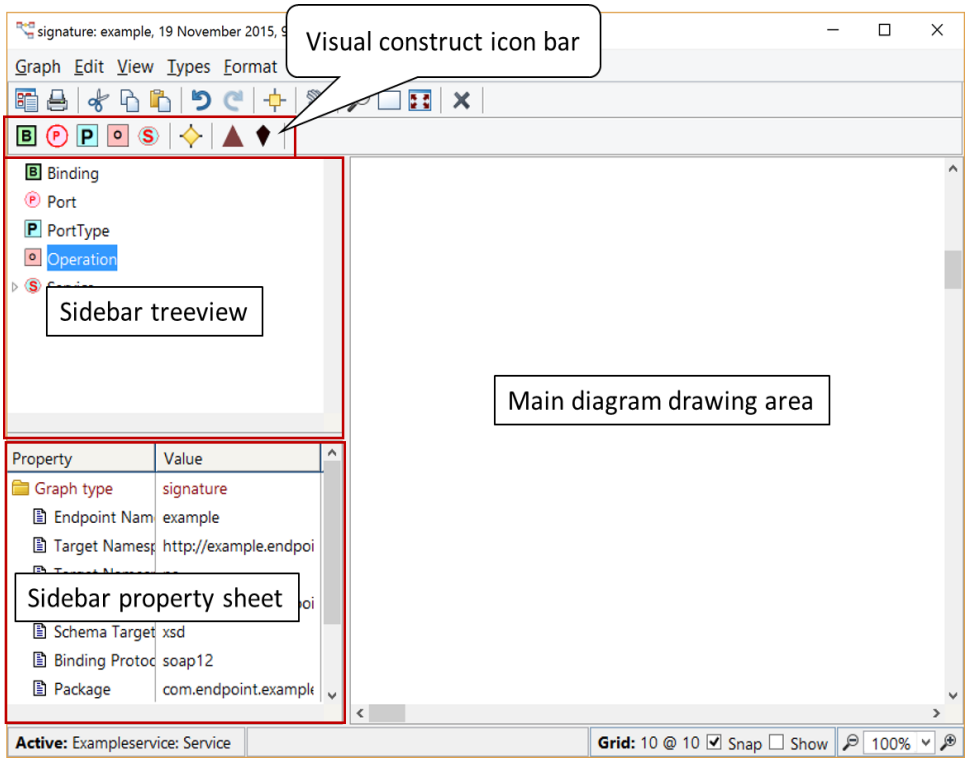


Figure 5.3 MetaEdit+ 5.1 diagram editor

5.2.1 Signature Modeling

We use our Signature DSVL to model the endpoint signature layer. Signature modeling starts from specifying endpoint level properties, including endpoint name, Java package, target namespace, URI for the provided service, database location, and user name and password. We enter all these properties in the endpoint signature model dialog box shown by Figure 5.4, which will pop up when starting a new endpoint signature modeling.

The next step is to define the five WSDL entity types in the main signature model: Service, Port, Binding, Porttype and Operation. The ERP endpoint contains 10 instances of Operation and one instance for each of the other entities. To instantiate Service, Port, Binding and Porttype, we assign each instance name by combining the word “Purchase” and the entity type. Apart from the name, each Operation instance is also assigned “in/out” to its pattern property, except for “logout” request that is “in-only”. Port instance “address” property is “purchase.endpoint.com”.

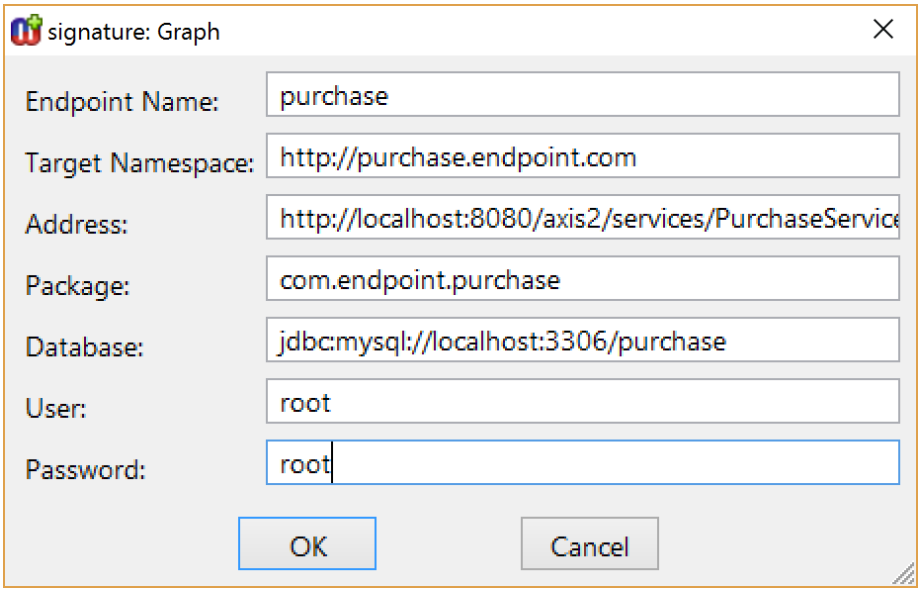


Figure 5.4. The example endpoint signature model dialog box

We use composition and association relationships to link them together. The relationships between these entity instances are: (1) a Service instance uses a composition relationship to link a Port instance; (2) a Binding instance uses two association relationships to link a Port instance and a Porttype instance; and (3) a Porttype instance uses composition relationships to link one or more Operation instances. Figure 5.5 shows the example WSDL model. The "PurchaseService" service is provided by the endpoint through "purchase.endpoint.com" address, specified by Service and Port instances respectively. The service includes 10 operations, mostly using request and response communication style.

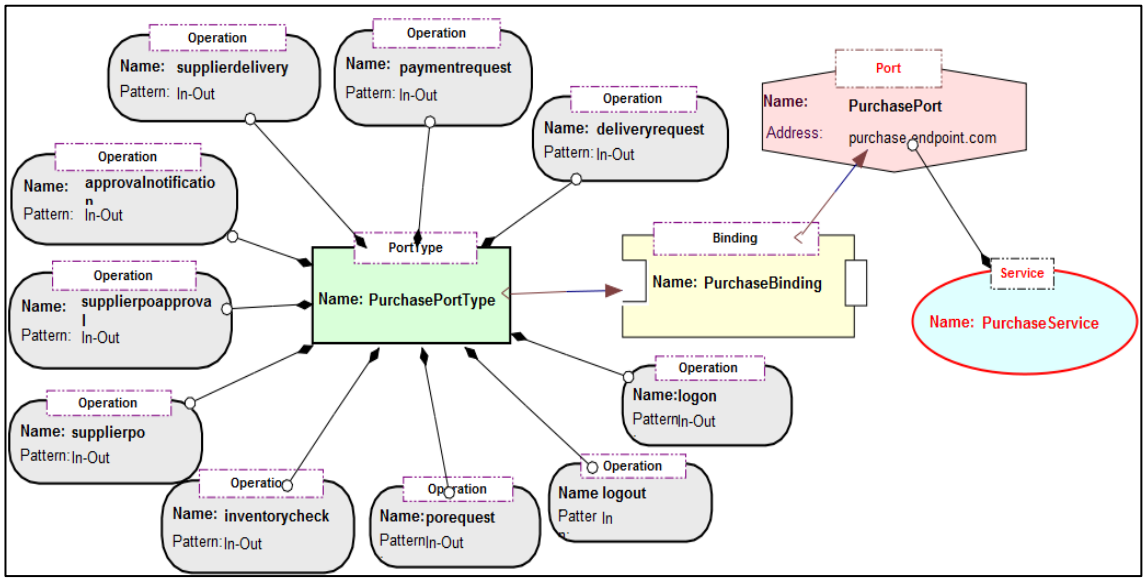


Figure 5.5. The example endpoint signature WSDL model

We take “paymentrequest” operation as an example to show how an endpoint operation request and response parameters are specified. To define “paymentrequest” operation, we need to decompose the operation by opening its operation model. The operation model consists of two message instances with assigned name property as “paymentrequest_request” and “paymentrequest_response” respectively. For their label property, the former is assigned “in” and the latter “out”. Figure 5.6 shows the request and response messages in “paymentrequest” operation.

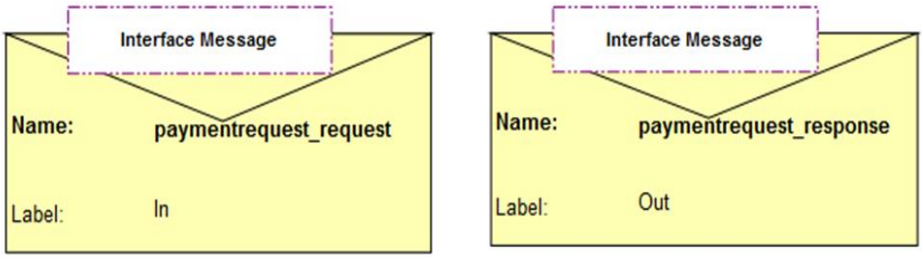
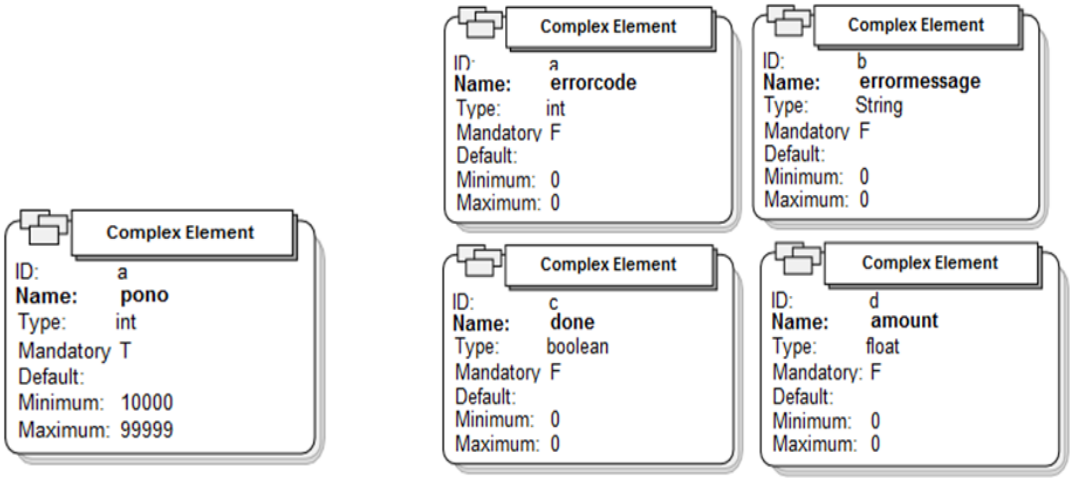


Figure 5.6. The example endpoint “paymentrequest” Operation instance

Message elements are defined by using Message DSVL to decompose request and/or response message(s) in an operation. The message of “paymentrequest_request” contains only one element “pono” (purchase order number). Its data type is defined as integer by selecting the corresponding value from the type property drop-down list. This element is mandatory, specified by selecting the mandatory property checkbox. Since a valid “pono” is a five-digit integer, the element’s minimum property is specified as 10000 and maximum property as 99999. The response message consists of four elements: “errorcode” “errormessage” “done” and “amount”. These are placed in the message by their ID property in alphabetic order. The data types for these elements are “errorcode” integer, “errormessage” string, “done” boolean and “amount” float. Figure 5.7 shows these elements’ definition in the request and response messages as it appears in the diagram editor.

5.2.2 Protocol Modeling

We use our Protocol DSVL to model the ERP endpoint protocol layer. Figure 5.8 illustrates the endpoint protocol model, where the emulated enterprise purchase process flows in clockwise direction. To explain how the endpoint protocol is modeled, we select three typical protocol behaviors. These are interactive session management, constraint state transition and transition iteration, marked as A, B and C in the diagram of Figure 5.8, respectively.



[a] Request message element [b] Response message elements

Figure 5.7. The example endpoint "paymentrequest" Message instances

A - Session Management: Endpoint protocol modeling always starts by specifying an interactive session. A session is managed by using an Idle and a Home state instances. We use a "logon" transition relationship to link the idle state to home state. With "logon" transition the endpoint state changes from idle to home, making it ready to receive operation requests. For the opposite direction, a "logout" transition relationship terminates a session and moves the endpoint state back to idle. A session can also be terminated by a timeout relationship, where the time in seconds can be specified by its dialog box.

B – Constraint Transition Relationship: Sometimes, state transitions are subject to certain runtime constraints. We use constraint transition relationships to model these runtime protocol behaviors. When the endpoint is at "inventorycheck" state, there are alternative process flows either to "supplierpo" or "paymentrequest". The choice between the alternative flows is subject to whether the product inventory can meet the PO quantity requirement of "porequest" operation request. Figure 5.9 shows the constraint condition definition dialog box, which specifies the condition for the transition from "inventorycheck" state to "supplierpo" state. The state transition is subject to the condition of that the "quantity" parameter of "porequest" request must be greater than "inventory" parameter of "inventorycheck" response.

C – Transition iteration: A loop relationship is used to model iteration over one or more operations. This is done by linking a "from" state to a "to" state using a loop relationship.

All states between the "from" and "to" states will be executed repeatedly. The endpoint uses a loop relationship to define the approval process of a supplier PO, which includes "supplierpoapproval" and "approvalnotification" operations. The approval process starts from the immediate manager of the purchaser until the manager with authority for the PO's amount.

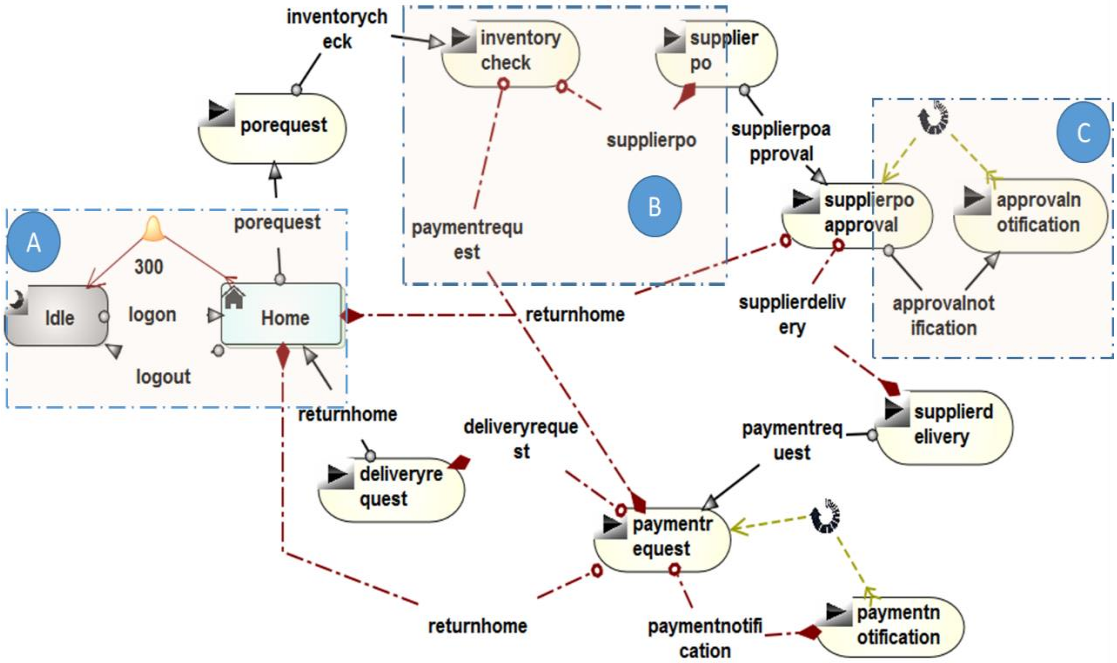


Figure 5.8. The example endpoint protocol model

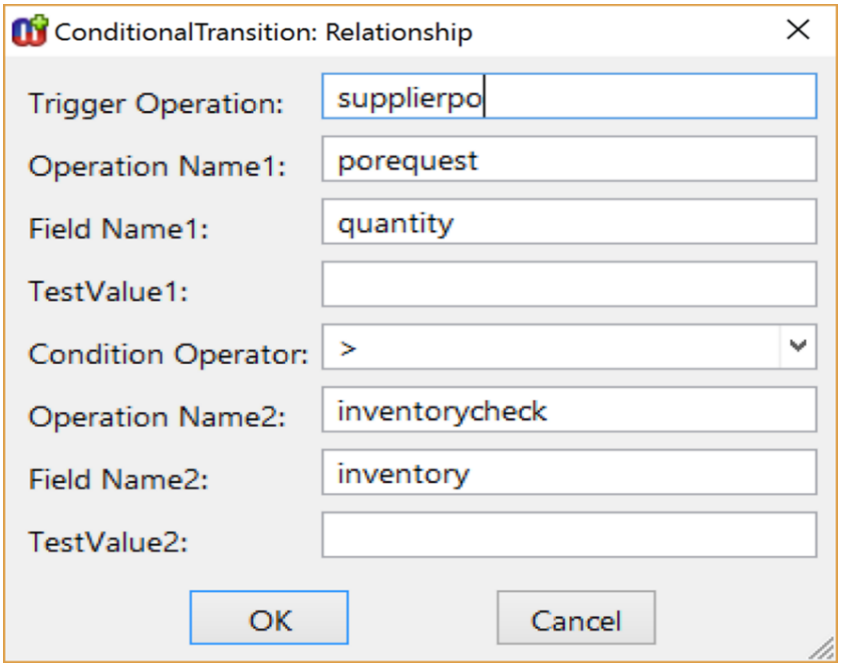


Figure 5.9. The example endpoint protocol model constraint condition definition

All the endpoint operations are in synchronous mode and "paymentrequest" is an unsafe operation. Synchronous operations are simulated by hypothetically providing a processing time. When an endpoint is handling a synchronous operation, any operation requests will be rejected. Similarly, unsafe operations are simulated by assuming an operation request transmission time, the following requests of the same operation will be rejected when the first request is in transmission. We can use state entity dialog box to simulate an operation with different business scenarios shown by Figure 5.10. As "paymentrequest" operation is in synchronous mode, we give its process time five second. It is also an unsafe operation and its transmission time is set for 15 seconds.

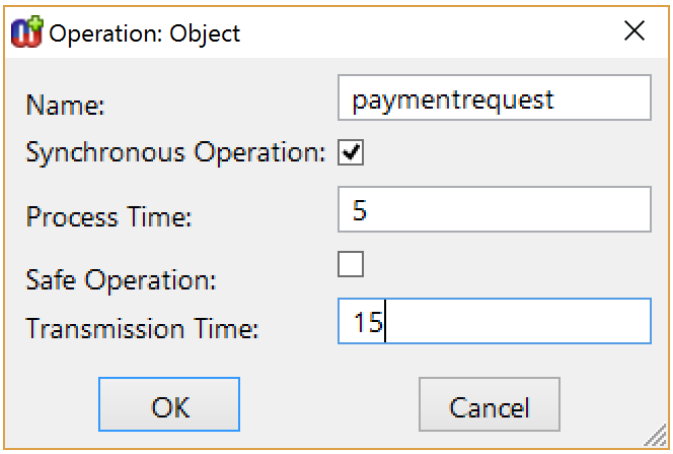


Figure 5.10. The example endpoint protocol model business scenarios simulation

5.2.3 Behavior Modeling

The top-level endpoint behavior model consists of the instances of Service Node and Data Store. Service nodes import their input and output parameters from the endpoint signature model; and data stores are created by reusing signature Message DSVL. For the ERP endpoint, we create 10 service nodes and six data stores shown in Figure 5.11 (a slave table Staff is defined inside its master table UserAccount). To reduce the diagram complexity, these service nodes can also be collapsed to show their name only and hide all input and output parameters.

We use one operation example "paymentrequest" to show how an endpoint behavior is modeled. Figure 5.12a shows "paymentrequest" operation service node, which consists of two nodes: "poinformationretrieve" to retrieve the PO information from the tables and "poamountcalculation" to work out the total PO amount. These two nodes are placed between an pair of entrance and exit bars.

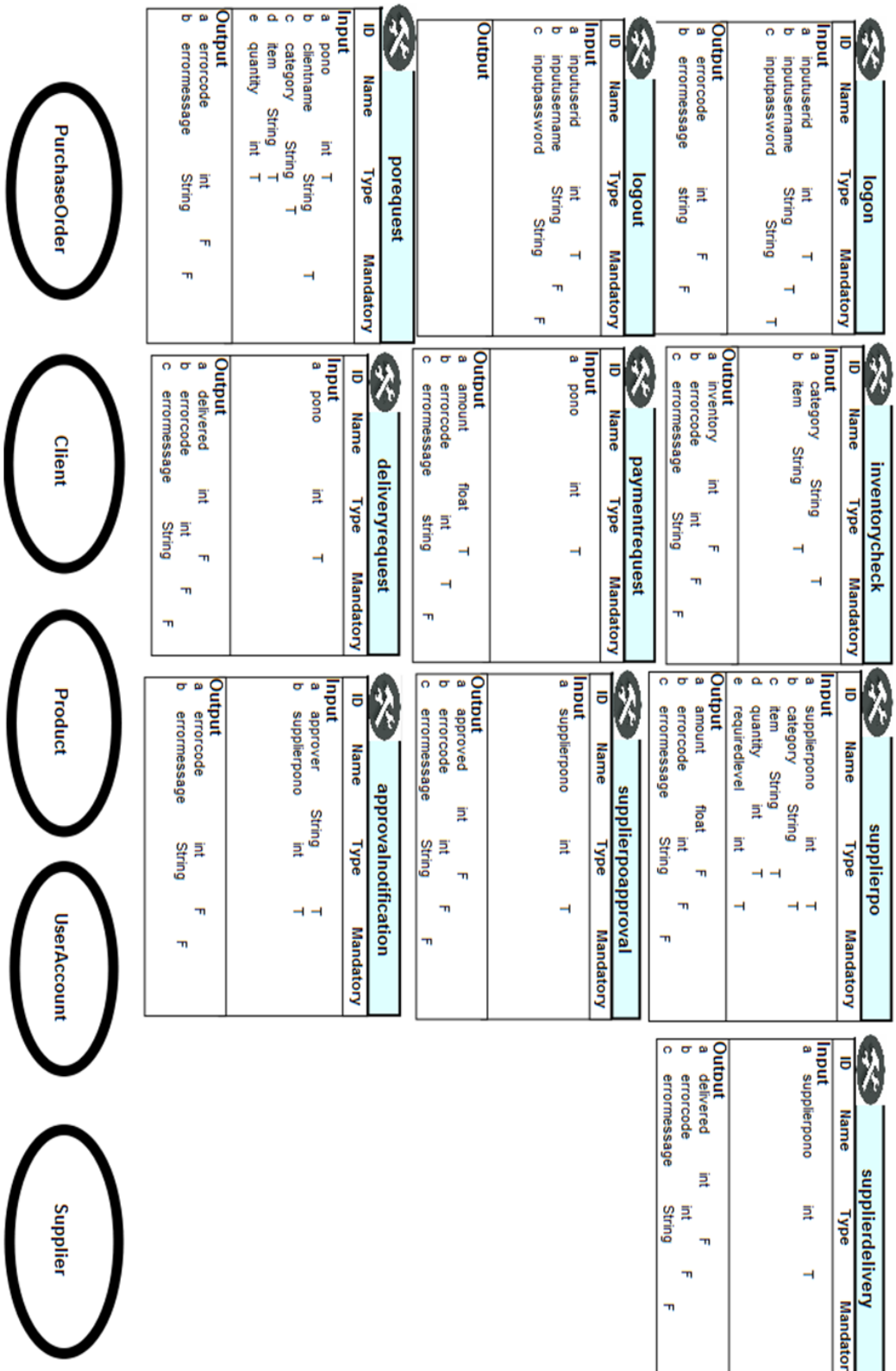


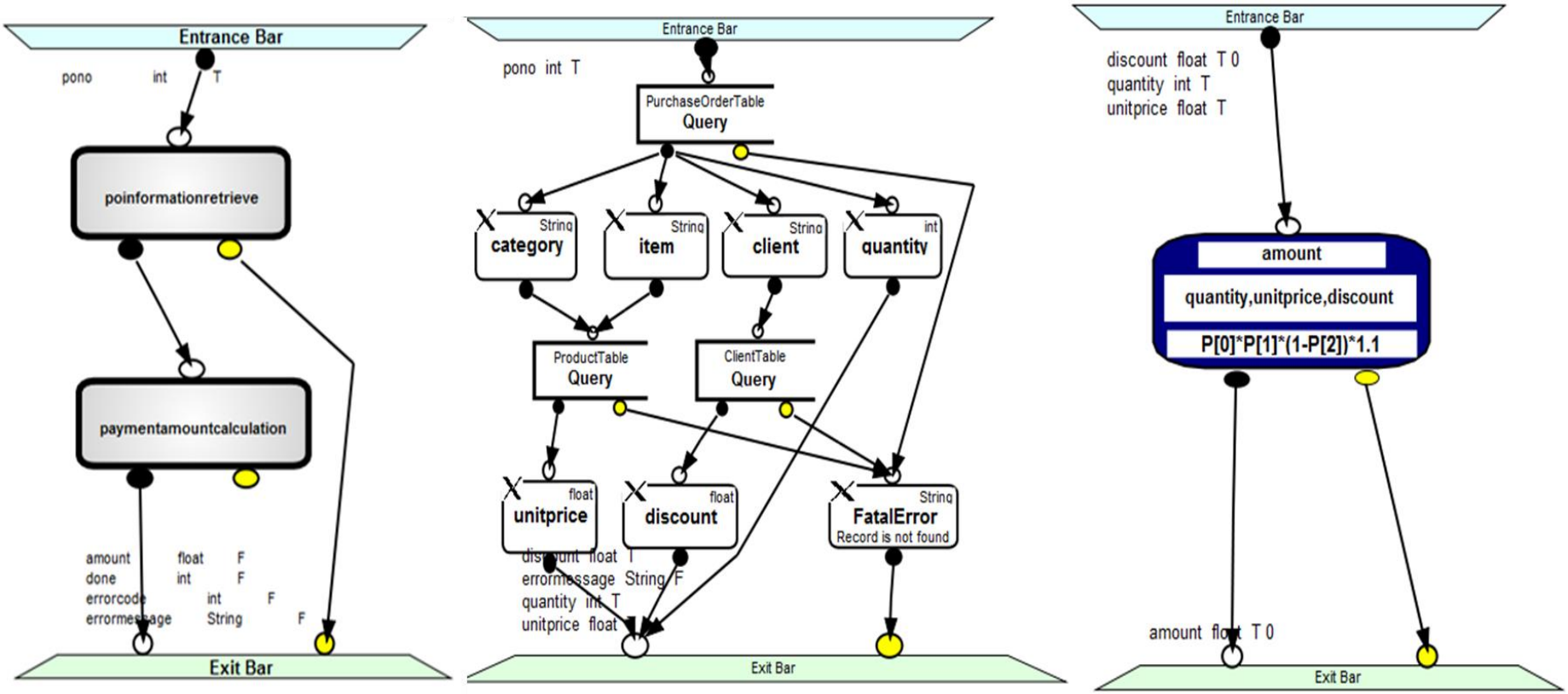
Figure 5.11. The example endpoint top level behavior model

Figure 5.12b illustrates “poinformationretrieve” node internal structure and dataflows. The node has one input parameter “pono”, and four output parameters -- “quantity” and “unitprice” of the PO item, client “discount” and “errormessage”. These parameters are specified by use of the node entrance and exit bars. Figure 5.13a is the dialog box of the node entrance bar, showing only “pono” parameter in it. To define the parameter’s properties, doubly click the parameter to bring up its definition dialog box as shown by Figure 5.13b. Particularly, the parameter is mandatory, which means that the node will generate exceptional output if it is missing.

As shown by Figure 5.12b, the node has three data query operations: (1) to retrieve the PO “category”, “item”, “quantity” and “client” from PurchaseOrder table by “pono”; (2) to retrieve the item “unitprice” from Product table by “category” and “item”; and (3) to retrieve “discount” from Client table by “client”.

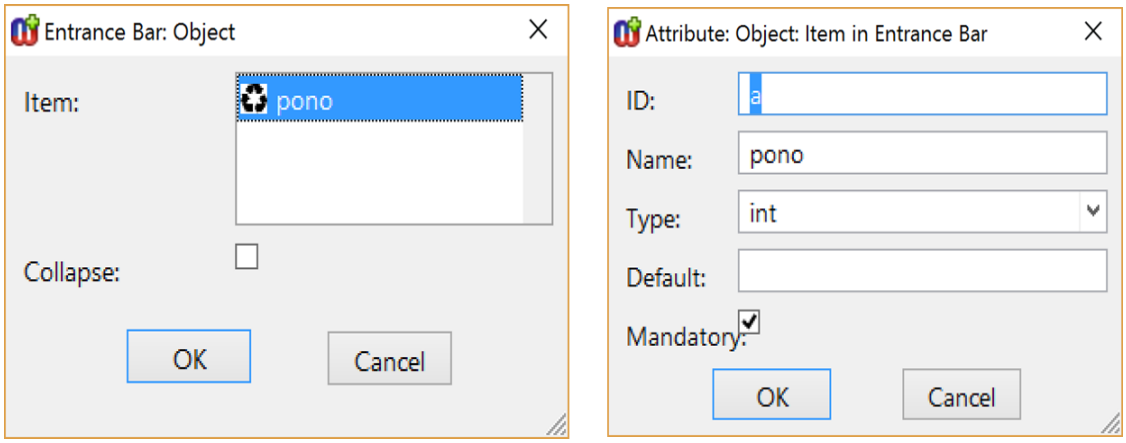
As an example to show how to define database operations, Figure 5.14 presents the first query operation by using the dialog box of JDBC operator. The data operation “Query” is selected from a drop-down list, and PurchaseOrderTable table must exist in the current database. The condition for the query operation is specified by a table field name “Pono” (with a condition operator) matching with the input parameter “pono”. If a matching record is found with the search condition in PurchaseOrderTable, the fields of “Category”, “Item” and “Quantity” are retrieved from the table. Otherwise, “errormessage” variable will be assigned the value of “Record is not found” and placed on “exceptional out” port of the exit bar.

The “paymentamountcalculation” node only contains an evaluator to calculate the PO “amount” as shown by Figure 5.12c. The node takes the output parameters of “poinformationretrieve” node as its input parameters to the evaluator. It has a variable name “amount” for holding the result at the top line, the parameters of “quantity”, “unitprice” and “discount” at the middle line, and the formula of “P[0]*P[1]*(1-P[2])*1.1” at the bottom line. An exception can occur, if any of the input parameters to the node is missing or their data types do not match with what have been defined in the entrance bar.



[a] Paymentrequest service node [b] poinformationretrieve node [c] paymentamountcalculation node

Figure 5.12. The example "paymentrequest" behavior model



[a] Dialog box for parameter addition [b] Dialog box for parameter definition

Figure 5.13. The entrance bar definition of "poinformationretrieve" node

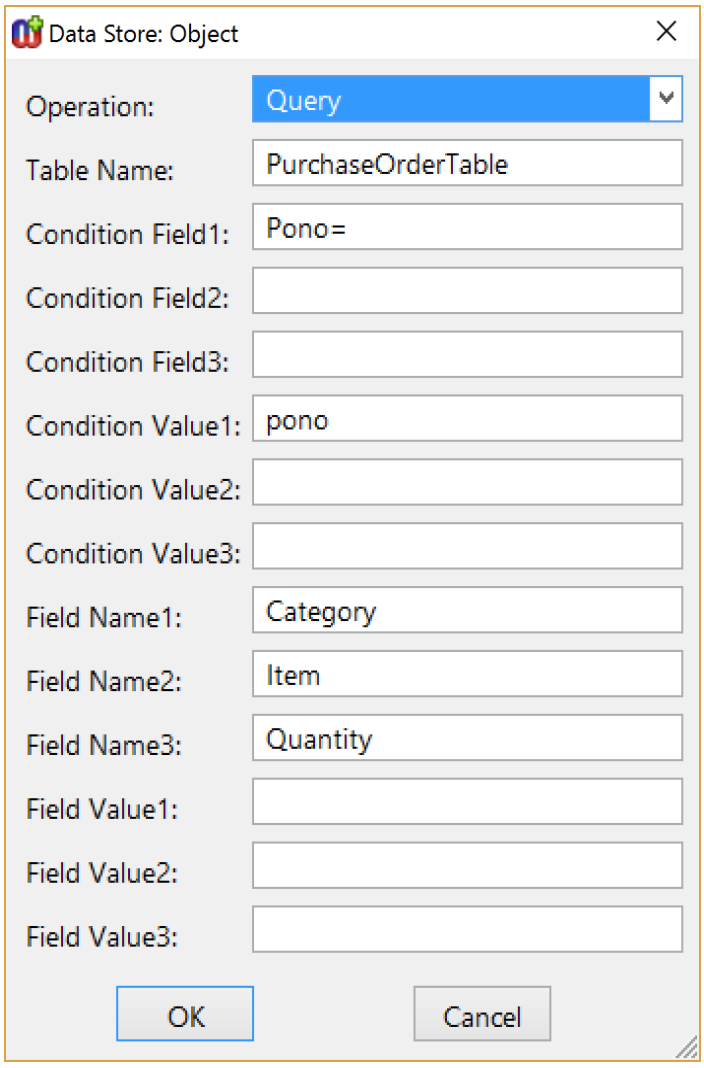


Figure 5.14. The dialog box for database operation definition

5.3 Testing Environment Generation

Our approach provides a very simple and easy way to generate operational endpoints from their models. There are three tasks: (1) to create two Java project folders (e.g. purchaseserver and purchaseclient) for hosting server and client side codes and load our miscellaneous files (such as utility classes, SUT API classes and Ant build files); (2) to transform models to code by code generators and copy them to the server project folder; and (3) to run our supporting toolset for packaging Tomcat service and providing testing service to SUTs.

To transform a model, we need to open the model graph. Figure 5.15 illustrates the steps to transform the example signature model to a WSDL file. We first select "*Generate*…" option from the Graph menu drop-down list. A pop-up box appears and lists all the available code generators for the model. We doubly click the right code generator for the form to be transformed to and press OK button. Then, the code will be generated automatically.

Figure 5.16 shows the testing service provided by the ERP endpoint at the URL *localhost:8080/axis2/services/listServices*. The endpoint provides 11 operations to SUT, including "logon" and "logout" for managing a sales session, "returnhome" for returning the endpoint to Home state and other eight operations for a sales workflow process.

To demonstrate how the endpoint provides its testing service to a SUT, we have created a dummy SUT for invoking endpoint operations as shown by Figure 5.17. The SUT calls the ERP endpoint testing service and send the following service requests in sequence: "logon" → "porequest" → "inventorycheck" → "supplierpo" → "supplierpoapproval" → "approvalnotification" (first level) → "approvalnotification" (second level) → "approvalnotification" (third level) → "supplierpoapproval" → "supplierdelivery" → "paymentrequest" → "deliveryrequest" → "logout". This SUT instantiates all operation API classes (see an example code of Figure 4.22), and they call the endpoint operations through an Axis2 stub class. The response messages from the endpoint operations are printed out on screen.

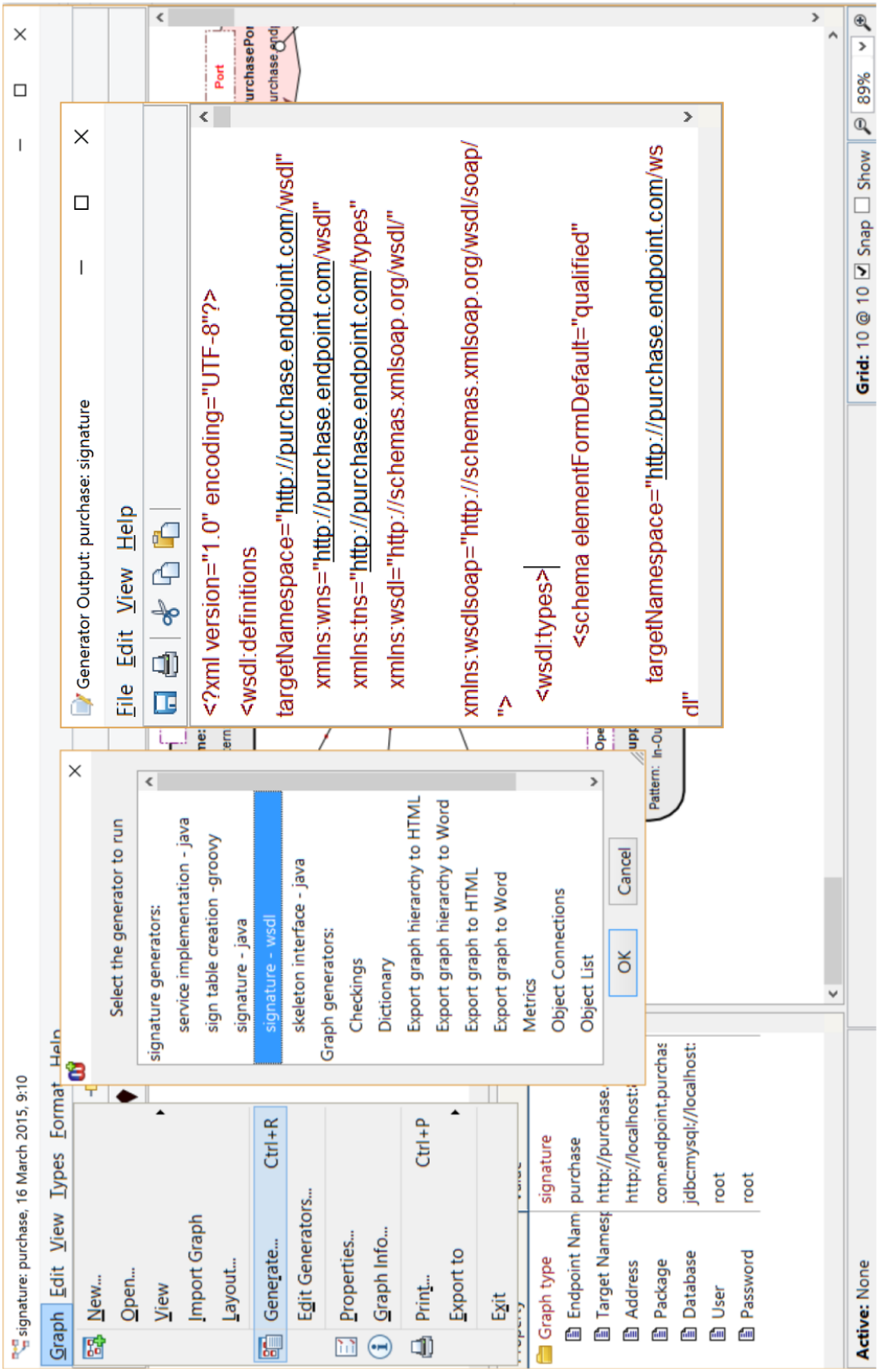


Figure 5.15. The example signature model transformation

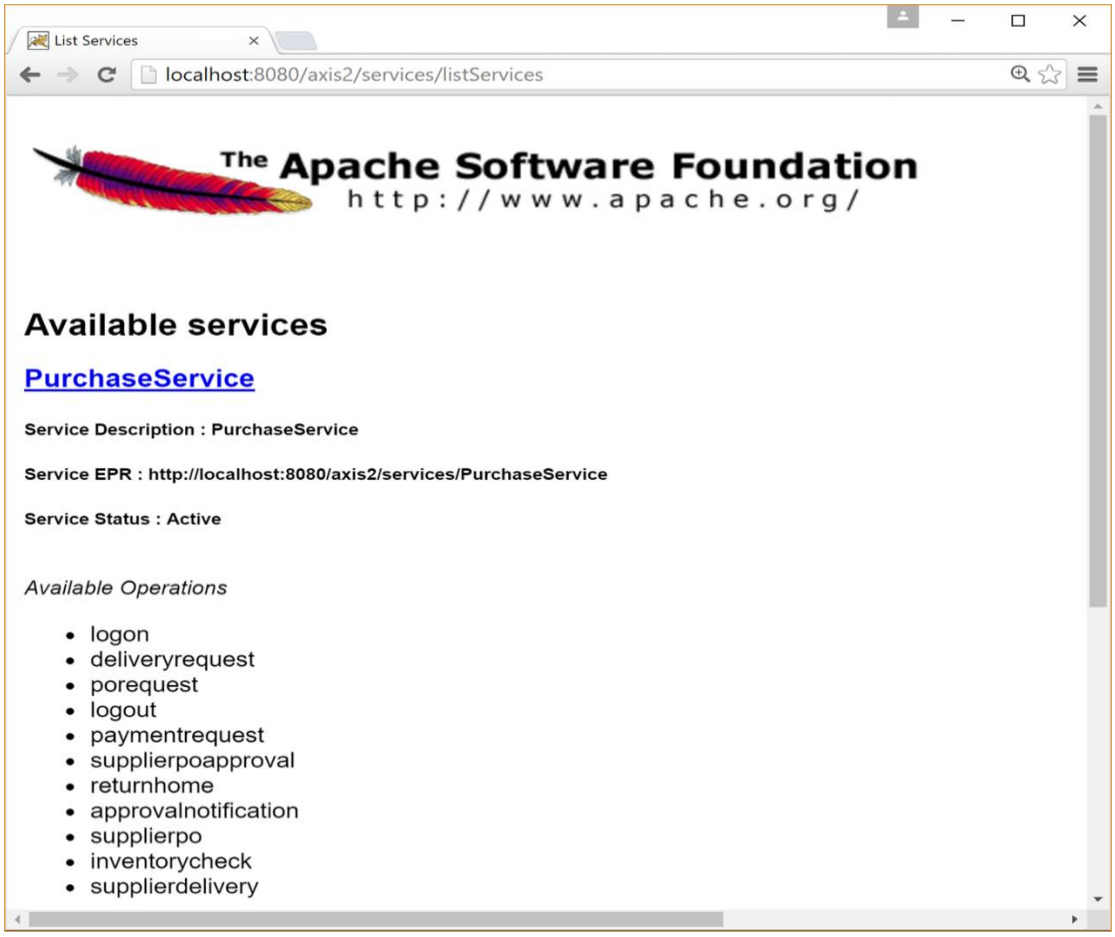


Figure 5.16. The example testing service through Tomcat

To capture and see the exchanged messages, we use TCPMon tool [147] to act as an intermediary between the SUT and endpoint. TCPMon accepts connection from the SUT on one port (e.g. 8888) and forwards the incoming traffic to the endpoint running on another port (e.g. 8080). Figure 5.18 shows a screenshot of TCPMon tool, capturing a SUT request message on the left and the response from the endpoint on the right. From the captured messages, we can see that the SUT sends a PO request with five parameters included in a SOAP envelope body. After processing the request, the endpoint generates a response with two parameters of an error code and error message.

To demonstrate how to model an endpoint using our TeeVML tool, we have recorded a short video at: **https://www.youtube.com/watch?v=H3Vg20Juq80**. The video provides a stepwise instruction to model an endpoint operation and create the endpoint by using our supporting tool. Also, we have put the example source codes, including MetaEdit+ and Java codes on line: https://sites.google.com/site/teevmlase/.

```
package com.endpoint.purchase;

public class PurchaseClient {

    public static void main(String[] args) {

//initiate all the operations
        LogonClient logon = new LogonClient();
        LogoutClient logout = new LogoutClient();
        PorequestClient porequest = new PorequestClient();
        InventorycheckClient inventorycheck = new InventorycheckClient();
        PaymentrequestClient payment = new PaymentrequestClient();
        DeliveryrequestClient delivery = new DeliveryrequestClient();
        ReturnhomeClient returnhome = new ReturnhomeClient();
        SupplierpoClient supplierpo = new SupplierpoClient();
        SupplierpoapprovalClient supplierpoapproval = new SupplierpoapprovalClient();
        ApprovalnotificationClient approvalnotification = new ApprovalnotificationClient();
        SupplierdeliveryClient supplierdelivery = new SupplierdeliveryClient();

        System.out.println(logon.Logon(12345,"jian","password"));
        System.out.println(porequest.Porequest(40000,"abcde","stationary","pen",5000));
        System.out.println(inventorycheck.Inventorycheck("stationary","pen"));
        System.out.println(supplierpo.Supplierpo(50000,"stationary","pen",5000,3));
        System.out.println(supplierpoapproval.Supplierpoapproval(50000));
        System.out.println(approvalnotification.Approvalnotification("simon",50000));
        System.out.println(approvalnotification.Approvalnotification("james",50000));
        System.out.println(approvalnotification.Approvalnotification("jenny",50000));
        System.out.println(supplierpoapproval.Supplierpoapproval(50000));
        System.out.println(supplierdelivery.Supplierdelivery(50000));
        System.out.println(payment.Paymentrequest(40000));
        System.out.println(delivery.Deliveryrequest(40000));
        logout.Logout(12345,"jian","password");
    }
}
```

Figure 5.17. The code of a dummy SUT class

5.4 Summary

In this chapter, we use the ERP system as an example to demonstrate how endpoint functional layers are modeled by using TeeVML. To show the testing functionality of our approach, we purposely choose the endpoint application with both static and dynamic interactive aspects with its SUTs.

In general, signature layer modeling is a tedious and time-consuming task. There are 10 operations, 19 request and response messages and several parameters in each of these messages. The key design consideration of Signature DSVL is to improve development productivity by increasing reusability. We adopt a three-level hierarchical DSVL architecture, and a significant amount of signature modeling effort is reduced. In contrast, protocol modeling is very simple and easy, we just drag and drop the state visual constructs to represent endpoint states and link a “from” state to a “to” state by a transition, constraint transition or timeout relationship.

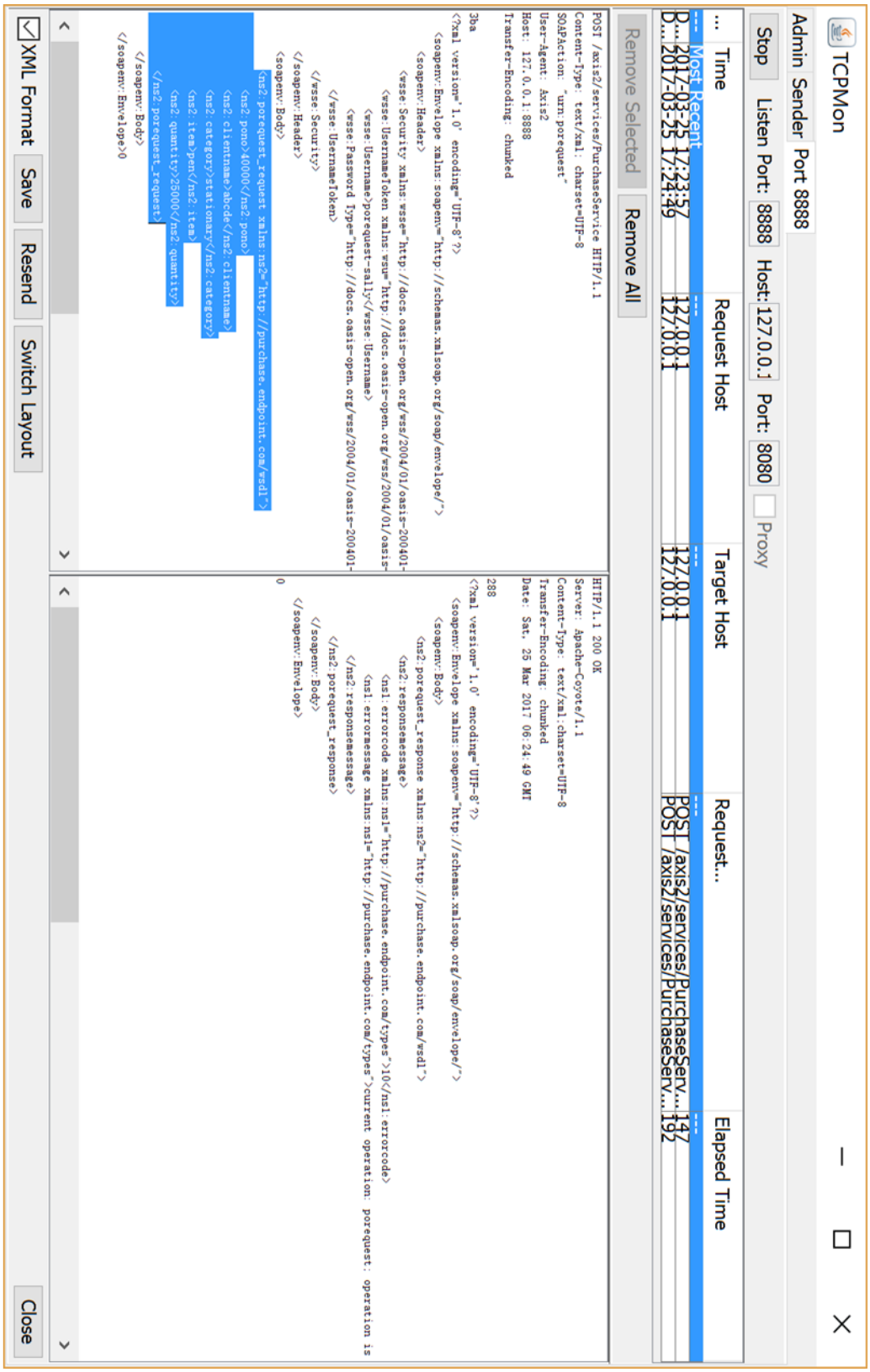


Figure 5.18. The request and response messages captured by TCPMon

Behavior modeling is a little more complicated than both signature and protocol modeling. Behavior DSVL includes some programming constructs to access persistent data, process business logics, and control dataflows. Behavior layer modeling is not directly involved in validating SUT operation requests. Rather, its result is used to determine alternative process flows to the next operation state. Therefore, the endpoint behavior modeling could be simpler than its real application, and its result does not require to be 100% accurate under all circumstances.

In a realistic enterprise environment, application security requirements may enforce extra constraints on the validity of a service request. Some of the constraints are role-based, so that some operations are accessible to a certain user group only. Others are security policy related, such as encryption requirement for service requests sending over an insecure network or specific pattern required for some service parameters. In the next chapter, we introduce our security modeling approach and Security DSVL, and use it to model an example application's security attribute.

CHAPTER 6

QoS Modeling – Security Attribute Example

In Chapter 1, we proposed a new software interface description framework where an endpoint is logically divided into horizontal layers for processing request messages and vertical attributes for confirming QoS aspects. In Chapter 3&4, we introduced our DSVLs for modeling endpoint horizontal layers. In this chapter, we select security as a key vertical QoS attribute to show how endpoints' vertical attributes can be modeled by using our approach.

Using the same development process as we followed for functional layer modeling, we first conduct a domain analysis to collect security modeling requirements and create a security metamodel. We then design our security DSVL and implement corresponding code generator. Here, we use the example introduced in Chapter 5 and modify it to add security requirements as a case study to demonstrate how QoS attributes, using the security as an example, can be modeled using our new security DSVL.

6.1 Introduction

An endpoint, as a constituent of a testing environment, validates service requests sent from a SUT and returns valid responses to the SUT to make it think is deployed in a real production environment. Other than validating the correctness of service requests from a functional point of view, an endpoint implementation should also impose some security requirements on these requests to assure the compliance with the company's security policy. There may have some security restrictions on the use of the endpoint service, for example to check the assigned permission to the user who uses a SUT to send requests to access the endpoint provided services.

6.1.1 Security Requirements

Security requirements should not be arbitrarily defined for individual systems. Instead, a system's security must be implemented in a systemic way to conform with the company's information security policy [148]. Information security policy is defined as "The set of

laws, rules, and practices that regulate how an organization manages, protects, and distributes resources to achieve specified security policy objectives. These laws, rules, and practices must identify criteria according to individuals' authority, and may specify conditions under which individuals are permitted to exercise their authority." [149] A security policy may (but not exclusively) include the following terms:

- **User Authentication** – users must be authenticated by their IDs and passwords and/or other means such as Single Sign On (SSO) [150] or One-Time Password (OTP) [151]. A client may send user ID and password to its service provider as part of a request message body or by a UsernameToken (username and password) as a request message header;
- **Operation and Data Accessibility** – an authorised user must be able to access both the operations and data of a service whenever they need;
- **Transmission Security** – requests sent from client must be protected by network and/or data security. Network security consists of the policies and practices to prevent and monitor unauthorized access, misuse, modification, or denial of a computer network and network-accessible resources. Data security relies on cryptographic technologies to transform data into unintelligible data for transmission;
- **The Principle of Least Privilege** – users must be able to access only the operations and data that are necessary for their legitimate purposes;
- **Avoidance of Conflict Of Interest (COI)** -- a person's professional judgement or actions regarding a primary interest may be unduly influenced by his/her secondary interest;
- **Other Security Aspects** – an endpoint may impose some security rules on its clients, such as restricted time frame for certain operations, allowable times for unsuccessful authentication, pattern requirement for user password, etc.

An effective security policy is developed with the understanding of the business process, security issues, potential attacks, required level of security, and factors that make a system vulnerable to attack. To identify potential security risks to computer systems, the five common security aspects [152] need to be understood:

- **Confidentiality** – is related to the secrecy of confidential data and unauthorized persons should not gain access to the data or know the content of the data;
- **Integrity** -- involves accuracy of data, and only authorized persons are allowed to create, edit, and delete data in an approved manner;
- **Authenticity** -- verifies the origin of the message and the identity of the person or system who sends the message;
- **Privacy** -- is the ability to protect users' personal secrets and prevent hackers from invasions of personal space;
- **Availability** -- means computer assets should be available for and accessible to authorized persons when they need and should not be interrupted.

To protect data from these security aspects, the company must ensure that first, its data are properly secured when they are stored and used in-premises and second, the data are prevented from malicious attacks by hackers when they are in transit. We discuss the details of the security measures for handling these two security situations in the followings of this section.

6.1.2 In-Premises Data Security

When systems and data are kept in an enterprise computing environment protected from external hackers' attacks by firewalls, it does not mean that we can be worry-free. Internal staff may violate the company's security policy by finding some security flaws during their daily work and breach systems' security for their own interest. For this scenario, the principle of least privilege and avoidance of conflict of interest are the two main considerations for assigning system and data permissions to users. Thus, an approach to model user and system permission assignment and control the accesses to systems and data could be a better choice than using some sophisticated cryptographic technologies.

There are many security control models in use to assign users' access to systems and data. These include some legacy models such as Chinese Wall [97] and Bell and LaPadula [98], current popular Role-Based Access Control (RBAC) model [99], and emerging Attribute-Based Access Control (ABAC) model [100]. A study showed that most commercially available enterprise software products are either compatible with RBAC or have

embedded role capabilities and RBAC model is used by the majority of companies with more than 500 employees [105].

RBAC defines a security control mechanism around roles and privileges and supports both Mandatory Access Control (MAC) [153] or Discretionary Access Control (DAC) [154] types. The components of RBAC such as role-permissions, user-role and role-role relationships make it simple to perform user permission assignments. Due to its ease of use and popularity, we select RBAC model for our in-premises data security modeling.

The National Institute of Standards and Technology (NIST) proposed a RBAC standard with four components from the essential to comprehensive aspects of RBAC: (1) Core RBAC, (2) Hierarchical RBAC, (3) Static Separation of Duty Relations (SSD), and (4) Dynamic Separation of Duty Relations (DSD) [99]. Particularly, we are interested in the third RBAC component, SSD, which combines the essential concepts in Core RBAC, role hierarchical structure in Hierarchical RBAC, and enforcing avoidance of conflict of interest security rule by SSD.

Figure 6.1 illustrates SSD component entities and their relationships. The right-hand part is an application with operations and assigned objects. To use the operations, the access right to the objects needs to be granted. The left-hand part is about the assignments of users and roles. Roles are in hierarchical structure and the roles and sub-roles are given permissions to use some particular operations. Users are assigned to roles with some assignment constraints, including the definition of mutually disjoint user assignments with respect to sets of roles. If a user is assigned to one role, the user may be prohibited from being a member of another role.

6.1.3 Data Security in Transit

In a distributed computing environment, service consumers and their providers may be hosted in different locations interacting with each other through insecure connections. Their communications are exposed to hacker's attacks from the outside world, and this could result in stolen, eavesdropped, manipulated and damaged data. Hackers are usually highly skilled computer experts and very sensitive to any security vulnerabilities. They are capable of breaking into networks to intercept and decrypt transmission data and cause damages to the company financially and legally.

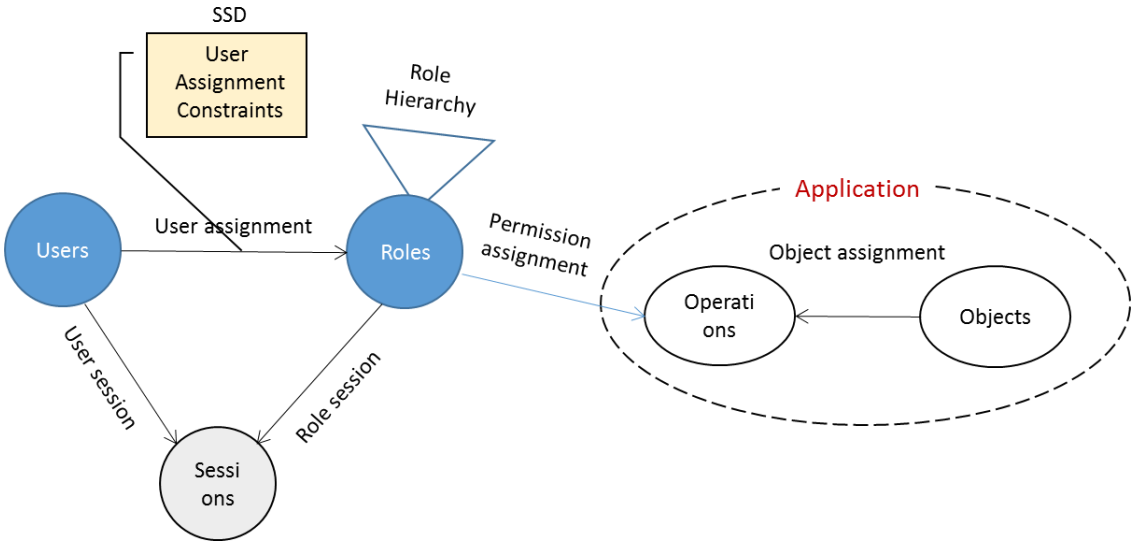


Figure 6.1. Static Separation of Duty relations (SSD) model -- A RBAC component

To ensure data security in transit, the three security requirement aspects of confidentiality, integrity and authenticity defined earlier must be met. As the data in transit are out of company's control, data security cannot be ensured by means of organizational management. In general, there are two different approaches to secure data in transit. One approach is to secure the communication channel to prevent the data from being intercepted illegally. Another approach is to encrypt the transmission data, so that a person who intercepts the data will not be able to know the content and/or make modification.

Public key cryptography is a cryptographic system that uses a pair of public and private keys to sign and verify messages [155]. The public key is certified for a system's ownership by a recognized Certificate Authority (CA). It may be disseminated to other parties a system may communicate with. The private key is known only to the system. Any other systems can encrypt a message using the public key of the receiver, but such a message can be decrypted only with the receiver's private key. Figure 6.2 depicts a use example of the public key cryptography for ensuring authenticity. A client uses its private key to sign the request sent to a service. The service uses the client's public key to verify the signature of the message. On the other end, the corresponding response is signed by the service using its private key and verified by the client using the service's public key.

In practice, a hash value (also called digest) is generated for a message to be signed. The digest is encrypted with the private key, and the encrypted digest value is sent with the

message. Using the signed message digest guarantees both message integrity (any modification to the message will change the digest value) and authenticity (the private key is used to encrypt the digest). The message confidentiality is achieved by encryption using the public key. Therefore, the three aspects of message exchange security are implemented by using a public-private key pair.

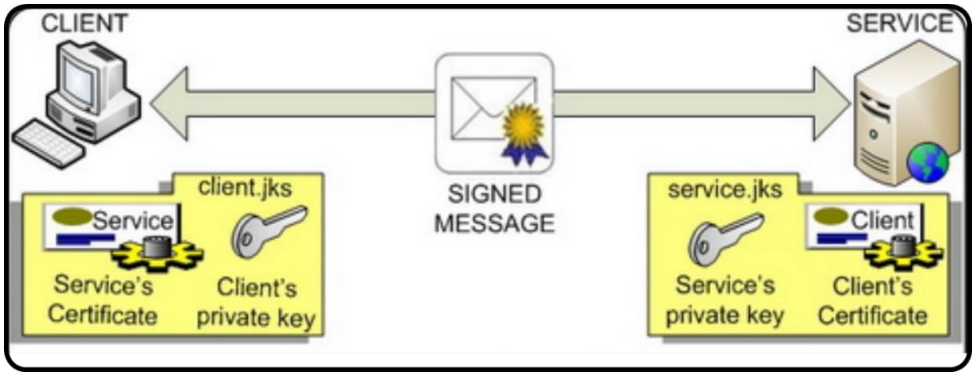


Figure 6.2. An example usage of public key cryptography

6.2 Security Domain Analysis

We conduct our security domain analysis using the motivating example and business case introduced in Chapter 1 with adding security attribute on the top of the existing functional layers. As information security is closely related to the business process of the company, our security domain analysis starts from investigating organizational structure and business process to support a sales process first. The security requirements for ensuring the proper access controls are then defined.

The business case for the domain analysis consists of an in-house back-end ERP system as the endpoint and a public cloud front-end CRM application as the SUT. To process a client's purchase order, a user uses the SUT to send requests to the endpoint for invoking the provided operations. From a security point of view, the endpoint must validate these requests based on the user's permission to access the operations. Different from the example we introduced in Chapter 1, the whole sales process cannot be handled by a single user any more. Instead, multiple users from different functional departments and divisions are involved in a workflow process and the users' identities are included in operation requests for user authentication.

For a broad coverage of the applicable domain, we use a global company with two divisions for its overseas and domestic clients as an example scenario. In this scenario, to

implement the principle of least privilege, the company has different functional departments for handling different tasks, such as Sales for initiating a purchase request, Purchase for managing supplier purchase activities, Finance for payment, etc. To avoid the conflict of interest, all supplier purchase orders must be approved by the purchaser's managers at different levels, and they must be different individuals.

Based on the company's organizational structure and business process, the endpoint has some security requirements for the SUT requests to access its service. It must validate these service requests from the security point of view before processing them functionally. We use a flowchart diagram to explain the endpoint security requirements by going through some operations of a typical sales process (see Figure 6.3):

- When a user starts a sales process to put a "porequest", the endpoint will check whether the user is from the sales department and the client belongs to the user's division. Otherwise, the endpoint will terminate the request process and report a security defect;

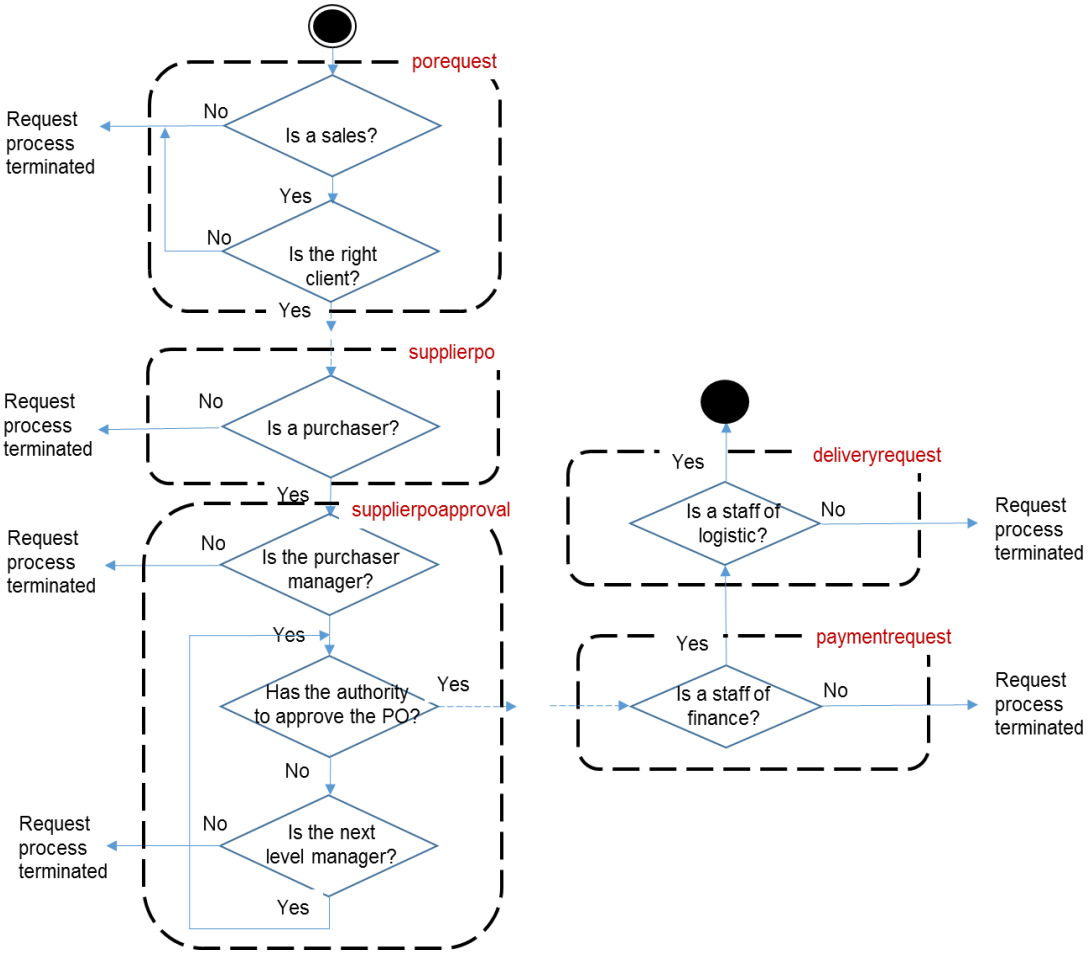


Figure 6.3. The ERP endpoint application security requirement

- If a supplier purchase is needed for meeting the PO's demand, a "supplierpo" request must be issued by a staff from the purchase department;
- The supplier PO must be approved by the immediate manager of the purchaser first. If the manager's approval limit is lower than the PO amount, a higher level manager's approval would be needed. The PO approval is an iteration process until the required authority level is reached. The purchaser and managers must be different individuals;
- After the supplier PO delivery, a "paymentrequest" is sent by a staff of finance department. This is followed by a "deliveryrequest" sent by a staff from logistic department. This marks the end of a sales process.

To model these security aspects, the proposed security model should include the function and division roles of the company and operations and resources of the endpoint. The users belonging to a function role are assigned the permission to use an operation. Similarly, a resource's access right is given to the users of a division role.

Figure 6.4 illustrates the endpoint security control process involving these security entities. Below, we explain how the endpoint enforces its security rules to the service requests:

- The endpoint validates a service request based on the user permission, who initiates the request through a SUT;
- The user, who sends a request to an operation by using a SUT, must belong to a functional role that has permission to access the operation;
- The user, who sends a request to an operation by using a SUT, must belong to a division role that has permission to access the objects, that the operation must use to process the request;
- An operation request must meet some other security requirements based on the use scenarios, such as message encryption for sending request over an insecure connection.

From our study of the sales process, we develop our endpoint security metamodel shown by Figure 6.5. The metamodel is based on the NIST Static Separation of Duty Relations (SSD) RBAC model we introduced earlier. We add the division role and resource to the

NIST model for a more rigorous restriction on the service requests. To implement the avoidance of conflict of interest, we define the function role and user relationship as one-to-many (a user can only be assigned to one function or sub function role). Table 6.1 lists all the domain entities in the security metamodel, including their names, detailed semantic descriptions and inter-relationships among these entities.

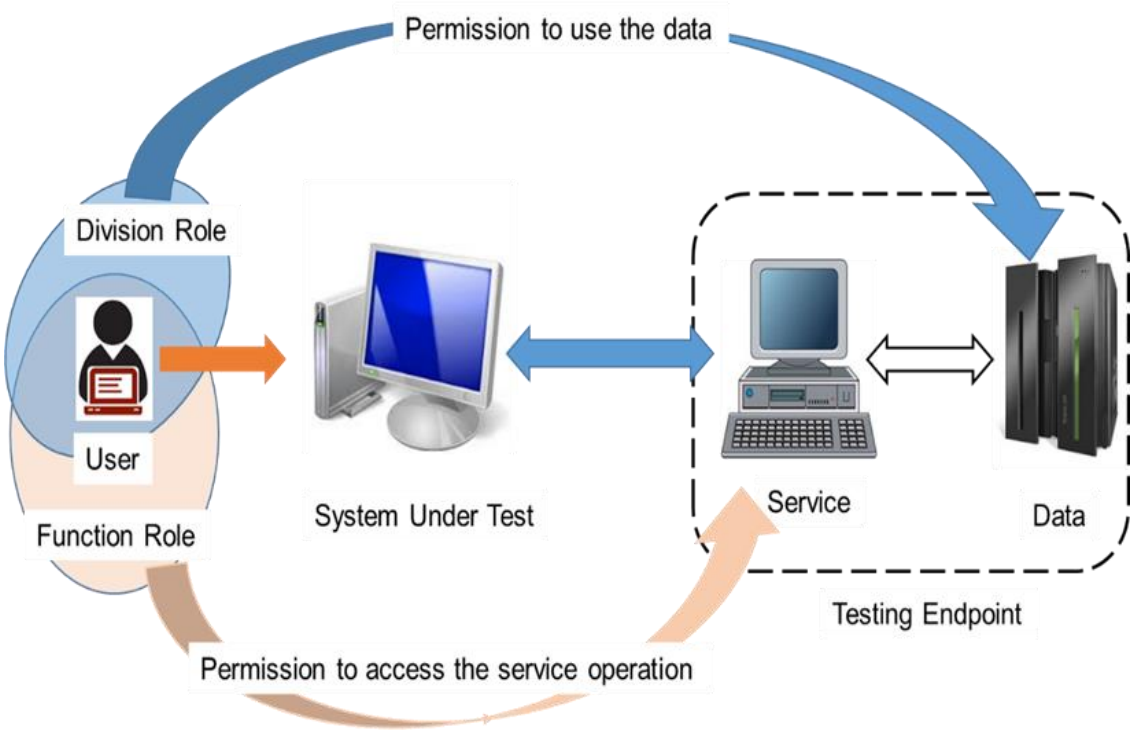


Figure 6.4. Endpoint security control process

As the CRM is a public cloud application, the communications between the ERP endpoint and CRM SUT are beyond the company's monitoring and control by firewalls. Depending on the network condition and application security requirements, different security techniques can be used to protect the communications:

- **Plain Text** – plain text is the most basic form to send messages without any security control. It is only suitable for both the client and server hosted in a secured environment, such as Intranet;

  HTTPS Protocol -- HTTPS is a HTTP connection encrypted by Transport Layer Security or Secure Sockets Layer [156]. It is used for authenticating the visited website and protecting of the privacy and integrity of the exchanged data. HTTPS protocol can ensure a communication confidentiality and integrity;

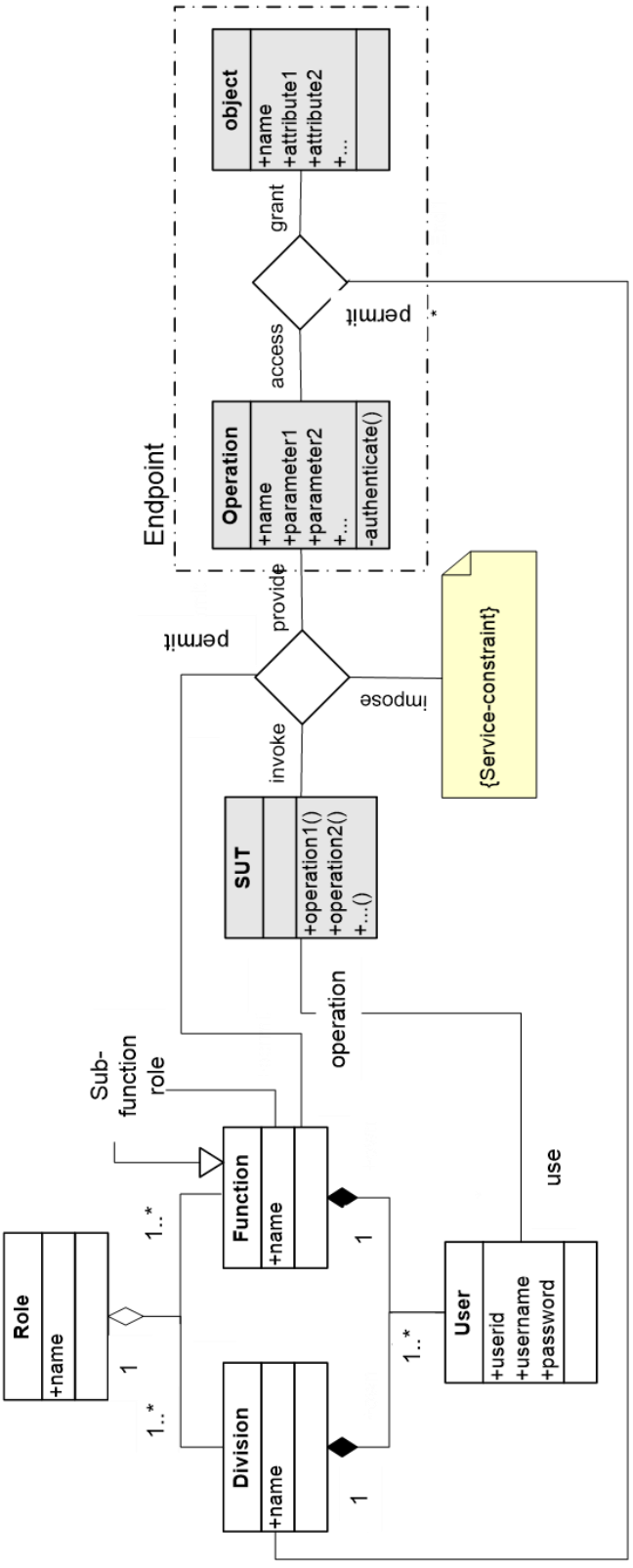


Figure 6.5. Endpoint security modeling metamodel

Table 6.1. Description of endpoint security metamodel

Entity	Description	Relationship
Role	The top level role consists of a number of function and division roles.	One-to-many relationship with Functional Role and Division Role.
Function Role	A function role has one or more user(s) for performing a specific task in a company and is often related to a company department. A function role may have sub-roles, an example is the manager role, which may have level1, level2, level3, … sub-roles.	One-to-many relationship with User; One-to-many relationship with Operation.
Division Role	A division role consists of a special group of users. It is normally related to a cross-function division in a company.	One-to-many relationship with User; One-to-many relationship with Object.
User	A staff in a company with a computer account.	Many-to-one relationship with Functional Role; Many-to-one relationship with Division Role.
SUT	A user uses the SUT to access endpoint service.	Many-to-many relationship with User.
Service-constraint	The restrictions on invoking a service operation. They are related to some special security requirement, such as an encryption technique for communications.	
Operation	Provided by the endpoint for performing a specific task. An operation often needs to access some objects.	Many-to-one relationship with Function Role; Many-to-many relationship with Object.
Object	Persistent data to support operations.	Many-to-one relationship with Division Role; Many-to-many relationship with Operation.

- **Digital Signature** -- digital signature is a cryptography technology for demonstrating the authenticity of messages being exchanged [157]. A valid digital signature gives a recipient reason to believe that the message was created by a known sender, and the sender cannot deny having sent the message and the

message was not altered in transmission. Digital signature can ensure the integrity and authenticity;

- **Password Digest** -- Password digest is one of the agreed-upon methods an application can use to negotiate how a user password is encrypted with a client [158]. It applies a hash function to the password with a random generated nonce and timestamp before sending it over the network. Password digest can ensure confidentiality**.**

The entity Service-constraint specifies which security technique is to be used for the communications between the endpoint and SUT.

6.3 Security DSVL Design

A good DSVL should provide the right level of abstraction and can describe intended solutions in domain terms and hide implementation details. It should be expressive enough in the problem domain and have precise semantics to enable formal reasoning about domain models. We use the same DSL development guideline as we followed for the functional layer DSVLs to develop our Security DSVL. In this section, we only provide the appearance designs of the DSVL visual constructs. Their properties and the usages are described through a case study in the next section.

The central piece of our Security DSVL is role construct, which defines the permissions for users to use endpoint operations and access resources. The visual symbol of the role construct is a rounded rectangle and uses iterative shapes behind the rectangle to represent a composite element. It is filled with light grey colour and a textual annotation at the top to distinguish it from other visual constructs. Figure 6.6 shows the symbol of the role visual construct.

Not all users of a function role assigned to an operation can access the operation, as the operation may have some additional runtime constraints. That is the reason why we add sub-role entity to the function role for specifying some properties of these runtime constraints. Sub-roles are defined by decomposing their parent role. The hierarchical function role structure helps to reduce diagram representation complexity and groups all sub-roles belonging to a parent together. As sub-role is closely related to role, their appearance designs should be in consistence to a certain extent. We design sub-role visual

construct in a rectangular shape with iterative shapes behind the rectangle and filled with light blue colour. Figure 6.7 shows sub-role visual construct.

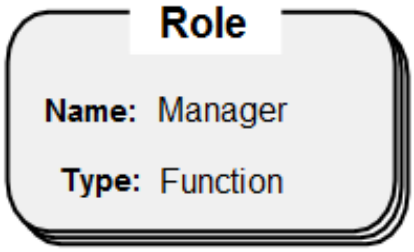


Figure 6.6. Role visual construct

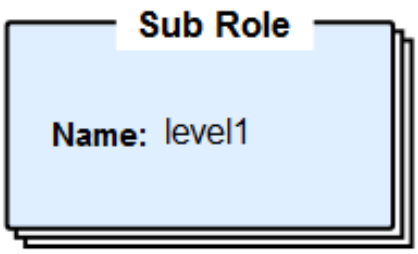


Figure 6.7. Sub-role visual construct

A user belongs to a role or a sub-role. As a role instance may contain both sub-roles and users, the visual distance between them should be big enough and their visual appearances must be easily distinguishable. We use shape and colour as the main visual variables, and user visual construct is designed as a trapezoidal shape and filled with light yellow colour. Figure 6.8 shows the symbol of user visual construct.

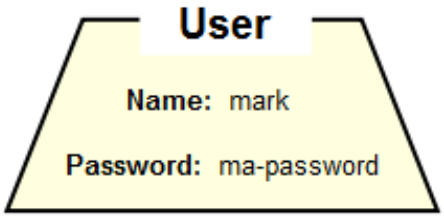


Figure 6.8. User visual construct

Operation visual construct represents endpoint service operations. The main visual design consideration is to discriminate it from role and resource visual constructs. We use a blank ellipse with textual annotation at the top. Figure 6.9 shows the symbol of operation visual construct.

Resource construct specifies the table records that users in a division role have the right to access through an operation. As both resource and operation constructs are part of an endpoint with closely related semantics, we use a similar visual symbol design as operation visual construct. Resource visual symbol is a blank hexagon with textual annotation. Figure 6.10 shows the symbol of resource visual construct.

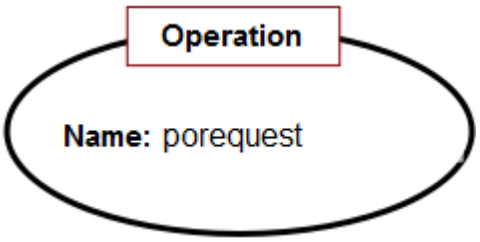


Figure 6.9. Operation visual construct

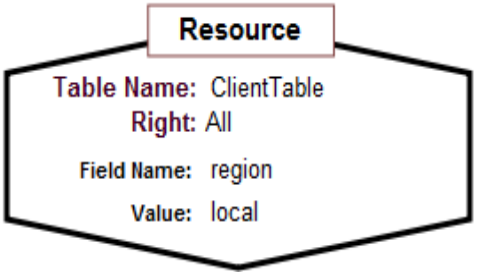


Figure 6.10. Resource visual construct

By using association relationships, function roles are assigned to operations and division roles to resources. Association relationship visual construct is a black arrow line shown in Figure 6.11.

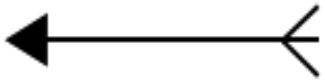


Figure 6.11. Association relationship visual construct

The last visual construct is security constraint, which is used for specifying other security aspects, such as which cryptographic technique is used for message transmission and the allowed number of authentication failures. The visual symbol is designed in a way to maximize its perceptual discriminability from other visual constructs. The visual construct is a rectangle filled with light yellow colour. There is also a unique security control icon on the top left. Figure 6.12 shows the symbol of security constraint visual construct.

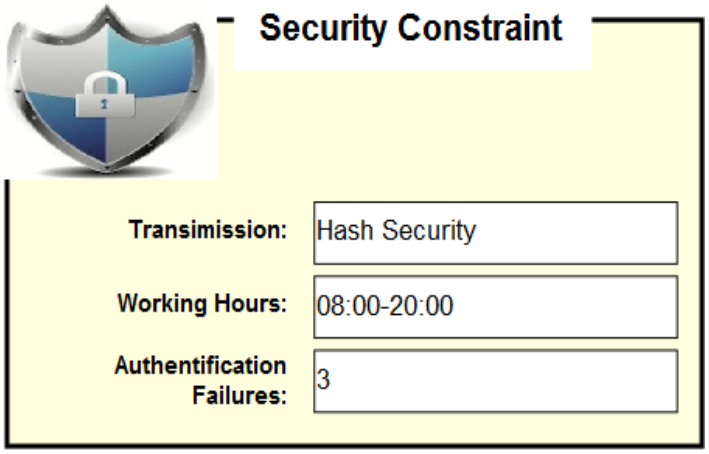


Figure 6.12. Security constraint visual construct

6.4 Case Study - Endpoint Security Modeling

In Chapter 5, we demonstrated how an example endpoint functional layers can be modeled by our TeeVML. In this section, we model endpoint security attribute and authenticate users' access rights to endpoint operations based on RBAC security model. We here demonstrate a case study by reusing the ERP application introduced in our security domain analysis of section 6.2.

To model an endpoint security attribute, we need to instantiate the DSVL visual constructs and link related instances together by using association relationships. There are three tasks in the order: (1) instantiating Role as function and division roles, Operation and Resource; (2) instantiating Sub-role and User, and adding their instances into function and division roles; and (3) specifying other security constraints.

6.4.1 Instantiation of Role, Operation and Resource

Both function and roles are initiated using Role entity by assigning name to their name property. For those function roles, the type property selects Function from a drop-down list. Similarly, the division instances are defined by selecting Division. Their instances are easily distinguishable by their type properties.

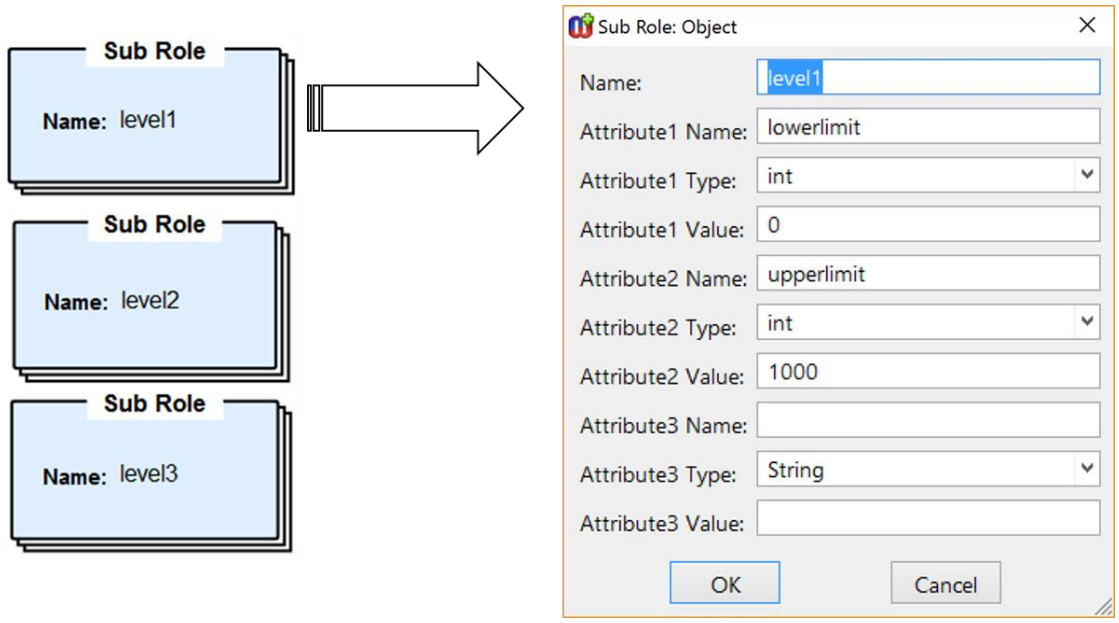
Operation instances are defined by using Operation entity and assigning the name property. In contrast, Resource entity instantiation is a little bit more complicated. We need to specify the records in a table to be accessible to a division role. These records are filtered by assigning the properties of table name, field name and value. The access right

to those records is defined by selecting a value of All, Add, Delete, Modify or Read from the right property drop-down list.

The cardinality relationship between function role and operation is many-to-one and they are linked by using association relationships from function role instances to operation instances. In the same way, division instances are assigned to resource instances.

6.4.2 Definition of Sub-Roles and Users

Only function role can have sub-roles, and they are defined by opening the decomposition graph of a function role instance. To define the runtime conditions, sub-role has three property groups of attribute name, type and value to specify three conditional variables. Whenever the conditions of these variables are met, the users in the sub-role can access the corresponding operation.



[a] Definition of sub manager roles [b] Dialog box for condition definition

Figure 6.13. Three sub-roles in a manager role and sub-role dialog box

Figure 6.13a shows three sub manager roles for different approval levels defined inside a manager role, and Figure 6.13b illustrates how level1 sub manager role is defined by using sub-role dialog box. There are two conditional variables for specifying the lower and upper approval limits, and a level1 manager can only approve a supplier PO with amount between 0 and 1,000. If the PO amount is over the limit, the next level manager's approval is needed.

User instances can be defined either in a Role or a Sub-role instance, and they are instantiated by filling in the visual construct name and password properties. The cardinality relationship between user and role is one-to-many, a role can have one or more users and a user belongs to only one role. Figure 6.14 shows two users defined in level1 sub manager role.

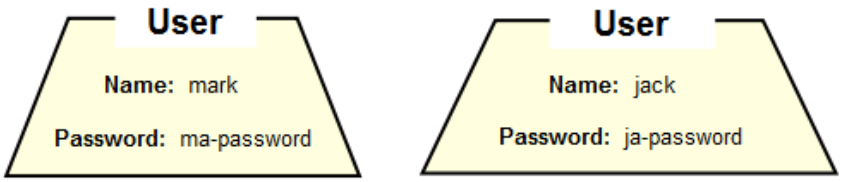


Figure 6.14. Two users in level1 sub manager role

6.4.3 Security Constraint Definition

There may have some extra constraints for the users in a function role to access assigned operations to the role. The most important one is the encryption requirement for transmitting request messages from a SUT to its endpoint. We have defined four policy files for the different security scenarios discussed in section 6.2, and users can select one from the transmission property drop-down list of security constraint visual construct. Thus, users do not need to know their implementation details, but only understand what the encryption requirement is for sending service requests. Other than the encryption requirement on message requests, users can also specify the time restriction on the use of the endpoint and the failure times for a user authentication.

6.4.4 ERP Endpoint Security Attribute Modeling

Figure 6.15 is the endpoint security model. The left-hand part shows that five function roles are assigned to eight operations. A function role can be assigned to more than one operations, but not vice versa. To access these operations, hash security policy must be used; the time for access the endpoint is between 8:00 to 20:00; and only three failures are allowed for user authentication. All these are defined by a security constraint instance at the top-right corner. Client and purchase order tables have separated groups of records accessible for different division users. Two divisions of domestic and overseas are assigned to these two tables based on the values of the two table fields of Region and Division.

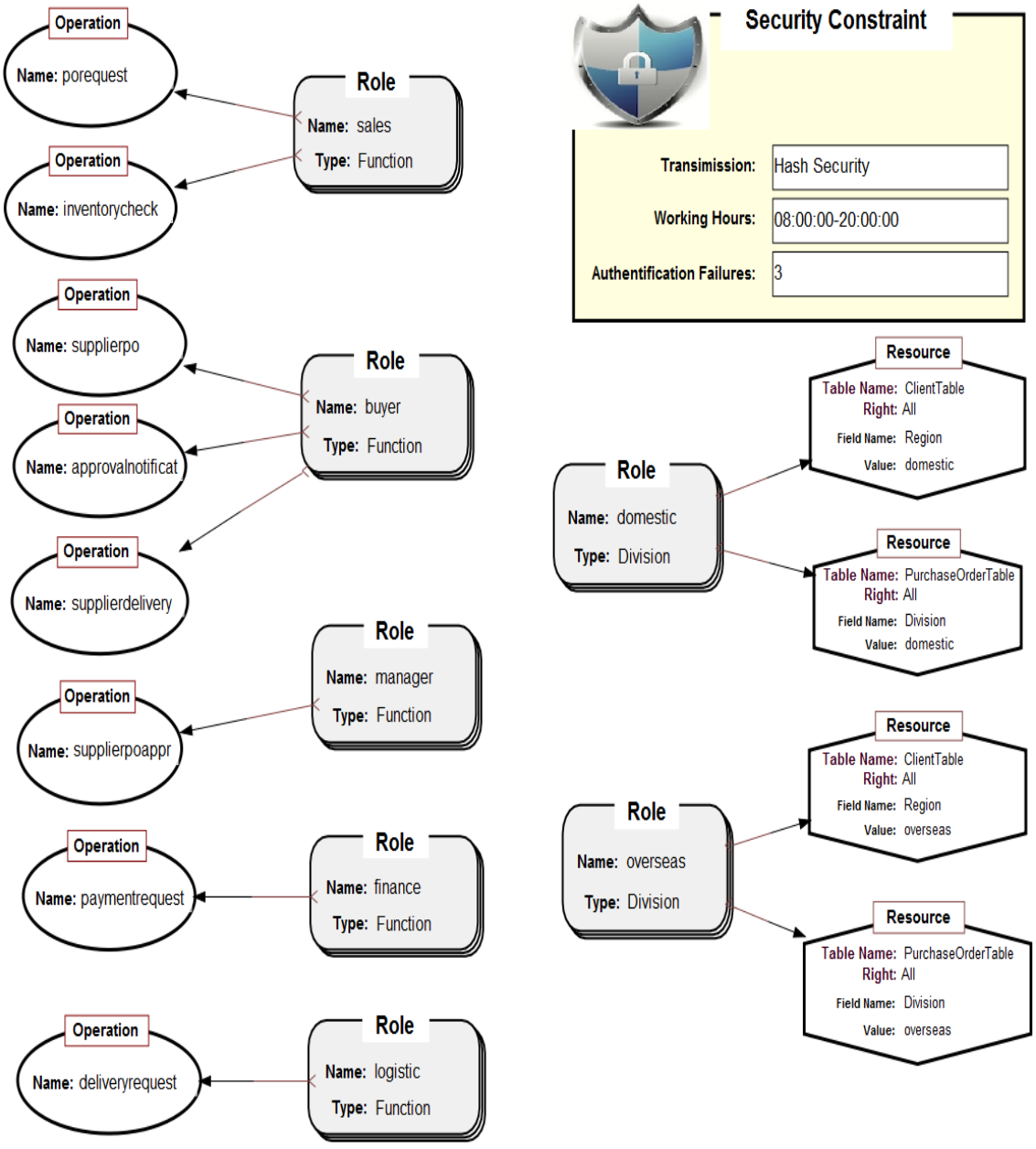


Figure 6.15. The example ERP endpoint security model

6.5 Implementation

The example endpoint security model discussed in the previous section needs to be transform to Java code. And the attached security rules, constraints and other security information must be stored in the endpoint database. To implement RBAC model, we must modify the endpoint database originally designed for functional layers modeling by adding security modeling related information, such as function and division roles, user assignments to these roles and relationships between operations and resources.

Accessing endpoints with enforced security requirements is different from those endpoints with functional layers only. There are not dedicated operations, such as "logon" and "logout", for managing a user session any more. Instead, user's credentials are normally sent with the request message for authentication, and each operation request must be authenticated before processing it. As the user's credentials may be sent over an insecure network, they must be protected from being intercepted and replaced by fraudulent data by applying some cryptographic technique.

These are the main areas to be discussed in this section.

6.5.1 Role-Based Access Control Implementation

To store permission information for RBAC implementation, we modify the database that is used for our endpoint functional layers modeling. Figure 6.16 illustrates the data modeling for our RBAC model with added tables and fields in red colour.

For grouping users together to assign their permissions for accessing operations and resources, we add a function role and a division role tables. The division role table is simple with only one field to specify division name. In contrast, to specify sub-role conditions, the function role table contains three variables, each with name, type and value fields. To associate a function role to an operation, a foreign key FunctionRole is added to the operation table. Similarly, a division role has a foreign key DivisionRole for assigning a division role to a resource.

For users to use an endpoint, they must be assigned to a function role and a division role. We add two foreign keys FunctionRole and DivisionRole to user table. For specifying security constraints aspects, we create a constraint table, which has five fields for specifying name and function role and defining a testing condition for a function role to access an operation.

We develop two Security DSVL code generators. One is for generating SQL script to create the tables and set up their relationships. Another one is to transform security models to Java code and store role permission information to the tables in Figure 6.16.

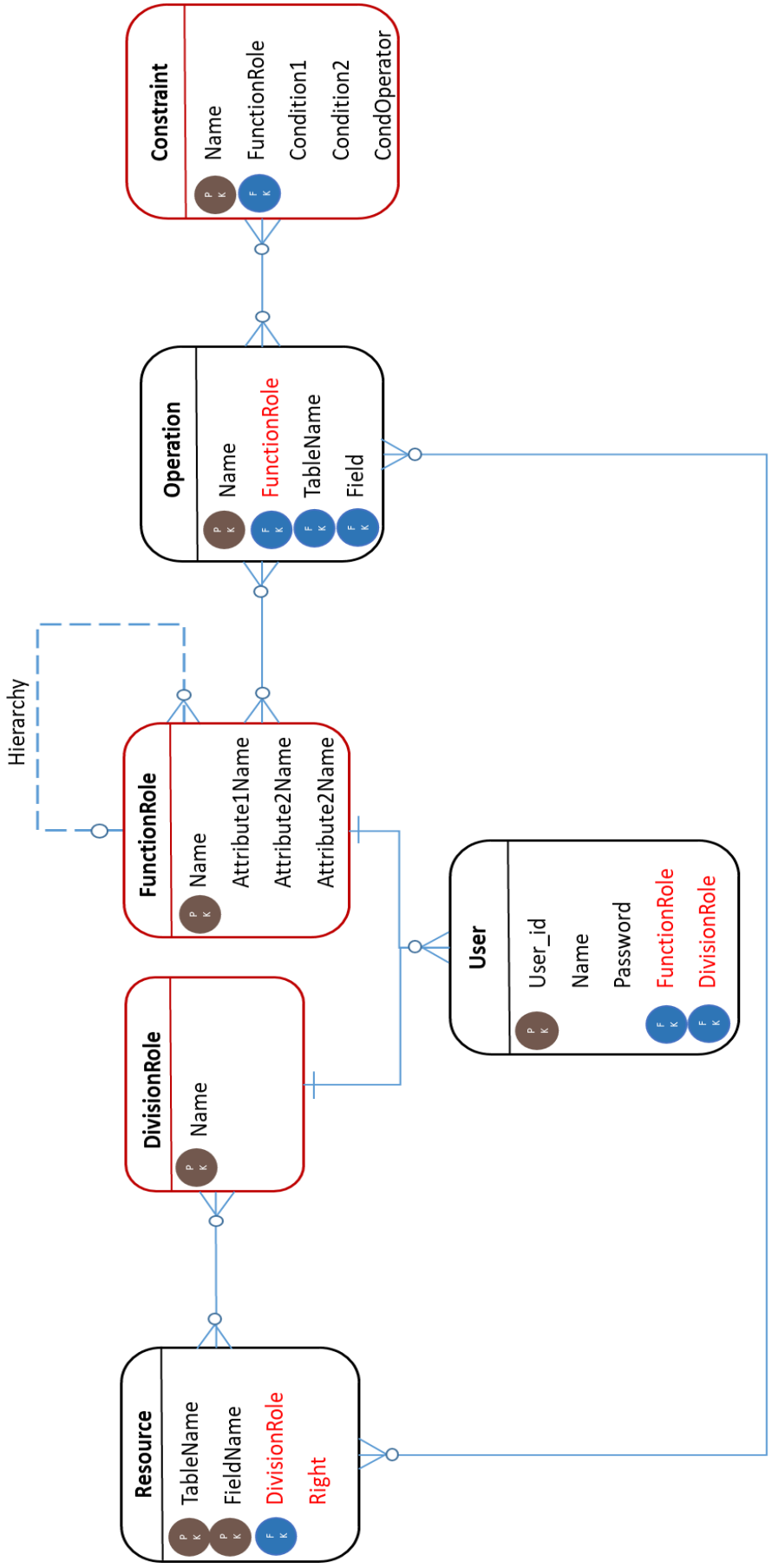


Figure 6.16. Data modeling for endpoint security implementation

6.5.2 Username and Password Security

To provide QoS to Web service consumers, Axis2 SOAP has the concept of handlers as the message interceptor for inserting QoS attributes into SOAP envelope headers [20]. Handlers intercept the flow of messaging and do whatever tasks they are assigned to do. Particularly for securing a message, the handler may need to pause the message flow, subject to some pre-defined preconditions and/or postconditions related to some security aspects.

It is not recommended to implement ad hoc security handling for individual applications, because even a minor and obscure oversight can lead to serious security vulnerabilities. Apache Rampart [159] is an Axis2 plug-in security module to implement WS-Security and WS-SecurityPolicy standards [64]. It intercepts messages at particular points to check or make changes to the messages' headers as appropriate. As both Axis2 and Rampart were developed and are maintained by Apache Software Foundation, Rampart is well integrated into Axis2 SOAP engine and provides native support for Axis2 security implementation. Also, Rampart supports most Web service security standards, it provides a comprehensive and trust worthy solution to address the security aspects of confidentiality, integrity and authenticity for Web services security.

UsernameToken describes how a Web service consumer can supply its user credential as a means of identifying the requestor to the Web service provider [160]. It conveys username and password information as a part of WS-Security header. Rampart implements several types of WS-Security security tokens with many options for how the tokens are constructed and used. The most basic form of UsernameToken sends both username and password in plain text as listed by a WS-SecurityPolicy configuration XML file in Figure 6.17. This policy consists of a standard WS-Policy wrapper around a WS-SecurityPolicy UsernameToken assertion. The IncludeToken attribute specifies the type of message flow included in the token. In this case, all messages flow from a request initiator to a request recipient.

The basic plain text UsernameToken does not provide much security, because both the username and password are exposed to anyone who is able to monitor and intercept the communication. Therefore, an encrypted communication channel over transport layer is a must. WS-SecurityPolicy defines a way to require the use of an encrypted channel, such as HTTPS protocol. Figure 6.18 lists a code snippet of the securing UsernameToken

policy. It includes a <sp:TransportBinding> element and nested a <sp:HttpsToken> element. The <sp:HttpsToken> element specifies that a secure HTTPS connection must be used in communicating with a service.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Client policy for Username Token with plaintext password,
sent from client
 to server only -->
<wsp:Policy wsu:Id="UsernameToken" xmlns:wsu=
    "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SupportingTokens
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702">
        <wsp:Policy>
          <sp:UsernameToken sp:IncludeToken=
              "http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient"/>
        </wsp:Policy>
      </sp:SupportingTokens>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Figure 6.17. WS-SecurityPolicy for plain text UsernameToken

```
      <sp:TransportBinding
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy
">
         <wsp:Policy>
           <sp:TransportToken>
              <wsp:Policy>
                 <sp:HttpsToken RequireClientCertificate="false"/>
              </wsp:Policy>
           </sp:TransportToken>
        </wsp:Policy>
      </sp:TransportBinding>
```

Figure 6.18. A code snippet of WS-SecurityPolicy for HTTPS connection

It is not always possible to secure an end-to-end communication channel by HTTPS, as the link may need to pass through some firewalls hosted in the intermediate nodes. Password digest is another way of securing UsernameToken even over insecure links. This technique uses a digest value computed over a string made up of two other text

values combined with the password. When properly used by both client and server, the combination of these values with the password in the digest makes it possible for the server to verify that the correct password was used when generating the digest. Figure 6.19 lists a code snippet of the WS-SecurityPolicy using a digest password. The <wsp:Policy> element specifies that HashPassword must be used for the client sending requests.

```
<sp:SupportingTokens
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702">
        <wsp:Policy>
          <sp:UsernameToken sp:IncludeToken=
              "http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
            <wsp:Policy>
              <sp:HashPassword/>
            </wsp:Policy>
          </sp:UsernameToken>
        </wsp:Policy>
      </sp:SupportingTokens>
```

Figure 6.19. A code snippet of WS-SecurityPolicy using a digest password

The password digest encryption can ensure message confidentiality, but not integrity and authenticity. On the other hand, HTTPS connection can guarantee confidentiality and integrity. But, its implementation depends on network conditions.

Message signing and encryption is a WS-Security public-key cryptography technique, providing a complete protection to message exchanges over an insecure network. It identifies the private/public key pair that is used to create the signature in each direction and supplies the passwords for accessing the keystore and private key. Obviously, the message signing and encryption technique has the superiority over other techniques on message protection. But, it requires a lot more specifications. The details of message signing and encryption implementation are out of this thesis' scope, and interested readers can refer to Sosnoski's article "Axis2 WS-Security signing and encryption" [161].

The four WS-SecurityPolicy based techniques discussed above provide username and password security for a wide range of different endpoint applications. We generate the corresponding policies, and their implementations are handled by Rampart security module. Users select a proper security technique based on endpoint security requirement

without knowing the implementation details. The code generator will pick up the right security policy and set up the corresponding runtime parameters from the security model.

To validate a user's permission to access an operation using Rampart, we create a callback class PWCBHandler to extract and decrypt the username and password from UsernameToken. The username is then passed to JDBC class UserPassword to validate whether the user has the right to invoke the operation by retrieving the user's persistent data based on RBAC model. If the user is found having the right to use the operation, the stored password is returned to the callback class. In this case, the callback class compares these two passwords from the persistent data and UsernameToken. If they match, the callback class returns a true value. Otherwise, it throws an exception to indicate a security violation error. Figure 6.20 lists the callback class code.

```
package com.endpoint.purchase;

import org.apache.ws.security.WSPasswordCallback;

 * Simple password callback handler. This just checks for UsernameToken information, and
public class PWCBHandler implements CallbackHandler
{
    UserPassword myUserPassword = new UserPassword();
    UserPassword.CredentialType returnPassword;

    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {

        for (int i = 0; i < callbacks.length; i++) {
            WSPasswordCallback pwcb = (WSPasswordCallback)callbacks[i];
            String id = pwcb.getIdentifer();
            returnPassword = myUserPassword.PasswordRetrieval(id);
            if (pwcb.getUsage() == WSPasswordCallback.USERNAME_TOKEN_UNKNOWN) {

                if (!returnPassword.found ||
                        !returnPassword.userPassword.equals(pwcb.getPassword())) {
                    throw new UnsupportedCallbackException(callbacks[i], "check failed");
                }

            } else if (pwcb.getUsage() == WSPasswordCallback.USERNAME_TOKEN)
                // used when hashed password in message
                if (returnPassword.found)
                  pwcb.setPassword(returnPassword.userPassword);
        }
    }
}
```

Figure 6.20. The code of callback class PWCBHandler

On the other end, a client must encrypt the username and password in UsernameToken and send to its endpoint as SOAP message header of an operation request through Axis2 Web service engine. To do this, we need to load a security policy from the classpath first. Figure 6.21a lists the code to load the policy file to an operation API class of a SUT. The loaded policy must be configured by assigning the policy file name, username and password. Figure 6.21b shows how the configuration is done in the example code.

```
 //* Load policy file from classpath.
private static Policy loadPolicy(String name) throws XMLStreamException {
    ClassLoader loader = PoRequestClient.class.getClassLoader();
    InputStream resource = loader.getResourceAsStream(name);
    StAXOMBuilder builder = new StAXOMBuilder(resource);
    return PolicyEngine.getPolicy(builder.getDocumentElement());
}
```

[a] A code snippet to load security policy

```
// configure and engage Rampart
ServiceClient client = stub._getServiceClient();
Options options = client.getOptions();
options.setProperty(RampartMessageData.KEY_RAMPART_POLICY,
    loadPolicy("policy.xml"));
options.setUserName(args[4]);
options.setPassword(args[5]);
client.engageModule("rampart");
```

[b] A code snippet to configure security policy

Figure 6.21. Two code snippets of a SUT client API class

To demonstrate WS-Security UsernameToken in action, we also use TCPMon tool [147] to capture their exchanged messages. The example is based a secured UsernameToken transmission. The user password is encrypted by a digest value computed over a random generated nonce and timestamp, the time at which the sender created the UsernameToken. Figure 6.22 shows the request and response messages sent through using a hash function UsernameToken. Both the nonce and timestamp are in plain text, but the password is encrypted. Comparing to use a plain text UsernameToken to authenticate a user (see Figure 5.18), the username and password in the UsernameToken are both visible. Therefore, we can conclude that anyone who intercepts the traffic still cannot penetrate the endpoint application by reproducing the user's password.

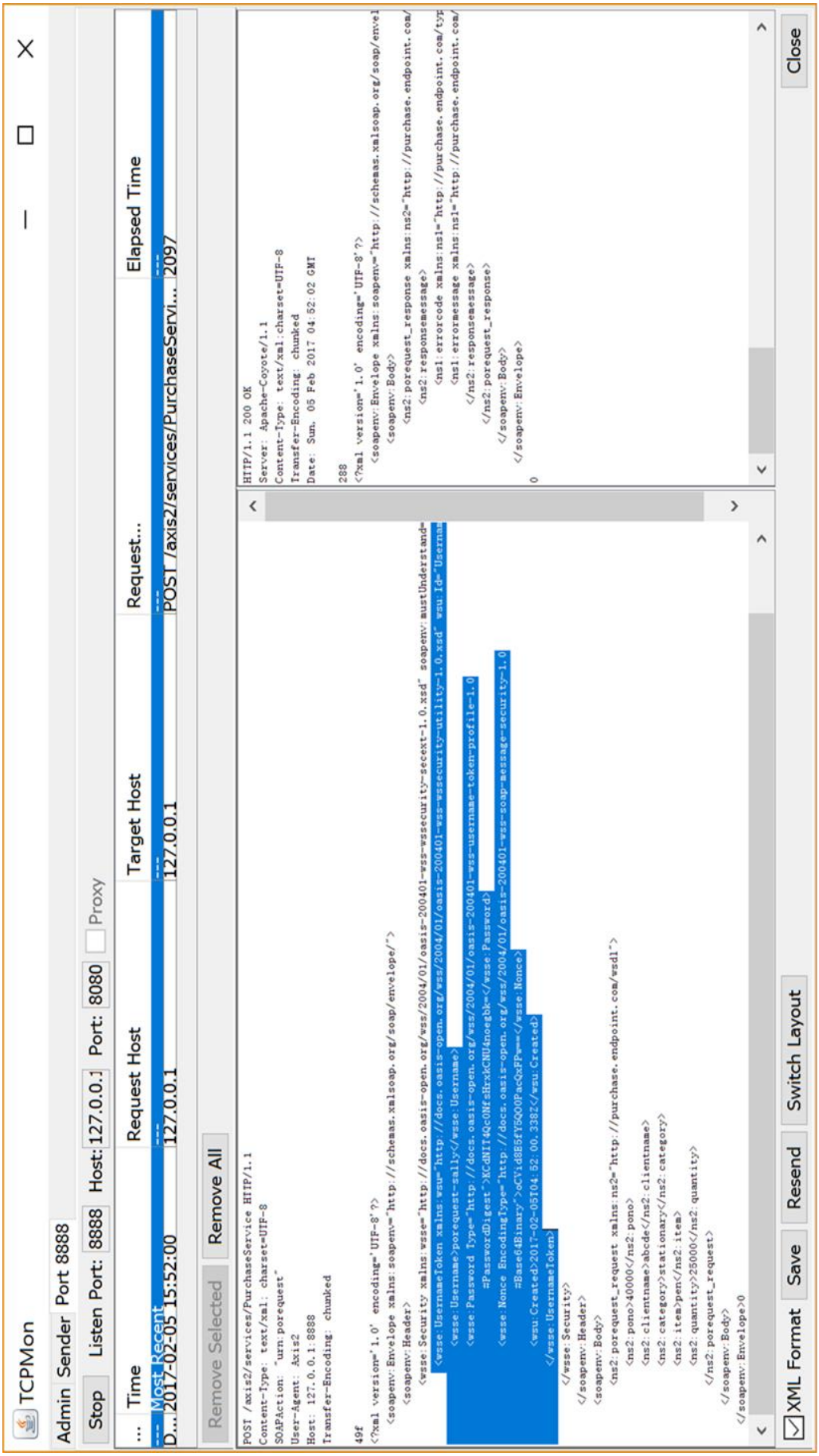


Figure 6.22. Request and response messages with a hash function UsernameToken

To automate the generation of endpoint security testing service, we create the build property file of Ant auto-build tool listed in Figure 6.23. Ant tool assigns the client and server policy files and sets up security related environment for running an endpoint.

```
# set axis-home to your Axis2 installation directory
axis-home=/java/axis2-1.4.1
# set the connection protocol to be used to access services
(http or https)
protocol=http
# set the name of the service host
host-name=localhost
# set the port for accessing the services (change this for
monitoring)
host-port=8080
# set the base path for accessing all services on the host
base-path=/axis2/services/
# set the name of the policy file to be used by the client
client-policy=plain-policy-client.xml
# set the name of the policy file to be used by the server
server-policy=plain-policy-server.xml
```

Figure 6.23. A security modeling build property file for Ant auto-build tool

6.6 Summary

Most enterprise applications in use today, no matter whether they are hosted in-house or at a remote site, have some security requirements on service requests to access their operations. These security requirements are essentially from two orthogonal dimensions of management on organizational business process and protection from systems and data being corrupted.

To model the security aspect of a business process, we propose a hierarchical RBAC model based security metamodel. The metamodel defines function roles to use endpoint operations and division roles to access endpoint resources. To incorporate other security aspects and enhance modeling extensibility, we add a security constraint entity to the metamodel. To ensure data confidentiality, integrity and authenticity, we adopt Rampart security module to implement Web service security standards for securing UsernameToken in transmission.

To make our Security DSVL ease of use, we design six simple and easy use visual constructs for assigning role-permission, user-role and hierarchical role relationships, and specifying other security aspects. To improve endpoint development productivity, we

create an Ant build file to automate all the tasks after endpoint modeling. Our Security DSVL is used to model a typical sale process in a global company. The common business rules of principle of least privilege and avoidance of conflict of interest are obeyed, and username and password are protected from being corrupted by using different security techniques.

As we discussed in Chapter 1, a fully functional endpoint may also have other vertical attributes, such as reliability, performance, etc. These attributes are very different in nature and independent from each other. To help domain experts to model these vertical attributes, our TeeVML should include all the corresponding DSVLs. To develop an vertical attribute DSVL, the proposed four-step process needs to be followed, including (1) decision making to decide whether a DSVL is needed for the attribute, (2) domain analysis to capture all the domain aspects of the attribute and define their inter-relationships; (3) DSVL design to design and develop the modeling language; and (4) implementation to create code generators for converting an attribute model to code.

Due to time constraints, we are not able to develop these additional QoS modeling DSVLs for the time being. We have planned our TeeVML enhancement to add other vertical attributes in our future work.

By now, we have introduced and discussed our DSVLs for modeling endpoint three horizontal layers and a vertical security attribute. In the next chapter, we compare our TeeVML with other two kinds of existing TEE approaches technically. This is followed by a user study to collect software professionals' opinions about our endpoint modeling tool.

CHAPTER 7

Evaluation

In this thesis, we define three key research questions:

RQ1 Can we emulate a functioning integration testing environment capable of capturing all interface defects of an existing or a non-existing system under test from an abstract service model?

RQ2 Would our model-based approach improve testing environment development productivity, compared to using third-generation languages (e.g. Java) to implement endpoints?

RQ3 Can we develop a user centric approach, easy to learn and use to specify testing endpoints by domain experts?

So far, we have introduced our TeeVML and demonstrated how an example endpoint can be modeled by the tool. In this chapter, we evaluate how well the issues related to these research questions have been addressed. We evolve three evaluation criteria from the three research questions respectively:

- **Testing Functionality** (addressing RQ1) – the approach should be able to develop a wide variety of endpoints, which could be used to capture all the interface defects of a new or an existing system under test;
- **Development Productivity** (addressing RQ2) – the approach should have high endpoint development productivity with less development effort and time;
- **Ease of Use** (addressing RQ3) – the approach should be easy to learn and adapt by non-technical background users.

These criteria were first evaluated by a technical comparison of our approach with currently available endpoint emulation approaches. Then, a qualitative comparison was followed to provide subjective ratings to these evaluation criteria and their attributes. This comparison motivated our new model driven DSM approach to address the shortcomings

of the existing approaches. After our approach was ready to use, we also conducted a user survey to evaluate the extent to which our approach was accepted by software testing experts and application developers for the same set of evaluation criteria. We have made some improvements on our early versions of TeeVML based on the feedbacks from the user survey.

7.1 Technical Comparison

Currently, there are two types of approaches to develop SIT environments: specification-based by manual coding (also called "manual coding") and interactive trace data record-and-replay (also called "interactive tracing"). The manual coding approaches are used by IT professionals to develop simplified versions of applications with external behavior manually [12, 13]. They perform this using available knowledge of underlying message syntax, interaction protocol, system behavior and relevant Quality-of-Service (QoS) aspects. The interactive tracing approaches create endpoint models from recorded request-response pairs between the endpoint system and an earlier version of a SUT automatically [48, 106]. Each endpoint's simulated response is generated by finding a closely matched request in the records stored in a trace database.

To compare these two types of approaches with our new TeeVML, we use the three defined evaluation criteria of Testing Functionality, Development Productivity and Ease of Use. We look into what key techniques these approaches adopt to meet the evaluation criteria.

7.1.1 Testing Functionality

Manual Coding: The key motivation of these approaches is to provide SUT performance testing by emulating a large number of endpoints of the same type hosted in a single machine. To achieve this objective, these approaches adopt a light-weight architecture design and some testing features are deliberately neglected [25]. Dynamic protocol behavior with runtime constraints cannot be modeled, as endpoint state transition is triggered by operations only and constraint conditions are ignored. Unless great effort is made, behavior layer functionality will be limited. QoS attributes are generally not considered, except for performance.

Interactive Tracing: To provide SIT, these approaches search for the right request matching on data byte level without any knowledge about upper-level message property

information. They can only tell whether a test is passed or failed but cannot provide any defect information. These approaches are not usable for testing a new application, as its interactive tracing data are not available. Another disadvantage is emulation accuracy, which is subject to the algorithm for finding the correct request in stored records [48].

TeeVML: Our endpoint provides SIT services from signature, protocol, behavior layers and QoS abstraction attributes. Signature layer supports all the applications with RPC communication style to define service operations and parameters; protocol layer can model both static and runtime behaviors by utilizing returned values from behavior model; and behavior layer uses a hierarchical structure dataflow programming for modeling complicated logic implementation. QoS attributes can be easily added to endpoint by modifying SOAP message header.

7.1.2 Development Productivity

Manual Coding: These approaches adopt a modular architecture design, where an endpoint type dependent message engine module is separated from an endpoint type independent network infrastructure and a system configuration module [25]. However, as the message engine is coded using a third-generation language manually, significant amount of development effort is required for each new endpoint type.

Interactive Tracing: An endpoint is created by recording the interactive tracing data between the endpoint and an earlier version of a SUT application. If some testing cases are not covered in the stored trace records, trace recording must be redone. These approaches do not need any endpoint development work, but some effort on testcases preparation and interactive tracing data recording is required.

TeeVML: Our approach models endpoints by layers and attributes, and their models are transformed to executable codes automatically. To improve development productivity, our approach uses high-level abstraction programming constructs, increases components reusability by adopting hierarchical structure designs, and provides a supporting tool for automating endpoint generation from models.

7.1.3 Ease of Use

Manual Coding: To develop an endpoint, developers must have both business domain knowledge and programming skills. Changing and adding new features to the endpoint message engine are very cumbersome and require a lot of coding effort.

Interactive Tracing: Neither business domain knowledge nor programming skills are required. However, developers must have a certain level of understanding of the endpoint application for preparing testcases.

TeeVML: Developers must have business domain knowledge, and some modeling skills. To achieve ease of use, we have applied the principles of Physics of Notations [123] to optimize our visual construct designs and made them more natural and obvious to end users.

7.2 Qualitative Comparison

To qualitatively compare these approaches, we examine them using a set of attributes reflecting more specific aspects of each evaluation criterion. We then give a four-point rating (**N** – not applicable, **L** - low, **M** - medium or **H** - high) representing the level of support these approaches for each attribute. The overall rating of each evaluation criterion summarizes individual attributes' rating and takes their importance into consideration. We list these evaluation attributes and their ratings we give to the three approaches in Table 7.1.

The interactive tracing approaches are given **H** rating for overall Development Productivity and Ease of Use evaluation criteria, as endpoints are created automatically from interactive trace data. Neither any development related techniques nor skill requirements on developers are required to development endpoints. On the downside, these approaches have two key shortcomings in terms of Testing Functionality. One, their use is subject to the availability of interactive tracing data. So, these approaches are not suitable for testing completely new applications, but regression testing for application upgrade. And two, they cannot report defect types and causes information, which are important for application developers to diagnose defects. Therefore, we give these approaches **L** rating for Testing Functionality criterion.

In contrast, both manual coding and our DSM approaches need developers to develop endpoints by using specific techniques and skills. Their Development Productivity and

Ease of Use criteria are rated by comparing and weighting their attributes. Since our approach uses higher level of abstraction models than code to express design intent, we give our approach **M** rating and manual coding **L** rating for these two criteria. Our TeeVML can report both static and dynamic protocol defects, as well as QoS defects, comparing with manual coding approaches reporting static protocol defects only. Therefore, our approach is given **H** rating and manual coding **M** for Testing Functionality criterion.

Table 7.1. Testing environment emulation approaches comparison (**MC**: Manual coding, **IT**: Interactive tracing, **TV**: TeeVML).

Attribute Description	MC	IT	TV
Testing Functionality			
The approach can detect all interface defects.	M	H	H
The approach reports signature defects.	H	N	H
The approach reports static protocol defects.	H	N	H
The approach reports dynamic protocol defects.	L	N	H
The approach reports QoS defects.	N	N	M
Business scenarios can be simulated.	L	N	M
The approach supports both existing and new application testing.	H	N	H
Overall	**M**	**L**	**H**
Development Productivity			
Endpoints are generated automatically.	N	H	N
The approach supports high-level abstraction.	L	N	H
The approach supports components reuse.	M	N	H
The approach has built-in error prevention mechanisms.	H	N	M
Network interface is generated automatically.	L	N	H
Supporting toolset is provided for automating testing service generation.	M	N	H
Overall	**L**	**H**	**M**
Ease of Use			

Special training is needed.	N	L	M
Endpoint application knowledge is required.	L	N	L
Programming skills are required.	L	N	N
Familiar visual notations are used.	N	N	H
Overall	**L**	**H**	**M**

From the technical and qualitative comparisons, we can conclude that interactive tracing approaches develop and deploy SIT environment quick yet cost effectively. But they still need a specification-based tool to specify some endpoints in an enterprise testing environment, as these applications' trace data can be neither available nor usable. As our TeeVML is superior to manual coding approaches in all the three evaluation criteria, it is the preferred complementary tool to interactive tracing approaches.

7.3 User Survey

User surveys incorporate a list of questions to extract specific data from a particular group of people. They provide a comprehensive mechanism for collecting information to describe, compare and explain knowledge, attitudes and behaviors [162]. Survey results are used to improve products' quality and functionality by guiding and correcting the design, development and refinement. We used our survey results to justify the use of our DSM approach and improve the early versions of our TeeVML.

7.3.1 Overview

To define the measurement scales for predicting users' acceptance of a software application, Davis created two specific subjective variables of the perceived Usefulness and perceived Ease of use [163]. In his definition, the Usefulness variable is "the degree to which a person believes that using a particular system would enhance his or her job performance". On the other hand, the Ease of use variable is defined as "the degree to which a person believes that using a particular system would be free of effort". The perceived Usefulness is related to our Testing Functionality evaluation criterion, and the perceived Ease of use corresponds to our Development Productivity and Ease of Use.

We evaluated our TeeVML by measuring these two variables through a two-phase user survey. In the first phase, we conducted a study with testing experts to examine what testing features they valued in endpoints, and what functionality TeeVML should provide

to develop such endpoints. In the second phase, we evaluated TeeVML's usability by collecting software developers' opinions on their experience with the tool to work on an assigned task. We wanted them to compare TeeVML with a third-generation language they were familiar with, as the way manual coding approaches do. We conducted Phase One and Phase Two survey in parallel from January 12, 2016 to April 1, 2016.

This user study, entitled "Evaluation of a domain-specific visual modeling language for enterprise testing environment emulation (TeeVML)", has been approved by Swinburne University Human Research Ethics Committee (SUHREC Project SHR Project 2015/326). The clearance letter from SUHREC is attached as Appendix I.

7.3.2 Questionnaire Design

An important consideration in design our questionnaires is the impact of our own biases. This bias is usually due to our perception and expectation of the answers the survey will provide. To minimise this kind of biases, we develop some ground rules for our questionnaire design:

- Develop neutral questions – to use the wording that does not influence the way the respondents think about the question;
- Use positive and negative question statements alternatively;
- Use multiple-choice questions;
- Pay attention to question order, so that the answer to one does not influence the response to the next.

To ensure the quality of questions, we conducted a pilot study with a test group to evaluate our draft questions. The objectives of the pilot study include:

- To validate that all the questions are understandable;
- To evaluate the reliability and validity of the questions;
- To ensure consistency among the relevant questions;
- To ensure our data analysis techniques matching our expected responses.

The question types include 5-point Likert Scale (5 to 1 representing strongly agree to strongly disagree), single-choice, multiple-choice, and open-ended questions. For the 5-

point Likert Scale questions, in favour responses encompass the answers of either 5 or 4 for a positive question, and 1 or 2 for a negative question. We count the number of in favour responses to measure the degree of acceptance to a particular question statement.

Our survey questionnaires are divided into two sections: participants' demographic information and TeeVML evaluation. To analysis TeeVML as a whole and each functional layer[7] separately, the evaluation section is further split into a summary part and an individual part. The survey results for the demographic section are presented in participant recruitment sections, and the responses from the evaluation section questions are analysed in data analysis sections. We design one questionnaire for each phase of the user survey. Phase One includes 32 questions and they are attached as Appendix II. Phase Two has 45 questions listed in Appendix III.

7.3.3 Phase One

7.3.3.1 Participant Recruitment

To provide expert opinion and valuable recommendations, Phase One participants must have had extensive software testing knowledge. Thus, our target participants are software testing engineers, research students with previous software testing experience and research students in software testing area. We prepared an invitation letter, stressing the usefulness of the user survey to both research and industry practice. The letter was sent to qualified persons through our contacts as an invitation to participate the survey. To be more representative among target population, we invited testing experts from industry and academia, local and oversea. We tried our best to recruit as many testing experts as possible but were only able to recruit 16 participants. This was mainly due to our strict qualification requirement on software testing experience.

Figure 7.1 shows participants' years of experience on IT and software testing, extracted from Phase One survey report. From the diagram, we can see that most participants (94%) have 6 to 16+ years IT experience and 2 to 10 years software testing experience. Most of them are also somewhat familiar or very familiar with SIT. Considering participants' software testing expertise, we have the confidence that the survey result is representative, even with relatively small number of survey participants.

[7] When we conducted the survey, our security DSVL had not been ready to use yet. So, we did not do the survey on security modeling.

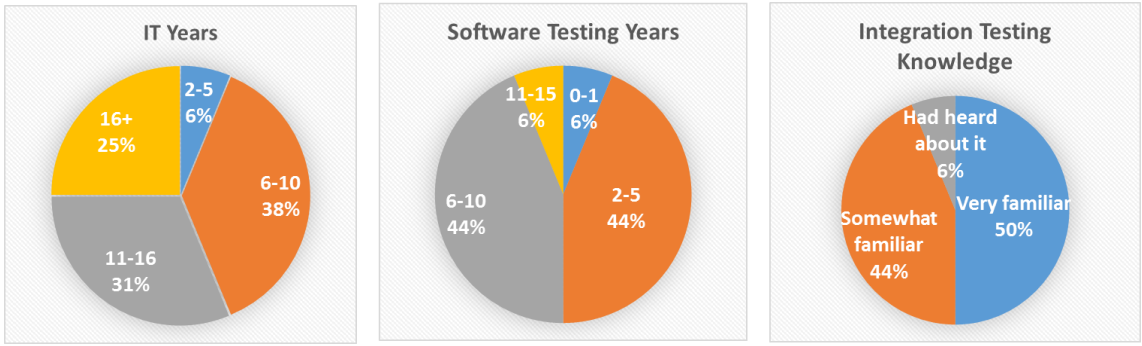


Figure 7.1. Phase One participants IT and software testing experience

7.3.3.2 Experiment Setup

Phase One survey was conducted by one-to-one interview. We used a PowerPoint presentation to explain what testing functionalities are required for an endpoint and how such an endpoint can be modeled by using our TeeVML. During the introduction, we encouraged participants to ask questions, and provided them with answers. If participants' time was allowed, we also showed them a recorded video for demonstrating how an endpoint is modeled. The interview took approximately 45-60 minutes on average.

After the interview, all participants were asked to fill an online questionnaire to give their opinions on each question statement. To clarify any ambiguities of the questions, an instructor was present in case the participants had any questions on the online survey.

7.3.3.3 Survey Results Analysis

To evaluate participants' acceptance of endpoints, we select a number of specific questions from Phase One survey report in Table 7.2 (for better result presentation, we separate the Likert Scale questions from the single and multiple-choice questions) and analyse their responses to these questions. The full report is listed in Appendix IV. We analyse these questions from two different angles: The first angle is about participants' acceptance of an endpoint as a whole and by each interface layer from a testing functionality point of view. The second angle is to find out the possible reasons why participants would consider using (or not using) our endpoints in their future projects.

Testing Functionality

Q8 reflects the overall usefulness of endpoints for conducting SIT. The responses to this question are quite positive with 14 out of 16 (87.5%) participants in favour. This is a good indication of the participants' acceptance of endpoints modeled by TeeVML.

Table 7.2. Questions and responses from Phase One survey report

[a] Likert Scale questions

No	Statement	Frequency				
		5	**4**	**3**	**2**	**1**
Q8	In your opinion, an emulated testing environment is useful for an application inter-connectivity and inter-operability test.	8	6	0	1	1
Q17	It is useful for an emulated testing environment to provide signature testing functionality to its system under test.	7	7	1	1	0
Q21	It is useful for an emulated testing environment to provide interactive protocol testing functionality to its system under test.	12	4	0	0	0
Q25	It is useful for an emulated testing environment to provide interactive behavior testing functionality to its system under test.	6	8	1	1	0
Q30	It is useful for an emulated testing environment to provide non-functional requirement testing features to its system under test.	2	11	3	0	0

[b] Single and multiple-choice questions

No	Question Statement and Choices	Frequency
Q9	What kinds of testing features do you want to see an emulated testing environment provides to system under test for interconnectivity and inter-operability test?	
	Correctness of message signature	13
	Correctness of interactive protocol	16
	Correctness of interactive behavior	14
	Conformance to non-functional requirement	11
	Other	1
Q12	How do you rank the importance to an emulated testing environment[8]?	
	Tool development productivity	4
	Ease of use its development tool	1
	Testing functionality provided to system under test	3
	Result reliability	8
Q13	What are the main motivations for you to use emulated testing environment?	
	Cost saving on application software and hardware investment	14

[8] Only the ranking 1 responses of each choice are listed.

	Effort saving on application installation and maintenance	10
	Lack of application knowledge	5
	Early detection of interface defects	15
Q14	What are your main concerns, which could prevent you from using emulated testing environment?	
	Extra development effort on testing endpoints	6
	Learning a new technology	6
	Inadequate testing functionality	7
	Emulation accuracy	7
	Result reliability	12

Q9 is a multiple-choice question for evaluating the usefulness and completeness of functionality that an endpoint should provide to its SUT. Except for the four features already implemented, we add an "Other" choice for allowing participants to specify any other useful features, our TeeVML does not support. Only one participant selected the choice "Other" and suggested to provide performance testing under different scenarios. From these responses, we can conclude that most participants were satisfied with the endpoint testing functionality modeled by TeeVML.

To further investigate different interface layers, Q17, Q21, Q25 and Q30 are used to get participants' opinion on the usefulness of modeling signature, protocol, behavior and non-functional requirement layers, respectively. All the four questions received quite positive responses. Particularly, we can see that protocol layer (Q21) received in favour responses from all participants. We reckon one of the main reasons why all participants wanted to have protocol testing is that most applications do not have a well-documented protocol specification. Protocol related defects can only be found by conducting SIT, rather than code reviewing or other means. The responses to Q9 confirm this finding that all participants wanted to see the protocol testing feature.

On the other hand, signature layer (Q17) had slightly less in favour response rate compared to protocol layer. The signature correctness is a must for a client to access services provided by a server. However, a few participants might have thought that endpoint signature could be easily coded and verified against product interface specification, hence actual testing would be unnecessary.

Reasons Why Users Use (or not Use) Endpoint

Q13 is a multiple-choice question and lists four reasons why users want to use endpoints. Responses to Q13 indicate that the top reason for using endpoints is early detection of interface errors. While responses to cost saving achieved high rating as well, it was ranked second. This is somewhat a surprising finding. In current practice, SIT is normally conducted during the later stages of software development lifecycle. This is partly because SIT environment is not available before then. If a rapid and cheap solution for testing environment deployment was available, developers might have preferred to conduct at least part of SIT earlier. Early interface defect detection is particularly important, when application development is outsourced to a third-party and its SIT environment is completely inaccessible.

Responses to Q14 indicate that most participants concerned about the reliability of endpoint testing results. We reckon the main reason is that software developers are used to using real applications for their SIT. However, an endpoint is actually a simplified version of its real application. Often, many implementation aspects of the application are neglected and treated as useless for SIT. This might have some impacts on SUT testing results. Our survey results indicate the importance of conducting an endpoint functionality design before modeling it.

7.3.4 Phase Two

7.3.4.1 Participant Recruitment

TeeVML target users are domain experts, most of them have extensive business knowledge but may lack coding skills. For this reason, we design our TeeVML in a way that all coding works are eliminated. On the other hand, to compare our approach with specification-based manual coding approaches, we prefer survey participants to be familiar with at least a third-generation language (for example Java or C#). From the above considerations, we list the qualifications of survey participants below:

- having some knowledge about enterprise applications;
- understanding software development process;
- familiar with a third-generation programming language;
- basic skills on modeling language (e.g. UML) and application modeling.

Using the same recruitment process as Phase One, 19 participants took part in the survey. Figure 7.2 provides the participants background. Most of them have IT background (95%) and 63% are familiar with software modeling.

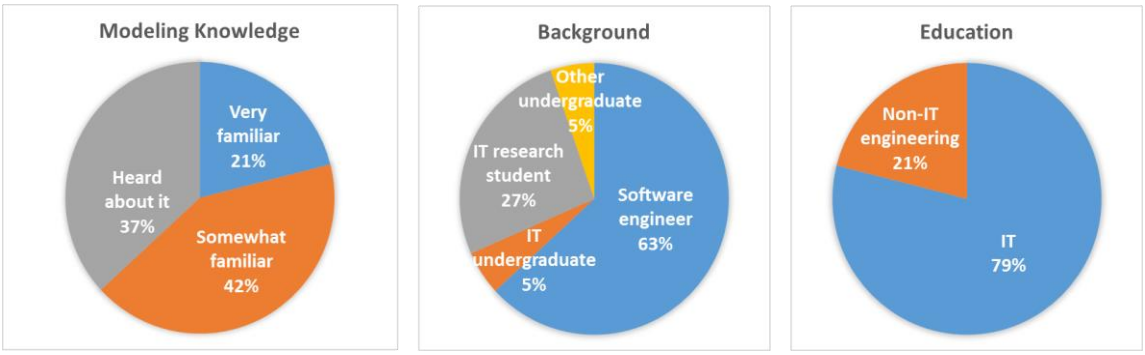


Figure 7.2. Phase Two participants' IT background

7.3.4.2 Experiment Setup

Phase Two survey is a three-step process. First, participants watched a twenty-minute training video. The video introduces TeeVML and shows the steps to model endpoints. By using the recorded video, all the participants are given the exact same training and introduction to the tool for minimizing bias on familiarity with TeeVML. Second, the participants were assigned a task to model the deposit operation of a simplified banking system. The task was performed individually, using TeeVML running in MetaEdit+ 5.1 application hosted in a Windows 10 laptop PC. Finally, all participants were asked to complete an online survey, based on their experience working on the assigned task. The duration for Phase Two was one to one and half hours on average.

In this user study, we only conducted the modeling task but ignored the rest, including the model transformation and testing endpoint generation. The main reason behind this is that this research is focus on modeling endpoints by domain users and other tasks can be done automatically by using a toolset we have developed. In addition, we have to restrict the survey time to be within one and half hours to two hours including the tool introduction and training, as some of the participants could be busy on their daily schedules. If a participant's time was allowed, we showed him/her the steps to transform the model to code and generate a testing endpoint. But, this is out of our user study scope.

7.3.4.3 Survey Results Analysis

Giving that the participants have used our tool to model an example endpoint, we want them to provide their opinions on whether the tool is ease of use and how much endpoint development productivity can be improved. The former uses the 10 questions from System Usability Scale (SUS) to evaluate the tool's ease of use as a whole. To identify the main causes influencing the ease of use criterion, we add a question to capture the actual time spending on the task and a subjective question of comparing our modeling approach with a traditional third-generation language. Phase Two survey result report is attached as Appendix V.

Ease of Use

SUS is a simple, 10 5-point Likert Scale questions to give a global view of subjective assessments of usability [164]. It is often used in carrying out comparisons of usability between systems. Table 7.3 lists the SUS questions, which consist of equal number of positive and negative questions and are arranged alternatively. By doing so, respondents are forced to make an effort to think whether they agree or disagree with each question statement.

SUS yields a single number representing a composite measure of the overall usability of the system being studied. To calculate SUS score, first sum the score contributions from each question. For questions 1, 3, 5, 7, and 9, the score contribution is the scale position minus 1. For questions 2, 4, 6, 8, and 10, the contribution is 5 minus the scale position. Then, multiply the sum of all the individual question scores by 2.5 to obtain the overall SUS score. The overall SUS mean score was calculated as 68 by a statistic study over a large number of products [165].

Table 7.3. System Usability Scale questions

No	Statement
Q12	You would like to use the tool in your future project.
Q13	You found the tool unnecessarily complex.
Q14	You found the tool was easy to use.
Q15	You would need support to be able to use the tool.
Q16	You found the various features of the tool were well integrated.
Q17	You found there was too much inconsistency in the tool.
Q18	You would image that most people would learn to use the tool very quickly.

Q19	You found the tool very cumbersome to use.
Q20	You felt very confident using the tool.
Q21	You needed to learn a lot of things before you could get going with the tool.

Figure 7.3 shows our TeeVML SUS scores for the individual questions. The overall SUS score is calculated as 78.3 out of 100 points, which is equal to 83% from a percentile ranks for raw SUS scores table [165]. From another angle, our SUS score falls between Good and Excellence in the adjective ranges of Acceptability scoring system proposed by Bangor et al. from a study on numerous products [166].

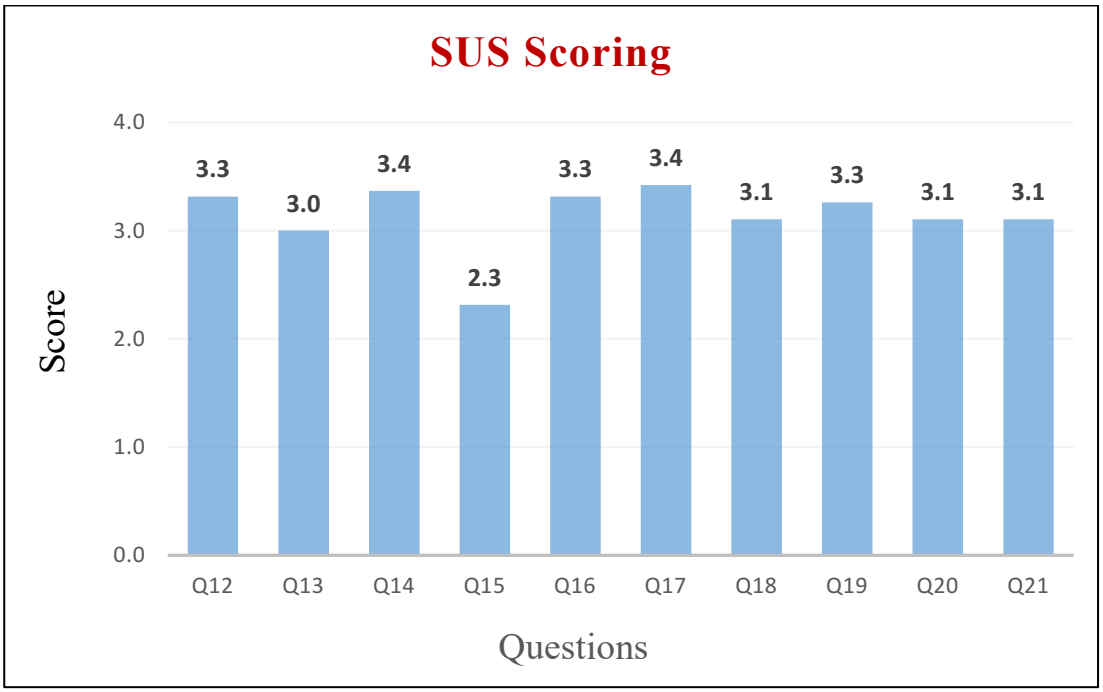


Figure 7.3. The survey results of SUS questions

To look into the details of the induvial questions, Q14 and Q17 received the highest score. Q14 result reflects the participants' recognition of our effort to maximize the cognitive effectiveness of TeeVML visual notations. Each DSVL consists of only a few visual constructs, and shape, colour and textual annotation visual variables are used to discriminate them from each other. For Q17, all the participants accepted our TeeVML architecture design to improve the consistence across different endpoint models. The operations and their parameters defined in signature model are imported and reused by protocol and behavior models. Q15, on the support needed to use the tool, received the lowest score. We reckon this was due to the fact that our introduction video was targeted

toward introduction of the tool and approach in general rather than a stepwise instruction of using the tool. As a result, more participants felt they needed instructor's support. We believe this will be rectified overtime with more usages.

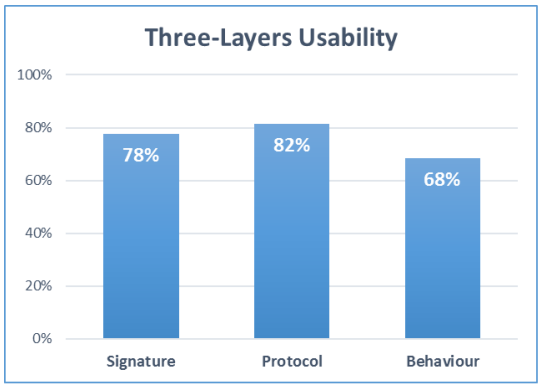
Table 7.4 presents Phase Two survey questions and responses from three functional layers: signature, protocol and behavior, and four usability dimensions of these layers: easy modeling, maintainability, error prevention and completeness. Figure 7.4 drills down TeeVML's usability into functional layers and usability dimensions by summering the percentage of in favour responses from Table 7.4.

From the layers' viewpoint (see Figure 7.4a), protocol layer had the highest usability score and behavior layer received the lowest. This result is agreeing with our expectations. Endpoint protocol modeling is simple and easy, and only four relationship types are used to specify various state transitions. In contrast, behavior modeling must deal with complicated logic processing, involving data manipulation, flow control, persistent data access, etc. For the usability dimension (see Figure 7.4b), maintainability received in favour response from all participants, and was followed by easy modeling. High maintainability is one of the key motivations for us to select a DSM approach, since any changes to an endpoint can be done by modifying its models only and engaging in coding is not required. More than half of participants were not satisfied with the error prevention mechanism provided by TeeVML. Although TeeVML supports most DSVL specific error prevention mechanisms, it does not currently provide comprehensive error and type checking.

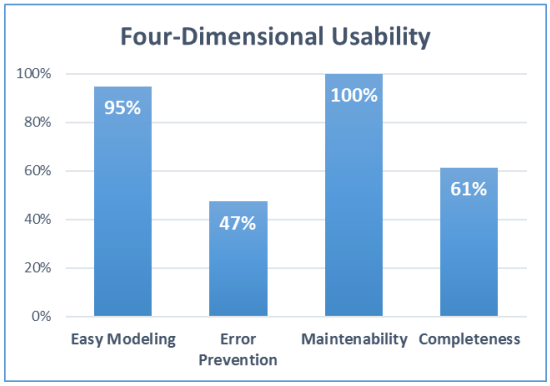
Table 7.4. Questions and responses for functional layers and usability dimensions

No	Question	Frequency				
		5	**4**	**3**	**2**	**1**
Q27	Endpoint signature is easily modeled by the tool.	9	9	0	0	1
Q29	It is easy to make changes to message signature model.	13	6	0	0	0
Q30	It is easy to make errors or mistakes during message signature definition.	0	3	7	5	4
Q31	It is capable of defining all types of message signatures you have seen.	2	11	6	0	0
Q33	Endpoint protocol is easily modeled by the tool.	12	7	0	0	0
Q35	It is easy to make changes to interactive protocol model.	13	6	0	0	0

Q36	It is easy to make errors or mistakes during interactive protocol definition.	1	1	5	6	6
Q37	It is capable of defining all interactive protocol scenarios you have seen.	4	8	6	1	0
Q39	Endpoint interactive behavior is easily modeled by the tool.	3	14	1	1	0
Q41	It is easy to make changes to interactive behavior model.	10	9	0	0	0
Q42	It is easy to make errors or mistakes during interactive behavior definition.	1	1	11	6	0
Q43	The tool has sufficient expressive power for creating behavior model with accurate outputs.	1	9	8	1	0



[a] Functional layers



[b] Usability dimensions

Figure 7.4. In favour responses for different interface layers and usability dimensions

Development Productivity

We have two questions specifically for evaluating TeeVML's productivity to model endpoints. Q9 is for participants to report their actual time spending on modeling the assigned task. Q22 captures participants' idea on how much of their time and effort will be reduced by using TeeVML, comparing with a third-generation language they are familiar with. Table 7.5 presents these two single-choice questions and corresponding responses in percentage.

For Q9, 79% participants could finish their task within 30 minutes, which is a typical endpoint operation modeling. Based on this result, we can generalize that it is possible to model a relatively complex endpoint with more than 10 operations within a day through using our tool support for TEE. From Q22 we can see that more than half of respondents (57.8%) agreed that using TeeVML would reduce "50% - 80%" or "80%+" of the time duration they use for endpoint development. No participant voted "Almost the same". As

a result, we can conclude that most participants agree that our TeeVML could increase endpoint development productivity.

Table 7.5. Endpoint modeling productivity questions and responses

No	Question Statement	%
Q9	How long did it take you to complete the task?	
	10 – 15 minutes	5.2
	16 – 20 minutes	21.0
	21 – 25 minutes	36.8
	26 – 30 minutes	15.7
	30+ minutes	21.0
Q22	In your opinion, comparing to a third generation language (e.g. Java) you are familiar with, how much would a typical endpoint development effort be reduced by using the tool?	
	Almost the same	0.0
	10 – 25%	10.5
	26 – 50%	31.5
	51 – 80%	47.3
	81%+	10.5

7.3.5 Open-Ended Questions

So far, we have analysed our user study results for the close-ended questions. This type of questions is conclusive in nature and is designed to generate survey results that are easily quantifiable. The formal form of the information from close-ended questions allows researchers to group responses into categories based on the options respondents have selected. However, a key drawback of close-ended questions is that researchers must already have a clear understanding of the topic of the questions and provide a complete list of options for respondents to select from. In our case for example, it is difficult for us to provide an exhausted list including all non-functional requirements.

In contrast, open-ended questions are exploratory in nature and allow respondents to provide answers without forcing them to select from pre-defined options. Therefore, open-ended questions provide rich qualitative data to researchers with opportunities to gain insight on all the opinions on a topic they are not familiar with. However, a key

disadvantage of this type of questions is the lack of the statistical significance needed for a conclusive research. To compromise these two types of questions, one solution is to add one more option Other to multiple-choice questions and allows respondents to enter their own answers to these questions. By doing so, survey results can be analysed based on the listed options statistically, and any unpredicted answers will be studied individually for gathering additional information.

In this user study, the participants must meet the strict requirements on qualification and working experience in IT fields. Most of them are the experts on either software application testing or development, or both. Therefore, we should encourage them to give us experts' advices on what/how such a modeling tool should be developed. From this consideration, we use some open-ended questions and multiple-choice questions with an added Other option to catch new ideas for us. By using these open-ended questions, we received many valuable comments and feedbacks. We evaluated these comments and feedbacks and some of them have been used to enhance our current version TeeVML. Others will be our future work for the next version.

Table 7.6 lists some of the open-ended questions, participants' comments and feedbacks. We briefly discuss our solutions in response to these comments and feedbacks.

7.4 Summary

This chapter describes the evaluation process of our TeeVML to model endpoints against the three evaluation criteria evolved from our research questions. We conducted a technical comparison and provided qualitative ratings with other two types of existing approaches. This was followed by a user study with software testing experts and developers.

Our technical comparison is based on what key techniques the approaches are adopted to meet the evaluation criteria. We compare their advantages and disadvantages and conclude the ratings among the three kinds of TEE approaches. Our DSM approach is superior to specification-based manual coding approaches but performs not as well as interactive tracing approaches for both development productivity and ease of use. So, our approach is suitable for either emulating a simple testing environment with a small number of endpoint types or providing a complementary means to those interactive tracing approaches.

Table 7.6. Open-ended questions, feedbacks and solutions

Question Statement	Comment/Feedback	Solution
Phase One		
Q9: What kinds of testing features do you want to see an emulated testing environment provides to system under test for interconnectivity and inter-operability test?	If possible, performance under different environment (such as idle, normal, heavy loaded).	Current version TeeVML does not support performance testing. As we use Axis2 SOAP engine, performance test can be easily added to endpoints by modifying SOAP message header. The performance testing feature will be added to the next version.
Q15: Is there anything emulated testing environment does not let you do that you would like to?	Its behavior expressive power may not good enough to handle complex logic processes.	Our current Behavior DSVL consists of a few simple visual constructs for basic logic implementations. We will improve the behavior expressiveness through adding object-oriented programming features in the next version.
	Signature parameter value range.	Axis2 can only verify operations and their parameter types, but not parameter value range. We have created a signature code generator to generate a Java class for verifying parameter value ranges.
	Signature modeling should support other types of transportation protocol, rather than just RPC.	The current TeeVML version uses WSDL 1.1 specification as its signature metamodel. It does not support RESTful communication architecture. In the next version, we will use WSDL 2.0 to support both RPC and RESTful.
	I am not sure whether the language and tool have the ability to simulate not only the expected behavior but also the unexpected (abnormal) cases which can be used to test the reliability and robustness.	The current TeeVML version does not support robustness QoS attribute. In the next version, we will add the robustness attribute to endpoint for simulating varieties of endpoint faulty scenarios.

Q20: Your comment on signature modeling	High productivity is very important, as a typical endpoint may have many services and each service may have many elements.	We have paid special attention to increase signature modeling productivity. The key solution is the two-level hierarchical reusability for reusing parameters and messages. Often, users do not need to create a new parameter or message, but choose an existing one.
Q23: What interactive protocol testing functionality should an emulated testing environment have?	To support concurrent accesses.	Current TeeVML version cannot model endpoint supporting concurrent accesses. In the next version, we may support multiple threads endpoint development, each of them handling one SUT access.
Phase Two		
Q25: Is there anything the tool does not let you do that you would like to?	Syntax checking; type checking	Current TeeVML version has some general domain rules to prevent certain kinds of modeling errors from human mistakes. However, as TeeVML is a visual language, it does not support type checking. For the next version, we will consider to develop a DSL for users to specify all modeling rules, and models will be verified before transforming to code.
	A more productivity approach to create the models. Like text file based model creation. The UI based solution is intuitive and good for elementary user however once the user is familiar with the tool, the text file based solution may be more productive.	Good suggestion. We will investigate the benefits from the use of textual language and compare the pros and cons of visual vs. textual languages.
	Is it possible to integrate the testcase execution and result verification into the framework? Will it provide extension mechanism such as plugin for user to	Good suggestion. We will conduct a feasibility study for these two features for our next version TeeVML.

	define their own modeling notations such as behaviors not pre-defined?	
Q.45: Your comment on usability of each interface layer sub-language of the TeeVML tool.	More logic processing constructs may need.	Endpoint is a simplified version of a real application, and many complicated internal logic implementations are neglected. To improve behavior expressive power, we may need to extend TeeVML with object-oriented programming constructs.
	It would be good to provide some templates or samples that are easy to start with.	Good suggestion. We will provide templates and samples in our help document.
	The layers make the logic clear, however, it may result productivity issues since the user has to open several windows to reach the variable want to modify.	We should balance the top-level diagram view complexity and the depth of sub-diagrams. On the other hand, the hierarchical structure will improve components reusability and increase productivity.

To measure the two variables of the perceived Usefulness and perceived Ease of use, we conducted our survey in two phases to interview software testing experts and assign a modeling task to software developers. From Phase One survey results we can see that most participants accepted the endpoints developed by our TeeVML from the functionality point of view. For endpoint interface layers, protocol layer is ranked highest receiving in favour responses from all participants.

Phase Two questionnaire is divided into two groups for evaluating usability and productivity. To have a global view of TeeVML's usability, we use 10 SUS questions and achieve a good overall SUS score 78.3. TeeVML's productivity is evaluated by a question of actual time spending on the assigned task and a subjective question comparing with a third-generation language. The responses from the former indicate that the tool is a quick and effective solution to model endpoints, and the latter show that it has a significant improvement on productivity comparing with specification-based manual coding.

To compare the usability with specification-based manual coding approaches, we recruited software developers to take part in Phase Two, instead of TeeVML's target user -- domain experts. As coding is not required for the assigned modeling task, software developers will not gain any advantage in doing the survey over domain experts. To assess to what extent programming skill will affect the survey results, we have further analyzed the SUS scores from two groups of participants. The first group consists of 12 software engineers, who have deep knowledge of software development. The second group includes the rests, and they are mainly IT research students. From programming skill point of view, the participants in the second group are not as good as those in the first group, but they achieved a slightly better average SUS result than the first group (79.2 vs. 77.8). From the survey result, we would anticipate the same result, if the survey were done by domain experts.

CHAPTER 8

Conclusions and Future Work

This thesis introduces our new model-driven domain-specific approach to TEE. We have discussed the full life-cycle development process of the approach in details. Also, we have described an evaluation of the approach by a technical comparison with other existing approaches and a user study for collecting IT professionals' opinions. This chapter brings this thesis to the end. We summarise the work being done to implement our approach and recommend future work for quality improvement and functionality enhancement.

8.1 Conclusions

During our research and approach development, we were specially paying our attentions to the three key research questions and their sub-questions raised in Chapter 1. To conclude this thesis, we briefly describe how the issues related to these research questions are addressed by our approach.

Research Question 1: *Can we emulate a functioning integration testing environment capable of capturing all interface defects of an existing or a non-existing system under test from an abstract service model?*

RQ1.1 Do the endpoints, developed by our approach, support both existing and new enterprise application SIT?

Endpoints are modeled rather than generated from the trace records of earlier versions of applications. Thus, our approach supports both existing and new application testing.

RQ1.2 Do the endpoints, developed by our approach, report all types of signature defects?

Service requests and their parameter types are validated by Axis2 SOAP engine, generated from the endpoint signature model. A Java class checks the upper and lower limits of signature parameters.

RQ1.3 Do the endpoints, developed by our approach, report all types of protocol defects, including static and dynamic defects?

Static protocol defects are captured based on the endpoint state from an FSM; dynamic protocol defects are detected from an EFSM with runtime constraints, which are based on the returned values from the behavior model.

RQ1.4 Do the endpoints, developed by our approach, report QoS defects, such as security defects?

Current version supports security defects detection from a RBAC model as a proof of concept. The approach has a built-in mechanism for adding in other QoS attributes easily.

RQ1.5 Can the endpoints, developed by our approach, simulate protocol scenarios, including time event, synchronous and unsafe operations?

The state entity of Protocol DSVL has properties to specify timeout of a state, synchronous and unsafe operations.

Research Question 2: *Would our model-based approach improve testing environment development productivity, compared to using third-generation languages (e.g. Java) to implement endpoints?*

RQ2.1 Does our approach support a higher-level abstraction beyond programming?

The designed DSVLs consist of high-level abstraction visual constructs for users to model endpoints.

RQ2.2 Does our approach support components reuse within a DSL and across DSLs?

Operations and parameters defined in signature model are reused by protocol and behavior models. Same behavior model nodes can be used in different components.

RQ2.3 Can our approach provide error prevention mechanisms embedded in DSLs?

Domain rules are applied to endpoint modeling, such as relationships among entities, entity types, entity properties definitions, etc.

RQ2.4 Does our approach automate endpoint generation process from models?

A building tool is provided to generate testing service from endpoint models automatically.

Research Question 3: *Can we develop a user centric approach, easy to learn and use to specify testing endpoints by domain experts?*

RQ3.1 Can we develop an approach that only uses problem domain concepts?

Yes, all visual constructs of the approach DSVLs are related to problem concepts.

RQ3.2 Can we develop an effective and usable approach that does not need any programming work?

Users develop endpoints by modeling layers and attributes, and code generators transform models to codes automatically.

RQ3.3 Can we develop effective and usable endpoint modeling DSLs using visual notations?

Yes, all the approach DSVLs use visual notations.

RQ3.4 Do our DSL visual notations support acceptable cognitive effectiveness?

All the visual constructs are designed based on the principles of Moody's Physics of Notation.

8.2 Future Work

To further improve our DSM approach to TEE in the future, we recommend some enhancements of the current version below:

Modeling Other QoS Attributes – Other than security, an endpoint may also have other QoS attributes, such as performance, robustness, reliability, portability and others. Specifically, it is desirable to simulate multiple instances of a same endpoint type to test a SUT's performance. Furthermore, an endpoint may purposely inject some errors into its response messages to test how a SUT can handle these faulty responses. We recommend giving priority to the performance and robustness QoS attributes.

Improving Behavior Expressiveness – Object-oriented programming has higher expressive power than imperative and procedural programming by supporting inheritance, polymorphism, encapsulation, etc. Making our Behavior DSVL object-oriented can simplify behavior modeling, increase development productivity and output accuracy, and have a better diagrammatic view of behavior model. Furthermore, to reduce modeling overhead in effort and time, some special purpose utility nodes should be provided with Behavior DSVL for common modeling features.

Model Syntax Checking DSL – Users may make mistakes during endpoint modeling. These mistakes include wrong associations of entity types, improper use of entity ports and cardinalities in relationships, incorrect use of entities, incorrect definitions of entity properties, and many others. It would be good, if there were a centralized DSL and a corresponding model to specify all the domain rules and view the definitions within a model and across models. Models are verified by this syntax checking model before transforming them to codes.

Endpoint Generation by Reverse Engineering -- Reverse engineering is the process of extracting knowledge or design information from a working application. The information can be used as inputs to a DSL for users to make necessary modifications before generating endpoint models. Obviously, one of the key advantages is that users do not have to have the knowledge of the applications to be emulated. Another one is the productivity improvement of endpoint modeling, many modeling entity instances and their relationships are already in the models. These are particularly important if we are going to emulate a large-scale distributed testing environment.

Testcase Generation – To conduct SIT, testcases are needed to cover a variety of business scenario and boundary conditions. Ad hoc approach to create testcases could result in inadequate testing and some interface defects could be undetectable. As we have the developed endpoint models already, a testcase code generator can be developed to transform endpoint models to testcases. With a support tool, SIT can be executed automatically with the testcases as inputs.

Executable Documentation – In a traditional software development process, software developers spend tremendous effort to code software programs based on the system requirements and design documents. Wouldn't it be nice if these documents are actually

executable and all the coding works can be skipped? We have already achieved the full code generation from endpoint models by using problem domain visual constructs. By generalizing our applicable domain beyond software testing, we can create an executable documentation approach to develop application prototypes or even production applications.

REFERENCES

[1] Accenture, "Accenture Technology Vision 2015," 2015.

[2] P. G. Neumann, "The Crash of the AT&T Network in 1990," 1990.

[3] E. Dustin, "Effective Software Testing: 50 Ways to Improve Your Software Testing," Addison-Wesley Longman Publishing Co., Inc., 2002.

[4] M. Pawar, R. Patel, and N. Chaudhari, "Survey of Integrating Testing For Component-based System," International Journal of Computer Applications, vol. 57, 2012.

[5] R. Yadav, "Oracle PeopleSoft Enterprise Financial Management 9.1 implementation an exhaustive resource for PeopleSoft financials application practitioners to understand core concepts, configurations, and business processes," ed: Birmingham: Packt Pub., 2011.

[6] T. Wong, "Salesforce.com For Dummies," 4th ed., Hoboken: Wiley, 2010.

[7] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "I/O Devices on VMware Workstation Hosted Virtual Machine Monitor," presented at the the General Track: USENIX Annual Technical Conference, 2001.

[8] J. Watson, "Virtualbox: bits and bytes masquerading as machines," Linux Journal, 2008.

[9] P. B. Gibbons, "A Stub Generator for Multilanguage RPC in Heterogeneous Environments," IEEE Transactions on Software Engineering, vol. 13, pp. 77-87, 1987.

[10] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, objects," presented at the In Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, Canada, 2004.

[11] J. Michelsen, "Key Capabilities of a Service Virtualization Solution," 2011.

[12] C. Hine, J.-G. Schneider, J. Han, and S. Versteeg, "Scalable emulation of enterprise systems," in Software Engineering Conference, Australian, pp. 142-151, 2009.

[13] J. Yu, J. Han, J.-G. Schneider, C. Hine, and S. Versteeg, "A virtual deployment testing environment for enterprise software systems," presented at the Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures, Italy, 2012.

[14] J. Liu, J. Grundy, I. Avazpour, and M. Abdelrazek, "TeeVML: Tool Support for Semi-Automatic Integration Testing Environment Emulation," presented at the

IEEE/ACM International Conference on Automated Software Engineering, Singapore, 2016.

[15] D. C. Schmidt, "Model-driven engineering," COMPUTER-IEEE COMPUTER SOCIET*,* vol. 39, p. 25, 2006.

[16] OMG, "Meta Object Facility (MOF) Specification," ed: The Object Management Group, 2000.

[17] A. Coen-Porisini, M. Pradella, M. Rossi, and D. Mandrioli, "A formal approach for designing CORBA-based applications," ACM Transactions on Software Engineering and Methodology (TOSEM)*,* vol. 12, pp. 107-151, 2003.

[18] MetaCase, "MetaEdit+ Domain-Specific Modeling (DSM) environment," Available: http://www.metacase.com/products.html, 2017

[19] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, "EMF: eclipse modeling framework", Pearson Education, 2008.

[20] D. Jayasinghe, *"Quickstart apache axis2"*, Packt Publishing Ltd, 2008.

[21] A. Vukotic and J. Goodwill, "Apache Tomcat 7", Apress, 2011.

[22] E. Christensen, F. Curbera, and G. Meredith, "Web Services Description Language (WSDL) 1.1. W3C," Note 15, www. w3. org/TR/wsdl, 2001.

[23] R. Thurlow, "RPC: Remote Procedure Call Protocol Specification Version 2," ed: The Internet Engineering Task Force, 2009.

[24] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," Information and Software Technology*,* vol. 43, pp. 841-854, 2001.

[25] C. Hine, "Emulation of Enterprise Software Environments," Ph.D. Thesis, Swinburne University of Technology, 2012.

[26] J. Liu, J. Grundy, I. Avazpour, and M. Abdelrazek, "A Domain-Specific Visual Modeling Language for Testing Environment Emulation," presented at the IEEE Symposium on Visual Languages and Human-Centric Computing, Cambridge, UK, 2016.

[27] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," presented at the Proceedings of the General Track: USENIX Annual Technical Conference, 2001.

[28] J. Watson, "VirtualBox: bits and bytes masquerading as machines," Linux J.*,* 2008.

[29] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho*, et al.*, "Xen and the art of virtualization," presented at the Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA, 2003.

[30] J. E. Smith and R. Nair, "The Architecture of Virtual Machines," Computer*,* vol. 38, pp. 32-38, 2005.

[31] J. Sanchez, "Squeezing Virtual Machines Out of CPU Cores," Available: http://www.vminstall.com/virtual-machines-per-core/, 2009.

[32] I. Turner-Trauring, "Write test doubles you can trust using verified fakes," Available: https://codewithoutrules.com/2016/07/31/verified-fakes/, 2016

[33] W. Bulaty, "Stubbing, Mocking and Service Virtualization Differences for Test and Development Teams," Available: https://www.infoq.com/articles/stubbing-mocking-service-virtualization-differences, 2016.

[34] P. B. Gibbons, "A stub generator for multilanguage RPC in heterogeneous environments," IEEE Transactions on Software Engineering, vol. 13, p. 77, 1987.

[35] Mockito, "Tasty mocking framework for unit tests in Java," Available: http://site.mockito.org/, 2016.

[36] RSpec, "Behaviour Driven Development for Ruby," Available: http://rspec.info, 2016.

[37] Mockery, "*Mocker,"*. Available: https://github.com/padraic/mockery, 2016.

[38] M. Utting and B. Legeard, "Practical model-based testing: a tools approach," Morgan Kaufmann, 2010.

[39] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," Software Testing, Verification and Reliability, vol. 22, pp. 297-312, 2012.

[40] I. Jacobson, G. Booch, and J. Rumbaugh, "The unified software development process," Addison-Wesley Longman Publishing Co., Inc., 1999.

[41] I. Schieferdecker, Z. R. Dai, J. Grabowski, and A. Rennoch, "The UML 2.0 testing profile and its relation to TTCN-3," in Testing of Communicating Systems, ed: Springer, pp. 79-94, 2003.

[42] J. Pardillo, "A systematic review on the definition of UML profiles," presented at the Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I, Oslo, Norway, 2010.

[43] J. A. Whittaker and M. G. Thomason, "A Markov chain model for statistical software testing," IEEE Transactions on Software engineering, vol. 20, pp. 812-824, 1994.

[44] S. J. Prowell, "TML: A description language for Markov chain usage models," Information and Software Technology, vol. 42, pp. 835-844, 2000.

[45] J. Tretmans, "Model based testing with labelled transition systems," in Formal methods and testing, ed: Springer, pp. 1-38, 2008.

[46] A. Abouzahra, J. Bézivin, M. D. Del Fabro, and F. Jouault, "A practical approach to bridging domain specific languages with UML profiles," in Proceedings of the Best Practices for Model Driven Software Development at OOPSLA, 2005.

[47] J. Michelsen and J. English, "Capabilities of Service Virtualization Technology," in *S*ervice Virtualization: Reality is Overrated, ed Berkeley, CA: Apress, , pp. 37-45, 2012.

[48] M. Du, J.-G. Schneider, C. Hine, J. Grundy, and S. Versteeg, "Generating service models by trace subsequence substitution," presented at the Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures, Canada, 2013.

[49] M. Du, S. Versteeg, J.-G. Schneider, J. Han, and J. Grundy, "Interaction Traces Mining for Efficient System Responses Generation," SIGSOFT Softw. Eng. Notes, vol. 40, pp. 1-8, 2015.

[50] J. William T. McCormick, P. J. Schweitzer, and T. W. White, "Problem Decomposition and Data Reorganization by a Clustering Technique," Operations Research, vol. 20, pp. 993-1009, 1972.

[51] J. C. Bezdek and R. J. Hathaway, "VAT: A tool for visual assessment of (cluster) tendency," in Proc. IJCNN, pp. 2225-2230, 2002.

[52] K. Jensen, L. Kristensen, and L. Wells, "Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems," International Journal on Software Tools for Technology Transfer*,* vol. 9, pp. 213-254, 2007.

[53] J. Grundy, Y. Cai, and A. Lui, "SoftArch/MTE: generating distributed system test-beds from high-level software architecture descriptions," ed., 2005

[54] Pact, "Enables consumer driven contract testing," Available: https://github.com/realestate-com-au/pact, 2017.

[55] J. Han, "A comprehensive interface definition framework for software components," in 5th Asia Pacific Software Engineering Conference, pp. 110-117, 1998.

[56] A. Beugnard, J.-M. J, l. Plouzeau, and D. Watkins, "Making Components Contract Aware," Computer, vol. 32, pp. 38-45, 1999.

[57] J. Nestor, W. A. Wulf, and D. A. Lamb, "IDL, Interface Description Language," 1981.

[58] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat, "An efficient implementation of Java's remote method invocation," in ACM Sigplan notices, pp. 173-182, 1999.

[59] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI," IEEE Internet computing*,* vol. 6, p. 86, 2002.

[60] W3C, "HTTP - Hypertext Transfer Protocol," Available: https://www.w3.org/Protocols/, 2006.

[61] J. Klensin, "Simple Mail Transfer Protocol," Available: https://tools.ietf.org/html/rfc5321, 2008.

[62] R. T. Fielding, "Architectural styles and the design of network-based software architectures," University of California, Irvine, 2000.

[63] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, "XML schema part 1: structures second edition," ed: W3C Recommendation, 2004.

[64] S. Seely, "Understanding WS-Security," Available: https://msdn.microsoft.com/en-us/library/ms977327.aspx, 2002.

[65] T. Karthikeyan and J. Geetha, "Contract First Design: The Best Approach to Design of Web Services," International Journal of Computer Science & Information Technologies, vol. 5, 2014.

[66] M. Samek, "A crash course in UML state machines," 2009.

[67] V. Alagar and K. Periyasamy, "Extended finite state machine," in Specification of Software Systems, ed: Springer, pp. 105-128, 2011.

[68] O. Nierstrasz, "Regular types for active objects," presented at the Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, Washington, D.C., USA, 1993.

[69] B. Selic, "Using UML for modeling complex real-time systems," in Languages, Compilers, and Tools for Embedded Systems. vol. 1474, F. Mueller and A. Bestavros, Eds., ed: Springer Berlin Heidelberg, pp. 250-260, 1998.

[70] L. De Alfaro and T. A. Henzinger, "Interface automata," ACM SIGSOFT Software Engineering Notes, vol. 26, pp. 109-120, 2001.

[71] U. Endriss, N. Maudet, F. Sadri, and F. Toni, "Protocol conformance for logic-based agents," in IJCAI, pp. 679-684, 2003.

[72] H. Wehrheim and R. H. Reussner, "Towards more realistic component protocol modelling with finite state machines," UNU-IIST, pp. 27, 2006.

[73] Y. Moffett, J. Dingel, and A. Beaulieu, "Verifying Protocol Conformance Using Software Model Checking for the Model-Driven Development of Embedded Systems," IEEE Transactions on Software Engineering, vol. 39, 2013.

[74] F. Plasil, S. Visnovsky, and M. Besta, "Bounding component behavior via protocols," in Technology of Object-Oriented Languages and Systems, TOOLS 30 Proceedings, pp. 387-398, 1999.

[75] Y. Jin and J. Han, "Specifying Interaction Constraints of Software Components for Better Understandability and Interoperability," in COTS-Based Software Systems. vol. 3412, X. Franch and D. Port, Eds., ed: Springer Berlin Heidelberg, pp. 54-64, 2005.

[76] J. Yu, J. Han, S. O. Gunarso, and S. Versteeg, "A Business Protocol Unit Testing Framework for Web Service Composition," In 25th International Conference on Advanced Information Systems Engineering, Spain, 2013.

[77] C. Hine, J.-G. Schneider, J. Han, and S. Versteeg, "Modelling Enterprise System Protocols and Trace Conformance," presented at the Proceedings of the 21st Australian Software Engineering Conference, 2010.

[78] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, K. Govindarajan, A. Karp*, et al.*, "Web services conversation language (wscl) 1.0," W3C Note, vol. 14, 2002.

[79] A. Barros, M. Dumas, and P. Oaks, "A critical overview of the web services choreography description language," 2005.

[80] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," ACM SIGSOFT Software Engineering Notes, vol. 31, pp. 1-38, 2006.

[81] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson, "Behavioral interface specification languages," ACM Computing Surveys (CSUR), vol. 44, pp. 16, 2012.

[82] R. Floyd, "Assigning Meanings to Programs," in Program Verification, ed: Springer Netherlands, pp. 65-81, 1993.

[83] C. A. R. Hoare, "An axiomatic basis for computer programming," Commun. ACM*,* vol. 12, pp. 576-580, 1969.

[84] C. Morgan, "Programming from specifications," Prentice-Hall, Inc., 1990.

[85] X. Fu, T. Bultan, and J. Su, "Analysis of interacting BPEL web services," in Proceedings of the 13th international conference on World Wide Web, pp. 621-630, 2004,

[86] M. von Rosing, S. White, F. Cummins, and H. de Man, "Business Process Model and Notation—BPMN," 2015.

[87] T. B. Sousa, "Dataflow Programming Concept, Languages and Applications," in Doctoral Symposium on Informatics Engineering, 2012.

[88] J. C. Grundy, J. Hosking, K. N. Li, N. M. Ali, J. Huh, and R. L. Li, "Generating domain-specific visual language tools from abstract visual specifications," Software Engineering, IEEE Transactions on*,* vol. 39, pp. 487-515, 2013.

[89] N. Instruments. "Introduction to LabVIEW," Available: http://www.ni.com/getting-started/labview-basics/, 2017

[90] A. Fukunaga, W. Pree, and T. D. Kimura, "Functions as objects in a data flow based visual language," in Proceedings of the 1993 ACM conference on Computer science, pp. 215-220, 1993.

[91] P. T. Cox, F. R. Giles, and T. Pietrzykowski, "Visual object-oriented programming," in Visual object-oriented programming, ed: Manning Publications Co., 1995, pp. 45-66, 1995.

[92] B. Meyer, "Applying'design by contract'," Computer, vol. 25, pp. 40-51, 1992.

[93] J. Bau and J. C. Mitchell, "Security modeling and analysis," IEEE Security & Privacy, vol. 9, pp. 18-25, 2011.

[94] J. Bézivin, "In search of a basic principle for model driven engineering," Novatica Journal, Special Issue, vol. 5, pp. 21-24, 2004.

[95] D. Basin, J. Doser, and T. Lodderstedt, "Model driven security," in Engineering Theories of Software Intensive Systems, ed: Springer, pp. 353-398, 2005.

[96] L. Lucio, Q. Zhang, P. H. Nguyen, M. Amrani, J. Klein, H. Vangheluwe, *et al.*, "Advances in Model-Driven Security," Advances in Computers, vol. 93, pp. 103-152, 2014.

[97] D. F. Brewer and M. J. Nash, "The chinese wall security policy," in Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on, pp. 206-214, 1989.

[98] J. McLean, "A comment on the 'basic security theorem'of Bell and LaPadula," Information Processing Letters, vol. 20, pp. 67-70, 1985.

[99] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST standard for role-based access control," ACM Trans. Inf. Syst. Secur., vol. 4, pp. 224-274, 2001.

[100] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, *et al.*, "Guide to attribute based access control (ABAC) definition and considerations," NIST Special Publication, vol. 800, 2013.

[101] R. S. Sandhu, E. J. Coynek, H. L. Feinsteink, and C. E. Youmank, "Role-based access control models" IEEE computer, vol. 29, pp. 38-47, 1996.

[102] J. Jürjens, "UMLsec: Extending UML for secure systems development," in International Conference on The Unified Modeling Language, pp. 412-425, 2002.

[103] T. Lodderstedt, D. Basin, and J. Doser, "SecureUML: A UML-based modeling language for model-driven security," in International Conference on the Unified Modeling Language, pp. 426-441, 2002.

[104] J. B. Warmer and A. G. Kleppe, "The Object Constraint Language: Precise Modeling With Uml," ed: Addison-Wesley Object Technology Series, 1998.

[105] A. C. O'Connor and R. J. Loomis, "2010 Economic Analysis of Role-Based Access Control," 2010.

[106] D. L. Giudice, "The Forrester Wave™: Service Virtualization And Testing Solutions," 2014.

[107] S. Kelly and J. P. Tolvanen, "Domain-Specific Modeling: Enabling Full Code Generation," ed: Wiley, 2008.

[108] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," ACM Comput. Surv., vol. 37, pp. 316-344, 2005.

[109] J. M. Neighbors, "Software construction using components," University of California, Irvine, 1980.

[110] P. A. Laplante and C. J. Neill, "The Demise of the Waterfall Model Is Imminent," Queue*,* vol. 1, pp. 10-15, 2004.

[111] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," Information and software technology, vol. 50, pp. 833-859, 2008.

[112] E. Visser, "WebDSL: A Case Study in Domain-Specific Language Engineering," in Generative and Transformational Techniques in Software Engineering II. vol. 5235, ed: Springer Berlin Heidelberg, pp. 291-373, 2008.

[113] S. McConnell, "Rapid Development: Taming Wild Software Schedules," Microsoft Press, 1996.

[114] R. Prieto-Diaz, "Domain analysis: an introduction," SIGSOFT Softw. Eng. Notes, vol. 15, pp. 47-54, 1990.

[115] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," DTIC Document,1990.

[116] D. Spinellis, "Notable design patterns for domain-specific languages," J. Syst. Softw.*,* vol. 56, pp. 91-99, 2001.

[117] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel, "Design guidelines for domain specific languages," arXiv preprint arXiv:1409.2378, 2014.

[118] M. Voelter, "DSL Engineering: Designing, Implementing and Using Domain-specific Languages," CreateSpace Independent Publishing Platform, 2013.

[119] D. L. Atkins, T. Ball, G. Bruns, and K. Cox, "Mawl: a domain-specific language for form-based services," Software Engineering, IEEE Transactions on, vol. 25, pp. 334-346, 1999.

[120] D. L. McGuinness and F. Van Harmelen, "OWL web ontology language overview," W3C recommendation*,* 2004.

[121] K. A. Schneider and J. R. Cordy, "AUI: A programming language for developing plastic interactive software," in System Sciences, HICSS. Proceedings of the 35th Annual Hawaii International Conference on, pp. 3656-3665, 2002.

[122] R. A. van Engelen, "ATMOL: A domain-specific language for atmospheric modeling," *CIT.* Journal of computing and information technology, vol. 9, pp. 289-303, 2001.

[123] D. L. Moody, "The “Physics” of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering," Software Engineering, IEEE Transactions on, vol. 35, pp. 756-779, 2009.

[124] W. Citrin, "Strategic directions in visual languages research," *ACM* Computing Surveys (CSUR), vol. 28, p. 132, 1996.

[125] L. Li, J. Grundy, and J. Hosking, "A visual language and environment for enterprise system modelling and automation," Journal of Visual Languages & Computing, vol. 25, pp. 253-277, 2014.

[126] Z. Hemel, L. Kats, and E. Visser, "Code Generation by Model Transformation: A Case Study in Transformation Modularity," 2008.

[127] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/XT 0.17. A language and toolset for program transformation," Science of computer programming, vol. 72, pp. 52-70, 2008.

[128] M. van den Brand, P. Klint, and J. Vinju, "The Syntax Definition Formalism SDF," 1989.

[129] A. S. Gert Jan van Dorsten, Arthur Barendsen, "Core Banking Systems Survey," Available: https://www.nl.capgemini.com/resource-file-access/resource/pdf/Core_Banking_Systems_Survey_2008_0.pdf, 2008,

[130] IETF, "Lightweight Directory Access Protocol (LDAP) v3," ed: The Internet Engineering Task Force, 2006.

[131] J. Han, "Rich Interface Specification for Software Components," Peninsula School of Computing and Information Technology Monash University, McMahons Road Frankston, Australia, 2000.

[132] T. Wagner, T. Bashor, P. Meijer, and P. Humphrey, "Overview of the Eclipse Web Tools Platform," Available: http://www.oracle.com/technetwork/articles/grid/eclipse-web-tools-platform-093378.html, 2005.

[133] Altova, "Altova XMLSpy 2016 Professional Edition User and Reference Manua," Available: http://www.altova.com/documents/XMLSpyPro.pdf, 2016.

[134] en.wikibooks.org, "MySQL," 2013.

[135] J. C. Grundy, "A visual programming environment for object-oriented languages," presented at the Tools US Conference, Santa Barbara, USA, 1991.

[136] M. Boshernitsan and M. Downes, "Visual Programming Languages: A Survey," Computer Science, 2004.

[137] J. Zhang and D. A. Norman, "Representations in distributed cognitive tasks," Cognitive science, vol. 18, pp. 87-122, 1994.

[138] J. H. Larkin and H. A. Simon, "Why a Diagram is (Sometimes) Worth Ten Thousand Words," Cognitive Science, vol. 11, pp. 65-100, 1987.

[139] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework," Journal of Visual Languages & Computing, vol. 7, pp. 131-174, 1996.

[140] D. Moody, "Theory development in visual language research: Beyond the cognitive dimensions of notations," in Visual Languages and Human-Centric Computing, VL/HCC IEEE Symposium on, pp. 151-154, 2009.

[141] G. Dai, X. Bai, Y. Wang, and F. Dai, "Contract-based testing for web services," in Computer Software and Applications Conference, COMPSAC 31st Annual International, pp. 517-526, 2007.

[142] D. Amyot, H. Farah, and J.-F. Roy, "Evaluation of Development Tools for Domain-Specific Modeling Languages," in System Analysis and Modeling: Language Profiles. vol. 4320, R. Gotzhein and R. Reed, Eds., ed: Springer Berlin Heidelberg, pp.183-197, 2006.

[143] V. University, "GME: Generic Modeling Environment," Available: http://www.isis.vanderbilt.edu/Projects/gme/, 2008.

[144] Telelogic, "Telelogic tau g2 download," Available: http://softadvice.informer.com/Telelogic_Tau_G2_Download.html, 2017.

[145] K. Mittal, "Introducing IBM Rational Software Architect," 2005.

[146] XMF, "XMF and XMF-Mosaic," 2011.

[147] Apache, "Apache TCPMon," Available: https://ws.apache.org/tcpmon/index.html, 2013.

[148] B. Bulgurcu, H. Cavusoglu, and I. Benbasat, "Information security policy compliance: an empirical study of rationality-based beliefs and information security awareness," MIS quarterly, vol. 34, pp. 523-548, 2010.

[149] M. D. Abrams, S. Jajodia, and H. J. Podell, "Information Security: An Integrated Collection of Essays," 1995.

[150] V. Radha and D. H. Reddy, "A survey on single sign-on techniques," Procedia Technology, vol. 4, pp. 134-139, 2012.

[151] H.-C. Kim, H.-W. Lee, K.-S. Lee, and M.-S. Jun, "A design of one-time password mechanism using public key infrastructure," in Networked Computing and Advanced Information Management, NCM'08. Fourth International Conference on, pp. 18-24, 2008.

[152] O. Adeyinka, "Internet Attack Methods and Internet Security Technology," presented at the Proceedings of the Second Asia International Conference on Modelling & Simulation (AMS), 2008.

[153] Wikipedia, "Mandatory access control," Available: https://en.wikipedia.org/wiki/Mandatory_access_control, 2016

[154] Wikipedia, "Discretionary access control," 2017.

[155] M. E. Hellman, "An overview of public key cryptography," IEEE Communications Magazine, vol. 40, pp. 42-49, 2002.

[156] E. Rescorla, "HTTP Over TLS," RFC Editor, 2000.

[157] F. Piper, "Cryptography," Wiley Online Library, 2002.

[158] J. Kim, A. Biryukov, B. Preneel, and S. Hong, "On the security of HMAC and NMAC based on HAVAL, MD4, MD5, SHA-0 and SHA-1," in International Conference on Security and Cryptography for Networks, pp. 242-256, 2006,

[159] D. Sosnoski, "Java Web services: Axis2 WS-Security basics," Available: http://www.ibm.com/developerworks/library/j-jws4/, 2009.

[160] OASIS, "Web Services Security: UsernameToken Profile 1.1," ed: OASIS, 2005.

[161] D. Sosnoski, "Java Web services: Axis2 WS-Security signing and encryption," 2009.

[162] S. L. Pfleeger and B. A. Kitchenham, "Principles of survey research: part 1: turning lemons into lemonade," *SIGSOFT* Softw. Eng. Notes, vol. 26, pp. 16-18, 2001.

[163] F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," MIS quarterly, pp. 319-340, 1989.

[164] J. Brooke, "SUS-A quick and dirty usability scale," Usability evaluation in industry*,* vol. 189, pp. 4-7, 1996.

[165] J. Sauro and J. R. Lewis, "Quantifying the User Experience: Practical Statistics for User Research," Morgan Kaufmann Publishers Inc., 2012.

[166] A. Bangor, P. T. Kortum, and J. T. Miller, "An empirical evaluation of the system usability scale," Intl. Journal of Human–Computer Interaction, vol. 24, pp. 574-594, 2008.

APPENDIX I

Approval Letter from Swinburne University Human Research Ethics Committee

From: Astrid Nordmann
Sent: Monday, 11 January 2016 10:13 AM
To: John Grundy
Cc: RES Ethics; Jian Liu; Iman Avazpour; Mohamed Abdelrazek (mohamed.abdelrazek@deakin.edu.au)
Subject: SHR Project 2015/326 - Ethics clearance

To: Prof John Grundy/Mr Jian Liu, FSET

Dear John and Jian Liu

SHR Project 2015/326 – Evaluation of a domain-specific visual modelling language for enterprise testing environment emulation (TeeVML)
Prof. John Grundy, Mr Jian Liu (Student), Dr Iman Avazpour - FSET
Approved duration: 11-01-2016 to 11-01-2018 [adjusted]

I refer to the ethical review of the above project by a Subcommittee (SHESC3) of Swinburne's Human Research Ethics Committee (SUHREC). Your responses to the review as emailed on 11 January 2016 were put to the Subcommittee delegate for consideration.

I am pleased to advise that, as submitted to date, ethics clearance has been given for the above project to proceed in line with standard on-going ethics clearance conditions outlined below.

- All human research activity undertaken under Swinburne auspices must conform to Swinburne and external regulatory standards, including the *National Statement on Ethical Conduct in Human Research* and with respect to secure data use, retention and disposal.

- The named Swinburne Chief Investigator/Supervisor remains responsible for any personnel appointed to or associated with the project being made aware of ethics clearance conditions, including research and consent procedures or instruments approved. Any change in chief investigator/supervisor requires timely notification and SUHREC endorsement.

- The above project has been approved as submitted for ethical review by or on behalf of SUHREC. Amendments to approved procedures or instruments ordinarily require prior ethical appraisal/clearance. SUHREC must be notified immediately or as soon as possible thereafter of (a) any serious or unexpected adverse effects on participants and any redress measures; (b) proposed changes in protocols; and (c) unforeseen events which might affect continued ethical acceptability of the project.

- At a minimum, an annual report on the progress of the project is required as well as at the conclusion (or abandonment) of the project. Information on project monitoring and variations/additions, self-audits and progress reports can be found on the Research Intranet pages.

- A duly authorised external or internal audit of the project may be undertaken at any time.

Please contact the Research Ethics Office if you have any queries about on-going ethics clearance, citing the Swinburne project number. A copy of this email should be retained as part of project record-keeping.

Best wishes for the project.

Yours sincerely,
Astrid Nordmann
SHESC3 Secretary

APPENDIX II

Phase One Questionnaire

Section One: Demographic Information

1. I agree to take part in the survey
   - ☐ Yes
   - ☐ No
2. Your gender
   - ☐ Male
   - ☐ Female
   - ☐ Prefer not to say
3. Your age
   - ☐ 20-30
   - ☐ 31-40
   - ☐ 41-50
   - ☐ 51-60
   - ☐ 61+
4. How many years of IT experience do you have? (including industry and IT research)
   - ☐ 0-1
   - ☐ 2-5
   - ☐ 6-10
   - ☐ 11-15
   - ☐ 16+
5. How many years of software testing experience do you have? (including industry and IT research)
   - ☐ 0-1
   - ☐ 2-5
   - ☐ 6-10
   - ☐ 11-15
   - ☐ 16+

6. How familiar are you with software application inter-connectivity and inter-operability test?
   - ☐ Very familiar
   - ☐ Somewhat familiar
   - ☐ I had heard about it
   - ☐ Not familiar at all

7. How familiar are you with domain-specific modeling and domain-specific language?
   - ☐ Very familiar
   - ☐ Somewhat familiar
   - ☐ I had heard about it
   - ☐ Not familiar at all

Section Two: Overall Requirement for an Enterprise Testing Environment

8. In your opinion, an emulated testing environment is useful for an application inter-connectivity and inter-operability test.

   Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

9. What kinds of testing features do you want to see an emulated testing environment provides to system under test for inter-connectivity and inter-operability test? (may have multiple selections)
   - ☐ Correctness of message signature
   - ☐ Correctness of interactive protocol
   - ☐ Correctness of interactive behavior
   - ☐ Conformance to non-functional requirement
   - ☐ Other, please specify:

10. In which software development stage(s) will emulated testing environment be used? (may have multiple selections)
    - ☐ Integration testing
    - ☐ System testing
    - ☐ User acceptance testing
    - ☐ Regression testing
    - ☐ Other, please specify:

11. What is you preferred approach to developing an emulated testing environment?
    - ☐ A third generation general purpose programming language (e.g. Java)
    - ☐ A domain-specific texual modeling programming language
    - ☐ A domain-specific visual modeling programming language

☐ Other, please specify:

12. How do you rank the importance to an emulated testing environment? (please fill in a *number* from 1 to 4 to each box, and 1 is the highest priority and 4 is the lowest)

    ☐ Tool development productivity
    ☐ Ease of use its development tool
    ☐ Testing functionality provided to system under test
    ☐ Result reliability

13. What are the main motivations for you to use emulated testing environment? (may have multiple selections)

    ☐ Cost saving on application software and hardware investment
    ☐ Effort saving on application installation and maintenance
    ☐ Lack of application knowledge
    ☐ Early detection of interface defects
    ☐ Other, please specify:

14. What are your main concerns, which could prevent you from using emulated testing environment? (may have multiple selections)

    ☐ Extra development effort on testing endpoints
    ☐ Learning a new technology
    ☐ Inadequate testing functionality
    ☐ Emulation accuracy
    ☐ Result reliability
    ☐ Other, please specify:

15. Is there anything emulated testing environment does not let you do that you would like to?

    Please specify:

16. Your overall comment on this part:

Section Three: Requirement for an Enterprise Testing Environment on Each Interface Layer

Message Signature Modeling

17. It is useful for an emulated testing environment to provide signature testing functionality to its system under test.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

18. In your opinion, which is more important for the tool to model an endpoint message signature?
    - ☐ High productivity on signature modeling
    - ☐ Ease of use of the tool to model message signature

19. What signature testing functionality should an endpoint have? (may have multiple selections)
    - ☐ To test the correctness of each request service name and parameter names
    - ☐ To test correctness of parameter types and orders
    - ☐ To test if all mandatory parameters are provided
    - ☐ To test all parameter values within specified ranges
    - ☐ Other, please specify:

20. Your comment on signature modeling:

_____

Interactive Protocol Modeling

21. It is useful for an emulated testing environment to provide interactive protocol testing functionality to its system under test.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

22. In your opinion, which is more important for the tool to model an endpoint interactive protocol?
    - ☐ High productivity on interactive protocol modeling
    - ☐ Ease of use of the tool to model interactive protocol

23. What interactive protocol testing functionality should an emulated testing environment have? (may have multiple selections)
    - ☐ To validate a service by endpoint state
    - ☐ To validate a service by service parameter(s) and endpoint state
    - ☐ To validate a service by service return value(s) and endpoint state
    - ☐ To validate a service by endpoint internal event
    - ☐ To simulate synchronous process
    - ☐ To simulate unsafe operation
    - ☐ Other, please specify:

24. Your comment on protocol modeling:

_____

Interactive Behavior Modeling

25. It is useful for an emulated testing environment to provide interactive behavior testing functionality to its system under test.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

26. How do you rank the importance of interactive behavior modeling? (please fill in a number from 1 to 4 to each box, and 1 is the highest priority and 4 is the lowest)

    ☐ High productivity on interactive behavior modeling
    ☐ Ease of use of the tool to model interactive behavior
    ☐ Interactive behavior testing functionality provided to system under test
    ☐ Accuracy on return results

27. It is useful for an emulated testing environment to provide data store testing functionality to its system under test.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

28. You prefer to have approximate return results from an emulated testing environment, if its development effort can be reduced significantly.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

29. Your comment on interactive behavior modeling:

    _____

Non-functional Requirement Modeling

30. It is useful for an emulated testing environment to provide non-functional requirement testing features to its system under test.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

31. What are the non-functional requirement testing features an emulated testing environment should provide? (may have multiple selections)

    ☐ Performance test
    ☐ Security test
    ☐ Reliability test
    ☐ Other, please specify:

32. Your comment on non-functional requirement modeling:

    _____

APPENDIX III

Phase Two Questionnaire

Section One: Demographic Information

1. I agree to take part in the survey
   - ☐ Yes
   - ☐ No
2. Your gender
   - ☐ Male
   - ☐ Female
   - ☐ Prefer not to say
3. Your age
   - ☐ 20-30
   - ☐ 31-40
   - ☐ 41-50
   - ☐ 51-60
   - ☐ 61+
4. How familiar are you with software application inter-connectivity and inter-operability test?
   - ☐ Very familiar
   - ☐ Somewhat familiar
   - ☐ I had heard about it
   - ☐ Not familiar at all
5. How familiar are you with domain-specific modeling and domain-specific language?
   - ☐ Very familiar
   - ☐ Somewhat familiar
   - ☐ I had heard about it
   - ☐ Not familiar at all
6. What best describes your area?
   - ☐ Software engineer

- ☐ Research student in IT field
- ☐ Computer science or software engineering undergraduate
- ☐ Other undergraduate or postgraduate student
- ☐ Other, specify:

7. What is your educational background ?
   - ☐ Software engineering / Computer science
   - ☐ Engineering / Science (excluding software engineering and computer science)
   - ☐ Art / Business management
   - ☐ Other, please specify:

Section Two: Overall Usability of the TeeVML Tool

8. Have you completed the assigned task?
   - ☐ Yes
   - ☐ No

9. Have you completed the assigned task?
   - ☐ 10 – 15 minutes
   - ☐ 16 – 20 minutes
   - ☐ 21 – 25 minutes
   - ☐ 26 – 30 minutes
   - ☐ 31+ minutes

10. How many times have you asked for support?
    - ☐ None
    - ☐ One time
    - ☐ Two times
    - ☐ Three times
    - ☐ Four times or more

11. Which phase did you stop at?

System Usability Scale

12. You would like to use the tool in your future project.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

13. You found the tool unnecessarily complex.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

14. You found the tool was easy to use.

Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

15. You would need support to be able to use the tool.

Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

16. You found the various features of the tool were well integrated.

Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

17. You found there was too much inconsistency in the tool.

Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

18. You would image that most people would learn to use the tool very quickly.

Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

19. You found the tool very cumbersome to use.

Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

20. You felt very confident using the tool.

Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

21. You needed to learn a lot of things before you could get going with the tool.

Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

Justification for Use of the Tool

22. In your opinion, comparing to a third generation language (e.g. Java) you are familiar with, how much would a typical endpoint development effort be reduced by using the tool?

☐ almost the same
☐ 10% - 25%
☐ 26% - 50%
☐ 51% or more

23. What would be your main motivations for you to use the tool? (may have multiple selections)

☐ Ease of use
☐ Short learning curve
☐ High development productivity
☐ Ease of maintenance
☐ Other, please specify:

24. What would be your main concerns, which could prevent you from using the tool? (may have multiple selections)

☐ Extra time spending on learning a new language

- ☐ Lack of software modeling skills
- ☐ Inadequate expressive power
- ☐ Lack of syntax error checking mechanism
- ☐ Other, please specify:

25. Is there anything the tool does not let you do that you would like to?
    Please specify:

26. Your comment on overall usability of the TeeVML tool.

Section Three: Usability of Each Interface Layer Language of the TeeVML Tool

Message Signature Language

27. Endpoint signature is easily modelled by the tool.
    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

28. It is easy to visually see various parts and relationships of a message signature.
    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

29. It is easy to make changes to message signature model.
    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

30. It is easy to make errors or mistakes during message signature definition.
    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

31. It is capable of defining all types of message signatures you have seen.
    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

32. Are there any message signature notations that should be made clearer for the user? How?
    Please specify:

Interactive Protocol Language

33. Endpoint protocol is easily modelled by the tool.
    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

34. It is easy to visually see all valid service requests and their dependencies on other factors.
    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

35. It is easy to make changes to interactive protocol model.
    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

36. It is easy to make errors or mistakes during interactive protocol definition.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

37. It is capable of defining all interactive protocol scenarios you have seen.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

38. Are there any interactive protocol notations that should be made clearer for the user? How?

    Please specify:

Interactive Behavior Language

39. Endpoint interactive behavior is easily modelled by the tool.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

40. It is easy to visually see all inputs/outputs, data store manipulations and behavior logic processes.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

41. It is easy to make changes to interactive behavior model.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

42. It is easy to make errors or mistakes during interactive behavior definition.

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

43. The tool has sufficient behavioral expressive power for creating behavior model with accurate outputs?

    Strongly Disagree ☐ ☐ ☐ ☐ ☐ Strongly Agree

44. Are there any interactive behavior notations that should be made clearer for the user? How?

    Please specify:

45. Your comment on usability of each interface layer sub-language of the TeeVML tool.

APPENDIX IV

Phase One Survey Results Report

Functionality Report

Table of contents

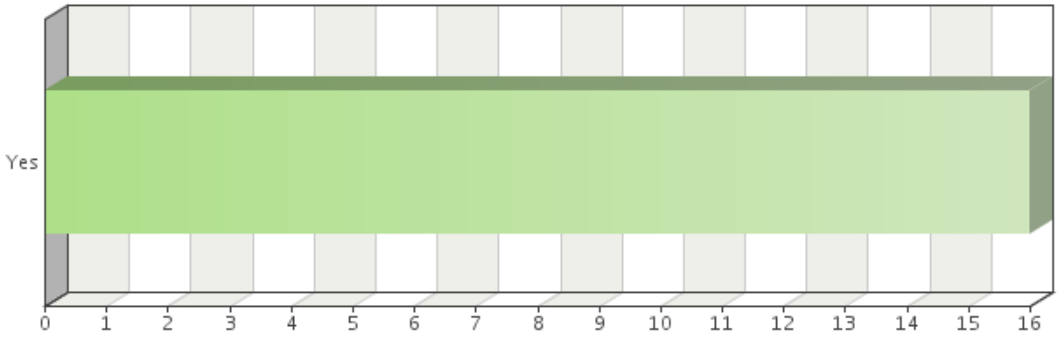
Report info....1
Question 1: I agree to take part in the survey....2
Question 2: Your gender: 3
Question 3: Your age:....4
Question 4: How many years of IT experience do you have? (including industry and IT research)....5
Question 5: How many years of software testing experience do you have? (including industry and IT res....6
Question 6: How familiar are you with software application inter-connectivity and inter-operability t....7
Question 7: How familiar are you with domain-specific modeling and domain-specific language?....8
Question 8: In your opinion, an emulated testing environment is useful for an application inter-conne....9
Question 9: What kinds of testing features do you want to see an emulated testing environment provide....10
Question 10: In which software development stage(s) will emulated testing environment be used? (may h....11
Question 11: What is you preferred approach to developing an emulated testing environment? 12
Question 12: How do you rank the importance to an emulated testing environment? (please fill in a num....13
Dropdown cell (row 1, column 2)....13
Dropdown cell (row 2, column 2)....13
Dropdown cell (row 3, column 2)....14
Dropdown cell (row 4, column 2)....14
Question 13: What are the main motivations for you to use emulated testing environment? (may have mul....16
Question 14: What are your main concerns, which could prevent you from using emulated testing environ....17
Question 15: Is there anything emulated testing environment does not let you do that you would like t....18
Question 16: Your overall comment on this part....19
Question 17: It is useful for an emulated testing environment to provide signature testing functional....20
Question 18: In your opinion, which is more important for the tool to model an endpoint message signa....21
Question 19: What signature testing functionality should an endpoint have? (may have multiple selecti....22
Question 20: Your comment on signature modeling....23
Question 21: It is useful for an emulated testing environment to provide interactive protocol testing....24
Question 22: In your opinion, which is more important for the tool to model an endpoint interactive p....25
Question 23: What interactive protocol testing functionality should an emulated testing environment h....26
Question 24: Your comment on protocol modeling....27
Question 25: It is useful for an emulated testing environment to provide interactive behavior testing....28
Question 26: How do you rank the importance of interactive behavior modeling? (please fill in a numbe....29
Dropdown cell (row 1, column 2)....29
Dropdown cell (row 2, column 2)....29
Dropdown cell (row 3, column 2)....30
Dropdown cell (row 4, column 2)....30
Question 27: It is useful for an emulated testing environment to provide data store testing functiona....32
Question 28: You prefer to have approximate return results from an emulated testing environment, if i....33
Question 29: Your comment on interactive behavior modeling....34
Question 30: It is useful for an emulated testing environment to provide non-functional requirement t....35
Question 31: What are the non-functional requirement testing features an emulated testing environment....36
Question 32: Your comment on non-functional requirement modeling....37
Question 33: Thank you very much for taking part in the survey. If you have any questions or issues....38

Report info

Report date:	Friday, March 4, 2016 1:45:12 PM EST
Start date:	Tuesday, January 12, 2016 4:25:00 PM EST
Stop date:	Friday, April 1, 2016 4:25:00 PM EST
Number of completed responses:	16

Question 1

I agree to take part in the survey.



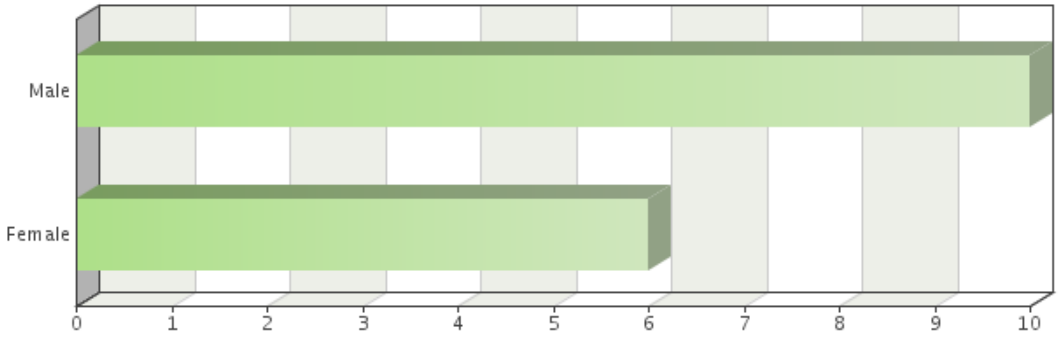
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Yes	16	100%	100%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 2

Your gender:



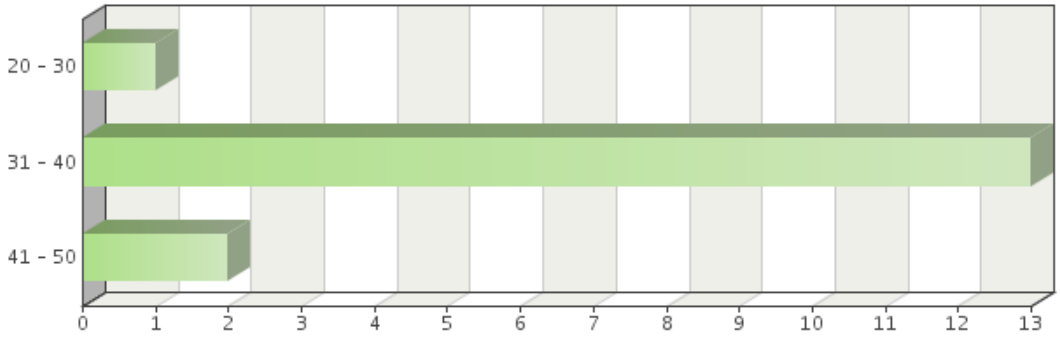
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Male	10	62.5%	62.5%
Female	6	37.5%	37.5%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 3

Your age:



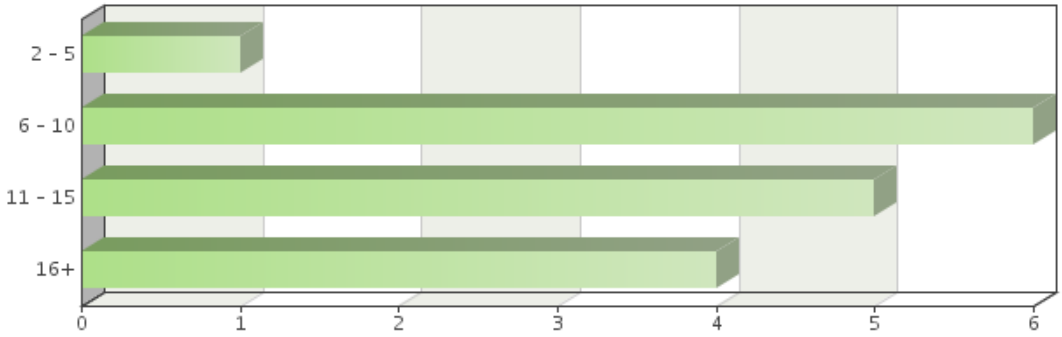
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
20 - 30	1	6.25%	6.25%
31 - 40	13	81.25%	81.25%
41 - 50	2	12.5%	12.5%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 4

How many years of IT experience do you have? (including industry and IT research)



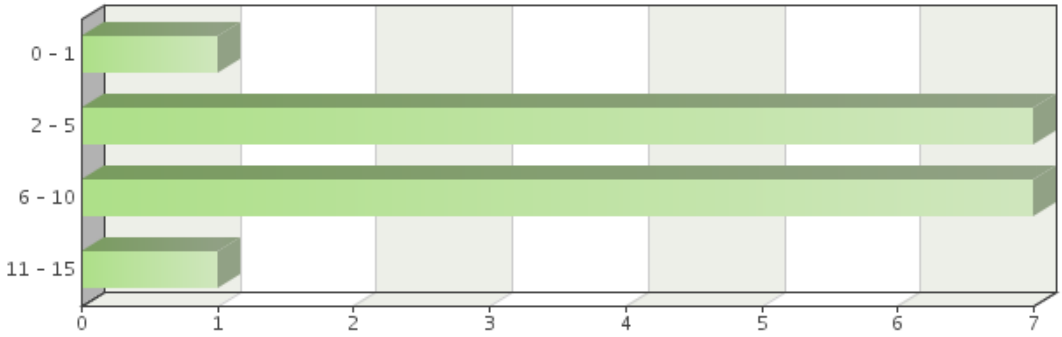
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
2 - 5	1	6.25%	6.25%
6 - 10	6	37.5%	37.5%
11 - 15	5	31.25%	31.25%
16+	4	25%	25%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 5

How many years of software testing experience do you have? (including industry and IT research)



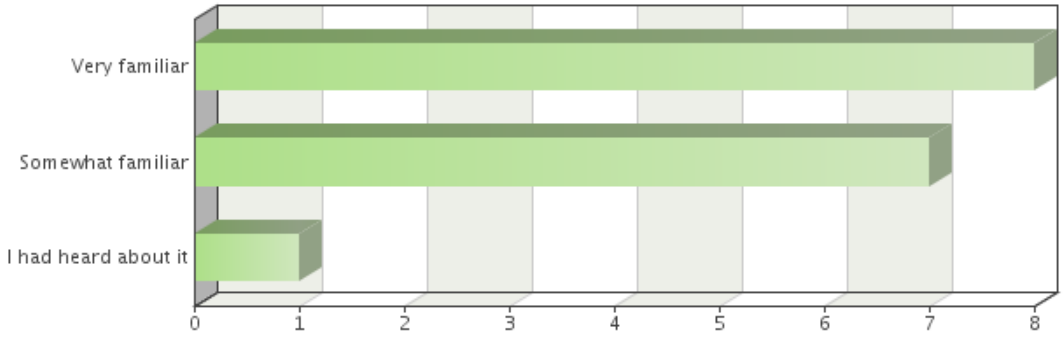
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
0 - 1	1	6.25%	6.25%
2 - 5	7	43.75%	43.75%
6 - 10	7	43.75%	43.75%
11 - 15	1	6.25%	6.25%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 6

How familiar are you with software application inter-connectivity and inter-operability test?



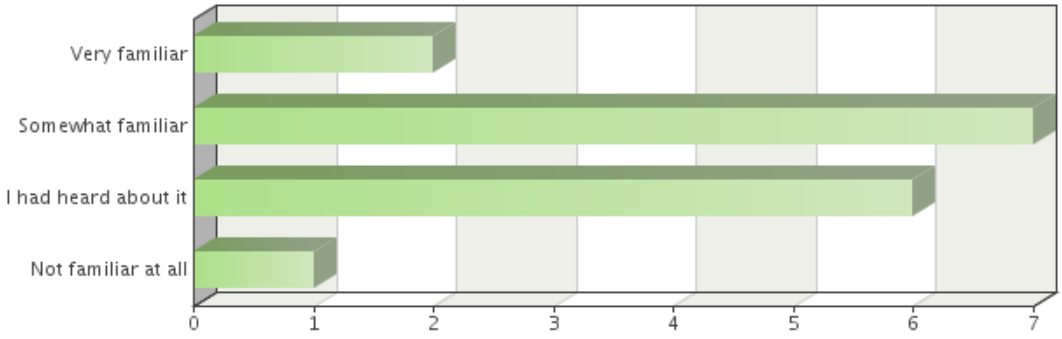
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Very familiar	8	50%	50%
Somewhat familiar	7	43.75%	43.75%
I had heard about it	1	6.25%	6.25%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 7

How familiar are you with domain-specific modeling and domain-specific language?



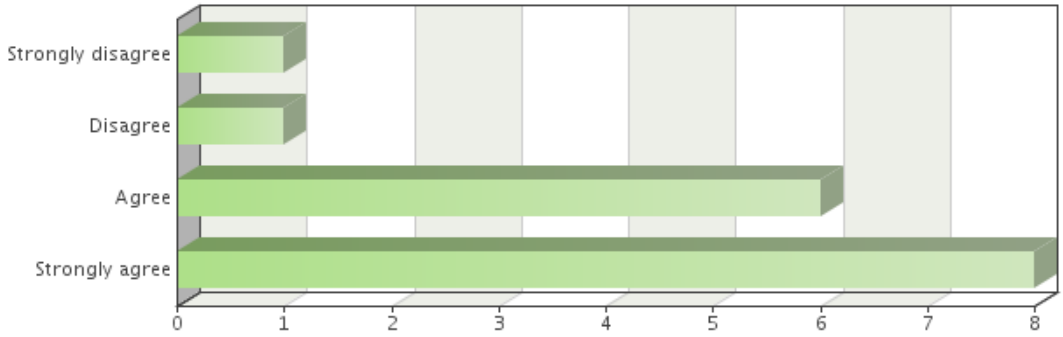
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Very familiar	2	12.5%	12.5%
Somewhat familiar	7	43.75%	43.75%
I had heard about it	6	37.5%	37.5%
Not familiar at all	1	6.25%	6.25%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 8

In your opinion, an emulated testing environment is useful for an application inter-connectivity and inter-operability test.



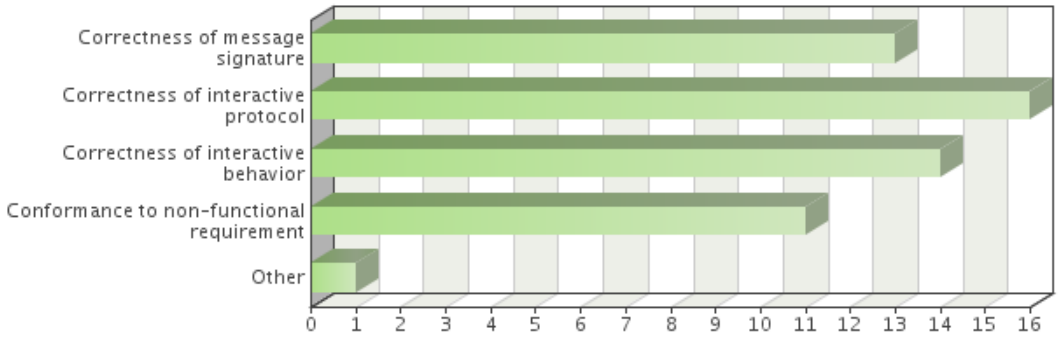
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Strongly disagree	1	6.25%	6.25%
Disagree	1	6.25%	6.25%
Agree	6	37.5%	37.5%
Strongly agree	8	50%	50%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 9

What kinds of testing features do you want to see an emulated testing environment provides to system under test for inter-connectivity and inter-operability test? (may have multiple selections)



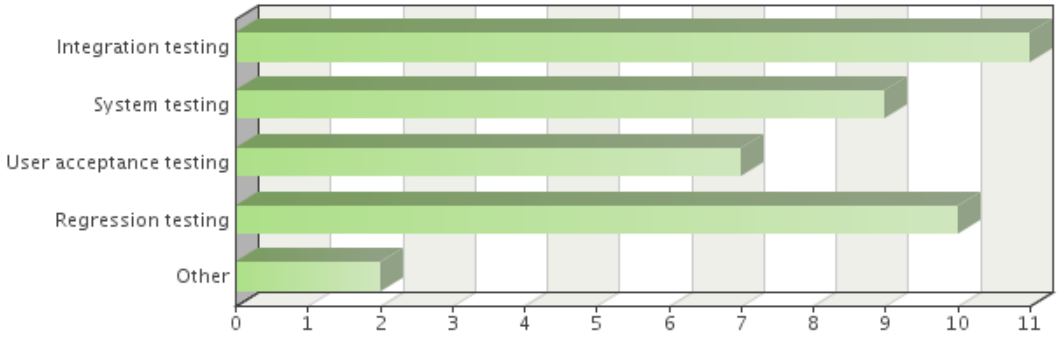
Frequency table

Choices	Absolute frequency	Relative frequency by choice	Relative frequency	Adjusted relative frequency
Correctness of message signature	13	23.64%	81.25%	81.25%
Correctness of interactive protocol	16	29.09%	100%	100%
Correctness of interactive behavior	14	25.45%	87.5%	87.5%
Conformance to non-functional requirement	11	20%	68.75%	68.75%
Other	1	1.82%	6.25%	6.25%
Sum:	55	100%	-	-
Not answered:	0	-	0%	-

Total answered: 16

Question 10

In which software development stage(s) will emulated testing environment be used? (may have multiple selections)



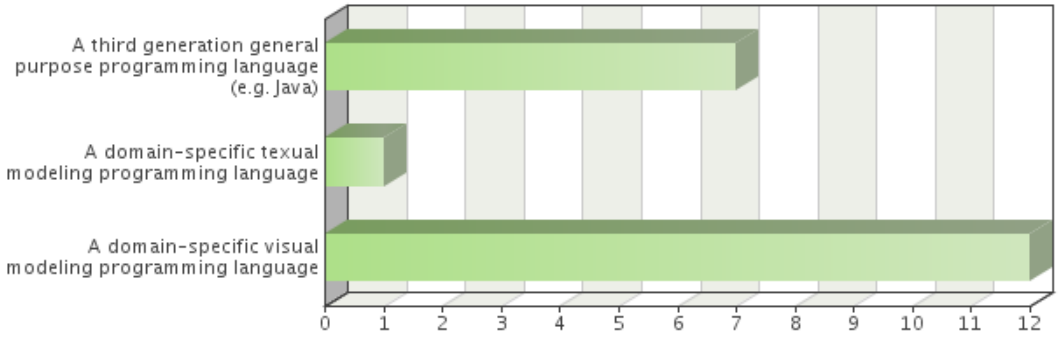
Frequency table

Choices	Absolute frequency	Relative frequency by choice	Relative frequency	Adjusted relative frequency
Integration testing	11	28.21%	68.75%	68.75%
System testing	9	23.08%	56.25%	56.25%
User acceptance testing	7	17.95%	43.75%	43.75%
Regression testing	10	25.64%	62.5%	62.5%
Other	2	5.13%	12.5%	12.5%
Sum:	39	100%	-	-
Not answered:	0	-	0%	-

Total answered: 16

Question 11

What is you preferred approach to developing an emulated testing environment?



Frequency table

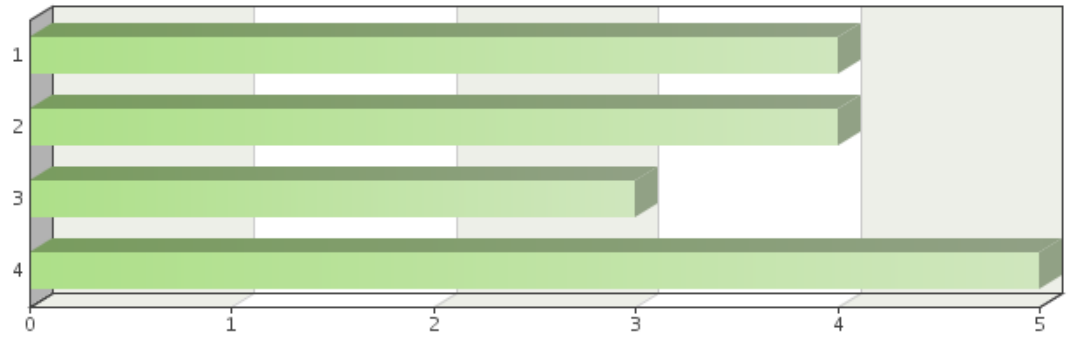
Choices	Absolute frequency	Relative frequency by choice	Relative frequency	Adjusted relative frequency
A third generation general purpose programming language (e.g. Java)	7	35%	43.75%	43.75%
A domain-specific texual modeling programming language	1	5%	6.25%	6.25%
A domain-specific visual modeling programming language	12	60%	75%	75%
Sum:	20	100%	-	-
Not answered:	0	-	0%	-

Total answered: 16

Question 12

How do you rank the importance to an emulated testing environment? (please fill in a number from 1 to 4 to each box, and 1 is the highest priority and 4 is the lowest)

Dropdown cell (row 1, column 2)

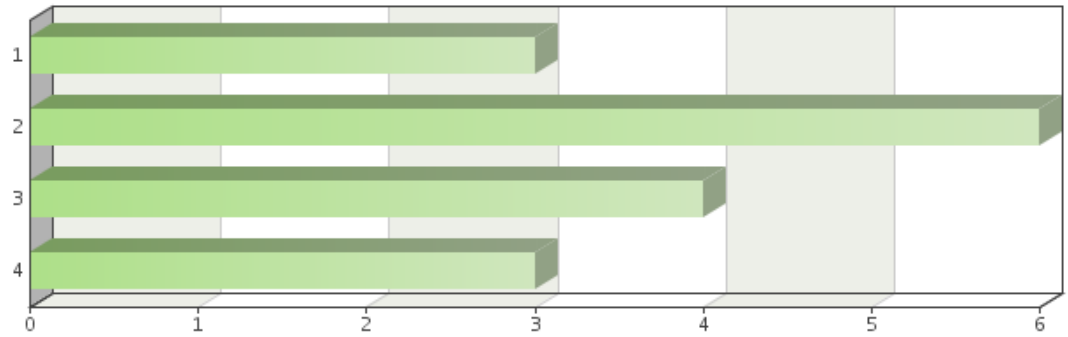


Frequency table

Items	Absolute frequency	Relative frequency	Adjusted relative frequency
1	4	25%	25%
2	4	25%	25%
3	3	18.75%	18.75%
4	5	31.25%	31.25%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Dropdown cell (row 2, column 2)

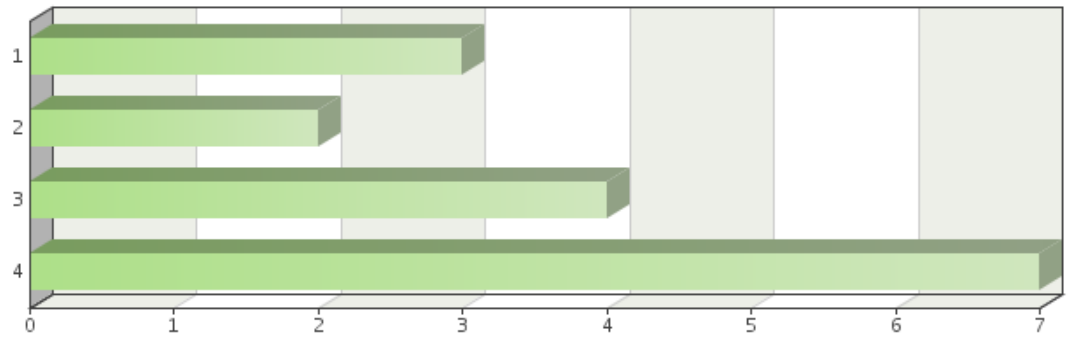


Frequency table

Items	Absolute frequency	Relative frequency	Adjusted relative frequency
1	3	18.75%	18.75%
2	6	37.5%	37.5%
3	4	25%	25%
4	3	18.75%	18.75%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Dropdown cell (row 3, column 2)

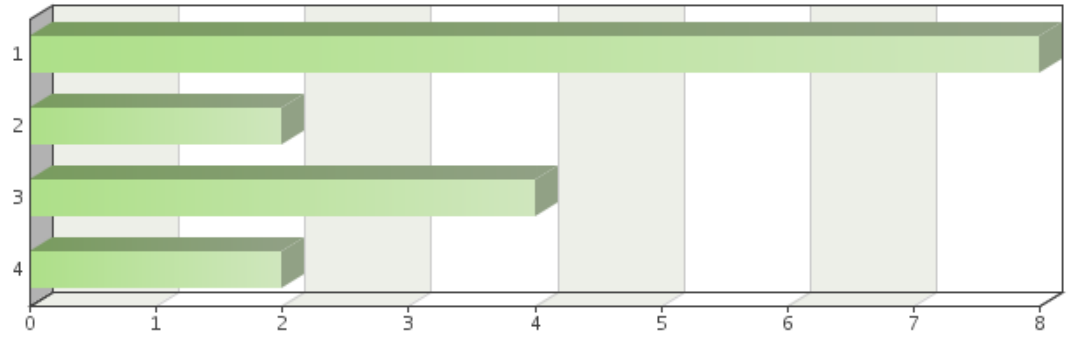


Frequency table

Items	Absolute frequency	Relative frequency	Adjusted relative frequency
1	3	18.75%	18.75%
2	2	12.5%	12.5%
3	4	25%	25%
4	7	43.75%	43.75%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Dropdown cell (row 4, column 2)



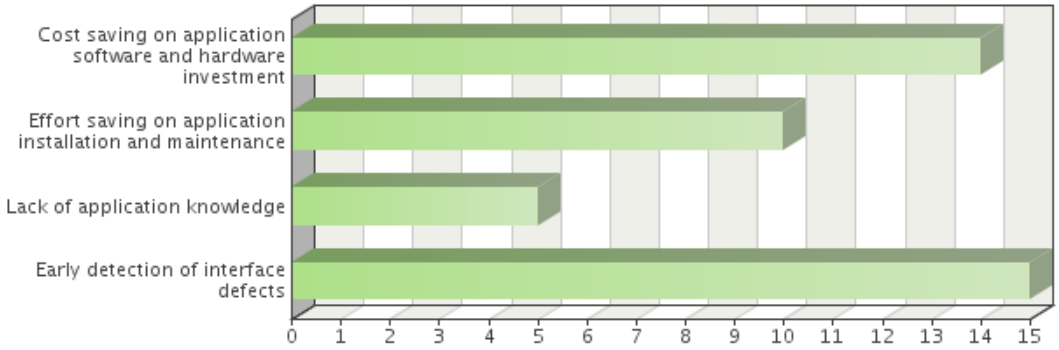
Frequency table

Items	Absolute frequency	Relative frequency	Adjusted relative frequency
1	8	50%	50%
2	2	12.5%	12.5%
3	4	25%	25%
4	2	12.5%	12.5%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 13

What are the main motivations for you to use emulated testing environment? (may have multiple selections)



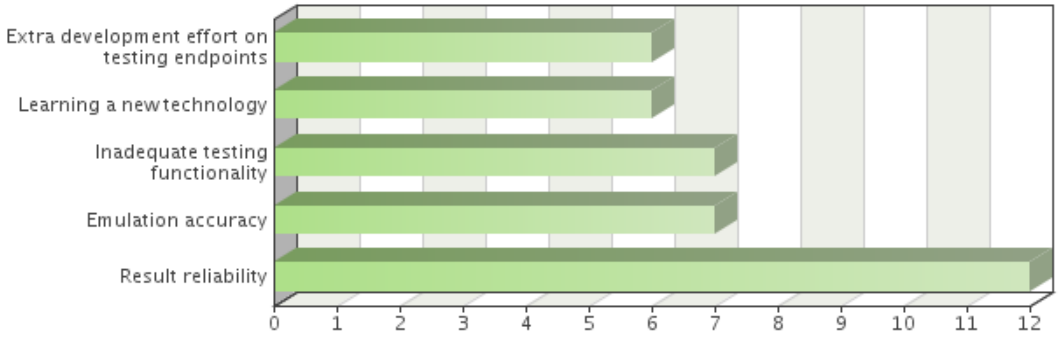
Frequency table

Choices	Absolute frequency	Relative frequency by choice	Relative frequency	Adjusted relative frequency
Cost saving on application software and hardware investment	14	31.82%	87.5%	87.5%
Effort saving on application installation and maintenance	10	22.73%	62.5%	62.5%
Lack of application knowledge	5	11.36%	31.25%	31.25%
Early detection of interface defects	15	34.09%	93.75%	93.75%
Sum:	44	100%	-	-
Not answered:	0	-	0%	-

Total answered: 16

Question 14

What are your main concerns, which could prevent you from using emulated testing environment? (may have multiple selections)



Frequency table

Choices	Absolute frequency	Relative frequency by choice	Relative frequency	Adjusted relative frequency
Extra development effort on testing endpoints	6	15.79%	37.5%	37.5%
Learning a new technology	6	15.79%	37.5%	37.5%
Inadequate testing functionality	7	18.42%	43.75%	43.75%
Emulation accuracy	7	18.42%	43.75%	43.75%
Result reliability	12	31.58%	75%	75%
Sum:	38	100%	-	-
Not answered:	0	-	0%	-

Total answered: 16

Question 15

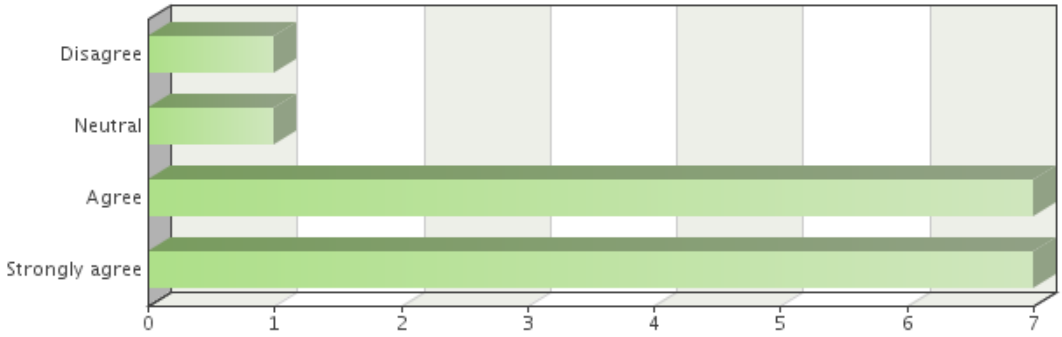
Is there anything emulated testing environment does not let you do that you would like to?

Question 16

Your overall comment on this part.

Question 17

It is useful for an emulated testing environment to provide signature testing functionality to its system under test.



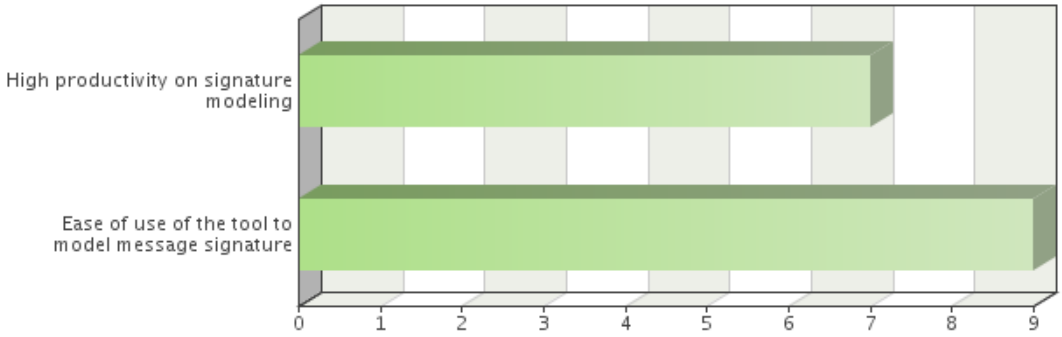
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Disagree	1	6.25%	6.25%
Neutral	1	6.25%	6.25%
Agree	7	43.75%	43.75%
Strongly agree	7	43.75%	43.75%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 18

In your opinion, which is more important for the tool to model an endpoint message signature?



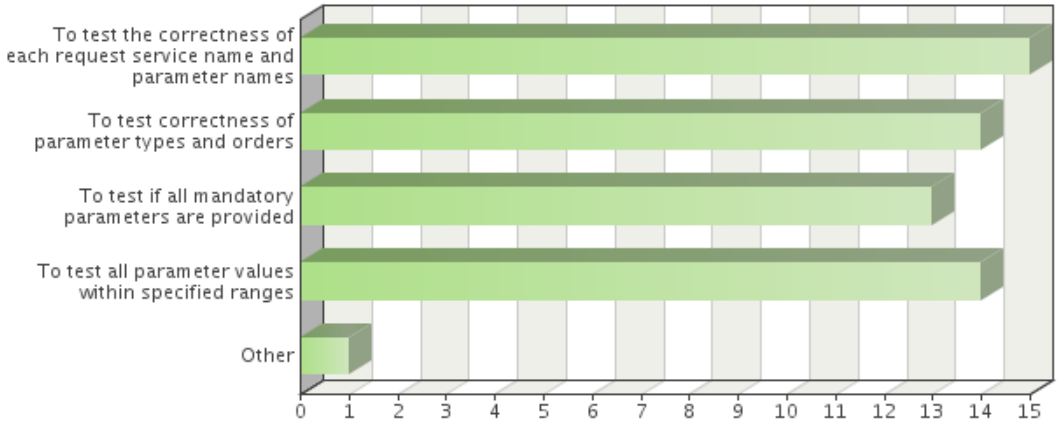
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
High productivity on signature modeling	7	43.75%	43.75%
Ease of use of the tool to model message signature	9	56.25%	56.25%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 19

What signature testing functionality should an endpoint have? (may have multiple selections)



Frequency table

Choices	Absolute frequency	Relative frequency by choice	Relative frequency	Adjusted relative frequency
To test the correctness of each request service name and parameter names	15	26.32%	93.75%	93.75%
To test correctness of parameter types and orders	14	24.56%	87.5%	87.5%
To test if all mandatory parameters are provided	13	22.81%	81.25%	81.25%
To test all parameter values within specified ranges	14	24.56%	87.5%	87.5%
Other	1	1.75%	6.25%	6.25%
Sum:	57	100%	-	-
Not answered:	0	-	0%	-

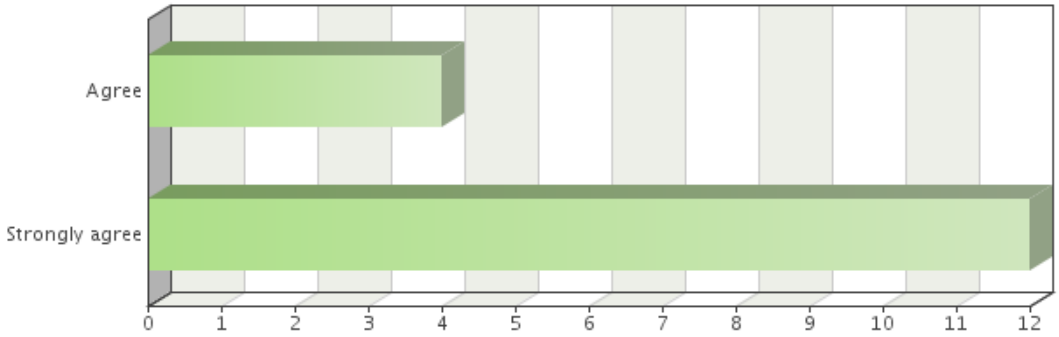
Total answered: 16

Question 20

Your comment on signature modeling

Question 21

It is useful for an emulated testing environment to provide interactive protocol testing functionality to its system under test.



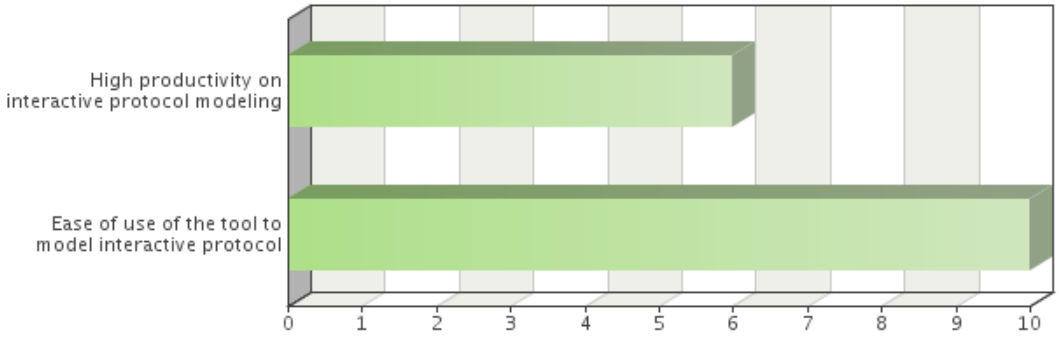
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Agree	4	25%	25%
Strongly agree	12	75%	75%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 22

In your opinion, which is more important for the tool to model an endpoint interactive protocol?



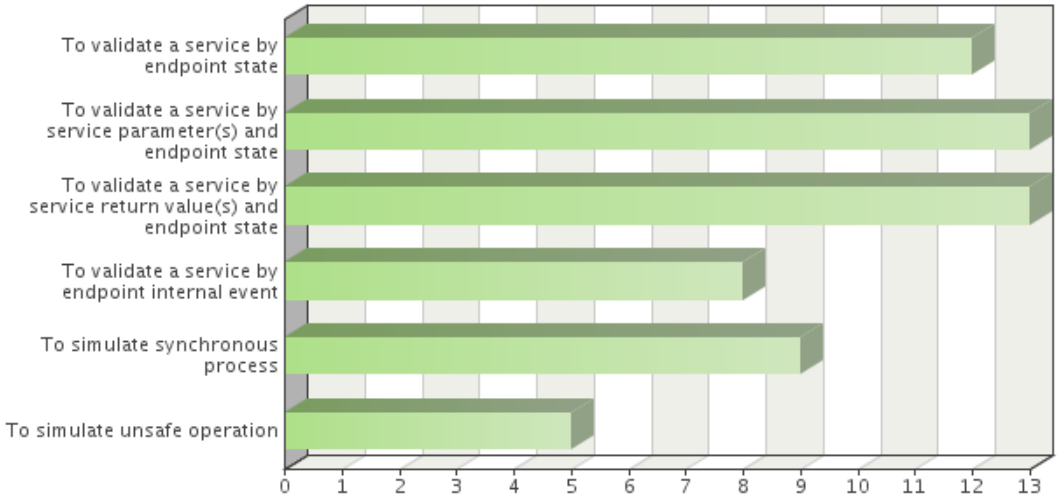
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
High productivity on interactive protocol modeling	6	37.5%	37.5%
Ease of use of the tool to model interactive protocol	10	62.5%	62.5%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 23

What interactive protocol testing functionality should an emulated testing environment have? (may have multiple selections)



Frequency table

Choices	Absolute frequency	Relative frequency by choice	Relative frequency	Adjusted relative frequency
To validate a service by endpoint state	12	20%	75%	75%
To validate a service by service parameter(s) and endpoint state	13	21.67%	81.25%	81.25%
To validate a service by service return value(s) and endpoint state	13	21.67%	81.25%	81.25%
To validate a service by endpoint internal event	8	13.33%	50%	50%
To simulate synchronous process	9	15%	56.25%	56.25%
To simulate unsafe operation	5	8.33%	31.25%	31.25%
Sum:	60	100%	-	-
Not answered:	0	-	0%	-

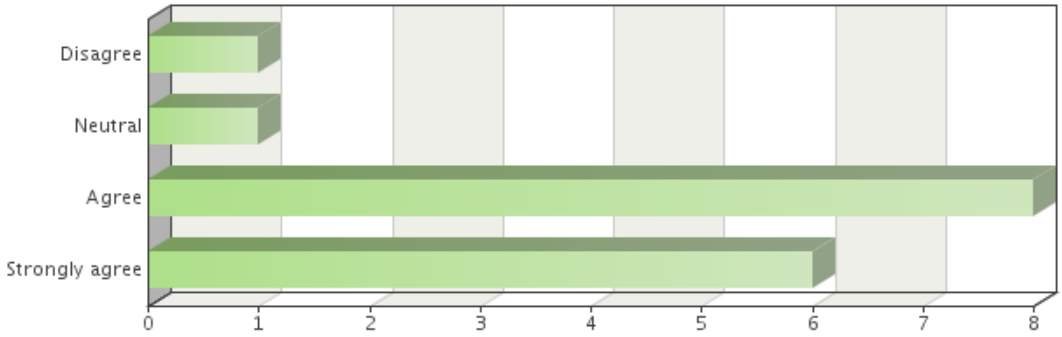
Total answered: 16

Question 24

Your comment on protocol modeling.

Question 25

It is useful for an emulated testing environment to provide interactive behavior testing functionality to its system under test.



Frequency table

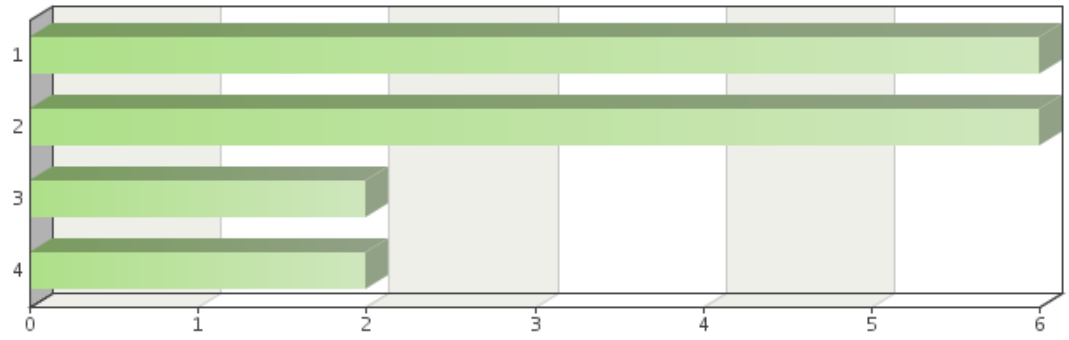
Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Disagree	1	6.25%	6.25%
Neutral	1	6.25%	6.25%
Agree	8	50%	50%
Strongly agree	6	37.5%	37.5%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 26

How do you rank the importance of interactive behavior modeling? (please fill in a number from 1 to 4 to each box, and 1 is the highest priority and 4 is the lowest)

Dropdown cell (row 1, column 2)

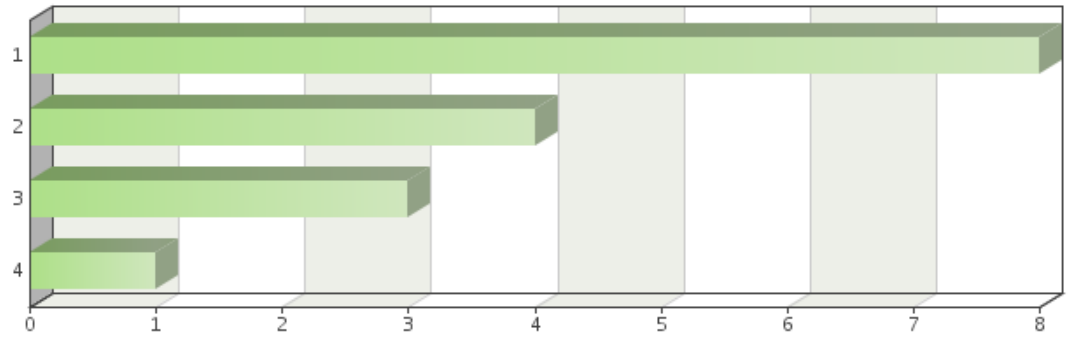


Frequency table

Items	Absolute frequency	Relative frequency	Adjusted relative frequency
1	6	37.5%	37.5%
2	6	37.5%	37.5%
3	2	12.5%	12.5%
4	2	12.5%	12.5%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Dropdown cell (row 2, column 2)

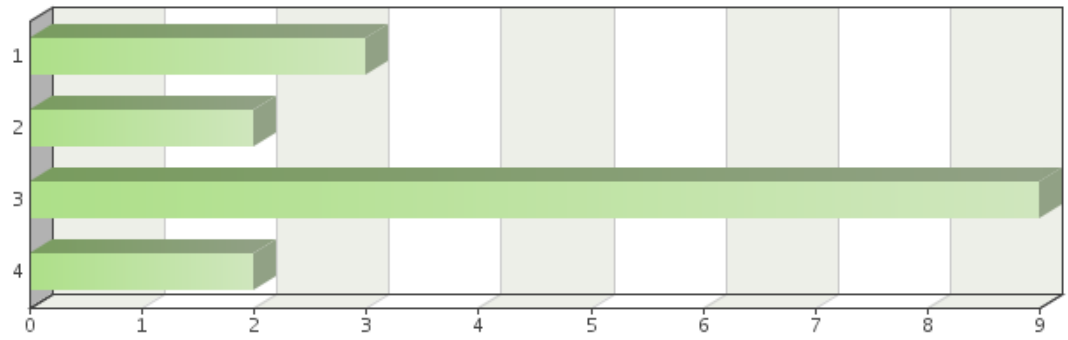


Frequency table

Items	Absolute frequency	Relative frequency	Adjusted relative frequency
1	8	50%	50%
2	4	25%	25%
3	3	18.75%	18.75%
4	1	6.25%	6.25%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Dropdown cell (row 3, column 2)

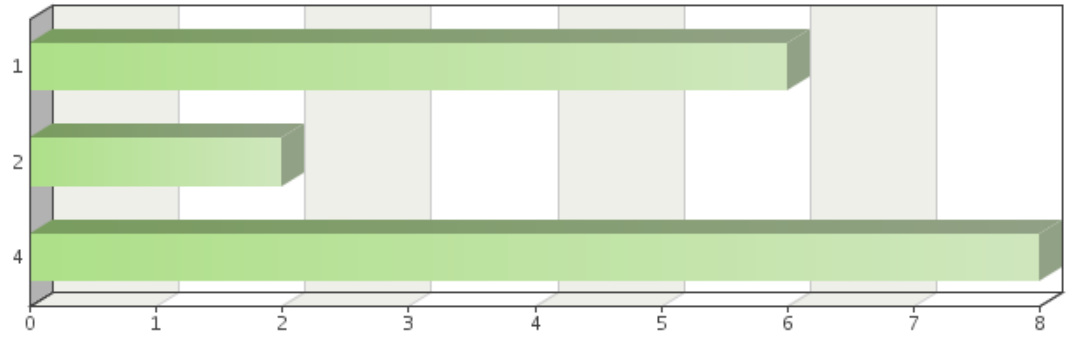


Frequency table

Items	Absolute frequency	Relative frequency	Adjusted relative frequency
1	3	18.75%	18.75%
2	2	12.5%	12.5%
3	9	56.25%	56.25%
4	2	12.5%	12.5%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Dropdown cell (row 4, column 2)



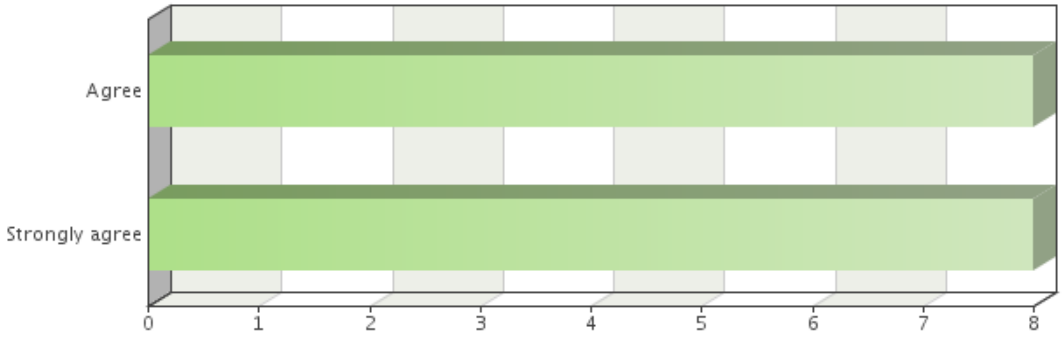
Frequency table

Items	Absolute frequency	Relative frequency	Adjusted relative frequency
1	6	37.5%	37.5%
2	2	12.5%	12.5%
4	8	50%	50%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 27

It is useful for an emulated testing environment to provide data store testing functionality to its system under test.



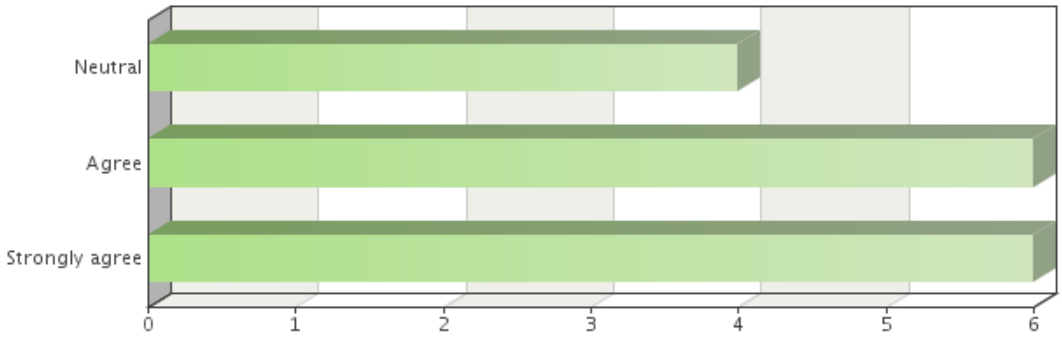
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Agree	8	50%	50%
Strongly agree	8	50%	50%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 28

You prefer to have approximate return results from an emulated testing environment, if its development effort can be reduced significantly.



Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Neutral	4	25%	25%
Agree	6	37.5%	37.5%
Strongly agree	6	37.5%	37.5%
Sum:	16	100%	100%
Not answered:	0	0%	-

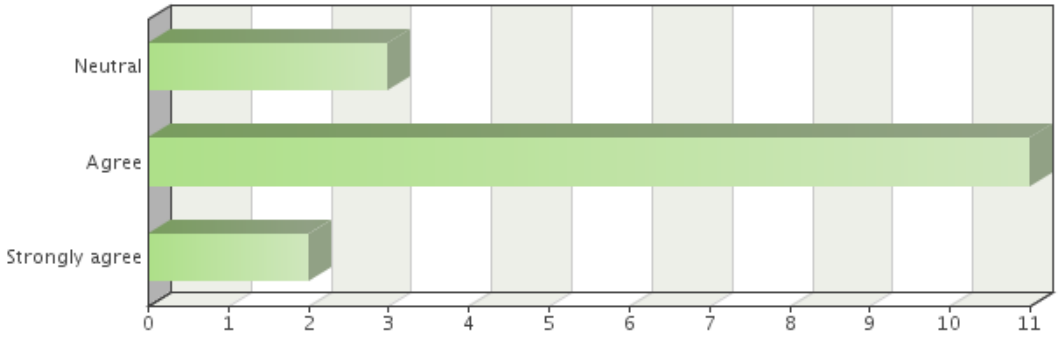
Total answered: 16

Question 29

Your comment on interactive behavior modeling.

Question 30

It is useful for an emulated testing environment to provide non-functional requirement testing features to its system under test.



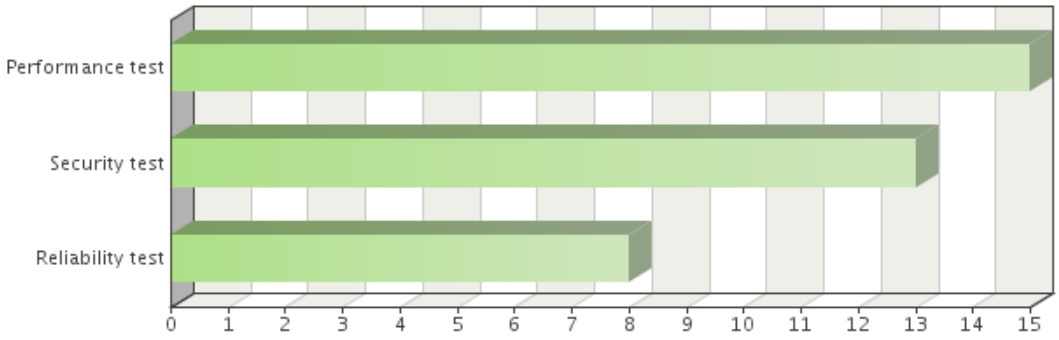
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Neutral	3	18.75%	18.75%
Agree	11	68.75%	68.75%
Strongly agree	2	12.5%	12.5%
Sum:	16	100%	100%
Not answered:	0	0%	-

Total answered: 16

Question 31

What are the non-functional requirement testing features an emulated testing environment should provide? (may have multiple selections)



Frequency table

Choices	Absolute frequency	Relative frequency by choice	Relative frequency	Adjusted relative frequency
Performance test	15	41.67%	93.75%	93.75%
Security test	13	36.11%	81.25%	81.25%
Reliability test	8	22.22%	50%	50%
Sum:	36	100%	-	-
Not answered:	0	-	0%	-

Total answered: 16

Question 32

Your comment on non-functional requirement modeling.

Question 33

Thank you very much for taking part in the survey. If you have any questions or issues, please don't hesitate to contact me.

Email: jianliu@swin.edu.au Phone: 0451845630

APPENDIX V

Phase Two Survey Results Report

Usability Report

Table of contents

Report info....1
Question 1: I agree to take part in the survey....2
Question 2: Your gender: 3
Question 3: Your Age: 4
Question 4: How familiar are you withsoftware application inter-connectivity and inter-operability 5
Question 5: How familiar are you with domain-specificmodeling and domain-specific language? 6
Question 6: What best describes your area? 7
Question 7: What is your educational background ? 8
Question 8: Have you completed the assigned task? 9
Question 9: How long did it take you to complete the task? 10
Question 10: How many times have you asked for support? 11
Question 11: Which phase did you stop at? 12
Question 12: You would like to use the tool in your future project. 13
Question 13: You found the tool unnecessarily complex. 14
Question 14: You found the tool was easy to use. 15
Question 15: You would need support to be able to use the tool. 16
Question 16: You found the various features of the tool were well integrated. 17
Question 17: You found there was too much inconsistency in the tool. 18
Question 18: You would image that most people would learn to use the tool very quickly. 19
Question 19: You found the tool very cumbersome to use. 20
Question 20: You felt very confident using the tool. 21
Question 21: You needed to learn a lot of things before you could get going with the tool. 22
Question 22: In your opinion, comparing to a third generation language (e.g. Java) you are familiar 23
Question 23: What would be your main motivations for you to use the tool? (may have multiple selecti....24
Question 24: What would be your main concerns, which could prevent you from using the tool? (may hav....25
Question 25: Is there anything the tool does not let you do that you would like to? 26
Question 26: Your comment on overall usability of the TeeVML tool. 27
Question 27: Endpoint signature is easily modelled by the tool. 28
Question 28: It is easy to visually see various parts and relationships of a message signature. 29
Question 29: It is easy to make changes to message signature model. 30
Question 30: It is easy to make errors or mistakes during message signature definition. 31
Question 31: It is capable of defining all types of message signatures you have seen. 32
Question 32: Are there any message signature notations that should be made clearer for the user? How....33
Question 33: Endpoint protocol is easily modelled by the tool. 34
Question 34: It is easy to visually see all valid service requests and their dependencies on other f....35
Question 35: It is easy to make changes to interactive protocol model. 36
Question 36: It is easy to make errors or mistakes during interactive protocol definition. 37
Question 37: It is capable of defining all interactive protocol scenarios you have seen. 38
Question 38: Are there any interactive protocol notations that should be made clearer for the user? 39
Question 39: Endpoint interactive behavior is easily modelled by the tool. 40
Question 40: It is easy to visually see all inputs/outputs, data store manipulations and behavior lo....41
Question 41: It is easy to make changes to interactive behavior model. 42
Question 42: It is easy to make errors or mistakes during interactive behavior definition. 43

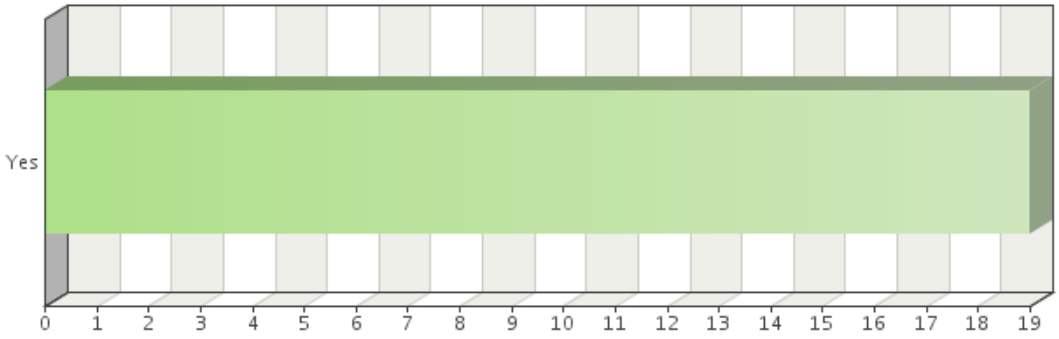
Question 43: The tool has sufficient behavioural expressive power for creating behavior model with a.....44
Question 44: Are there any interactive behavior notations that should be made clearer for the user? 45
Question 45: Your comment on usability of each interface layer sub-language of the TeeVML tool. 46
Question 46: Thank you for taking part in this survey. If you have any questions or issues, please 47

Report info

Report date:	Friday, March 4, 2016 2:22:48 PM EST
Start date:	Wednesday, January 13, 2016 3:37:00 PM EST
Stop date:	Friday, April 1, 2016 3:37:00 PM EST
Number of completed responses:	19

Question 1

I agree to take part in the survey



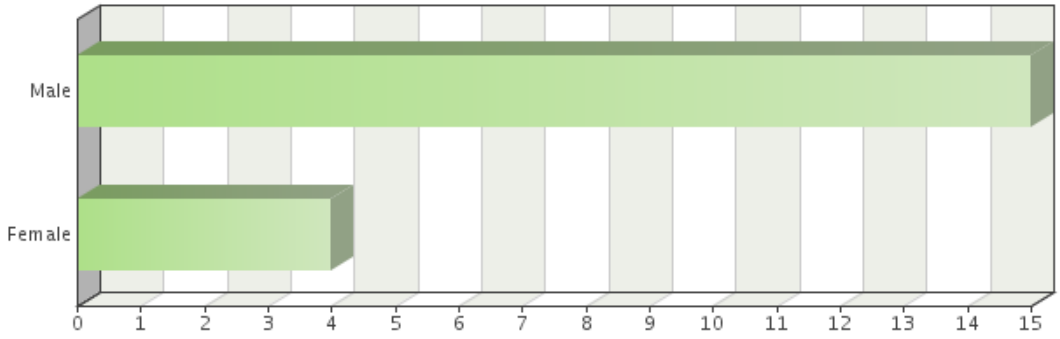
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Yes	19	100%	100%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 2

Your gender:



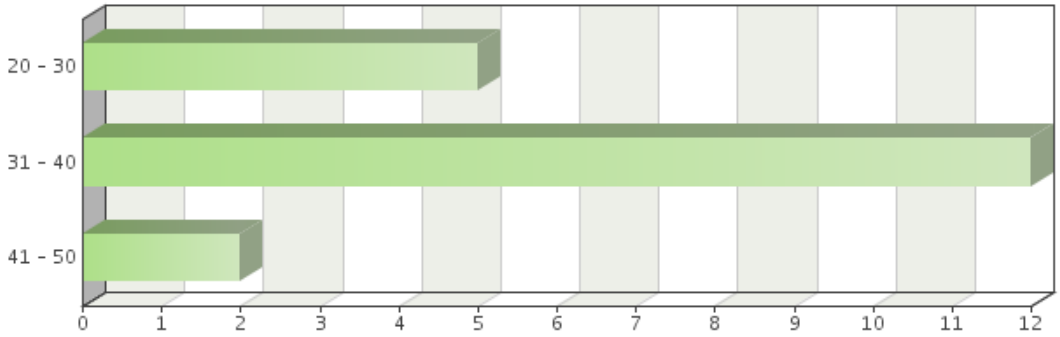
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Male	15	78.95%	78.95%
Female	4	21.05%	21.05%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 3

Your Age:



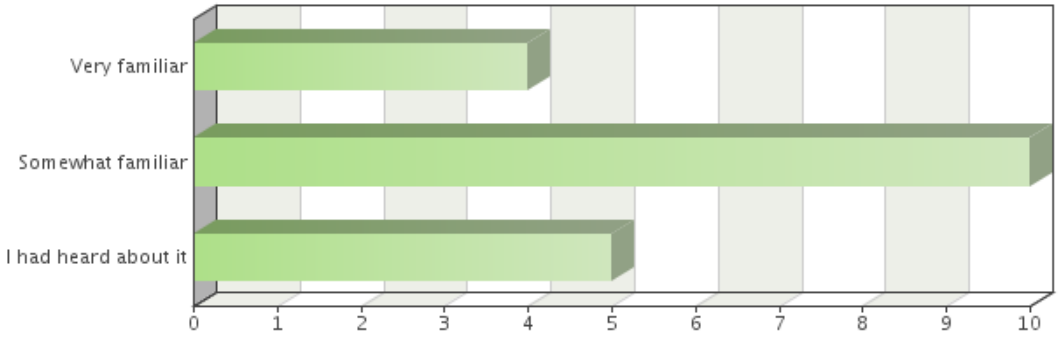
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
20 - 30	5	26.32%	26.32%
31 - 40	12	63.16%	63.16%
41 - 50	2	10.53%	10.53%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 4

How familiar are you withsoftware application inter-connectivity and inter-operability test?



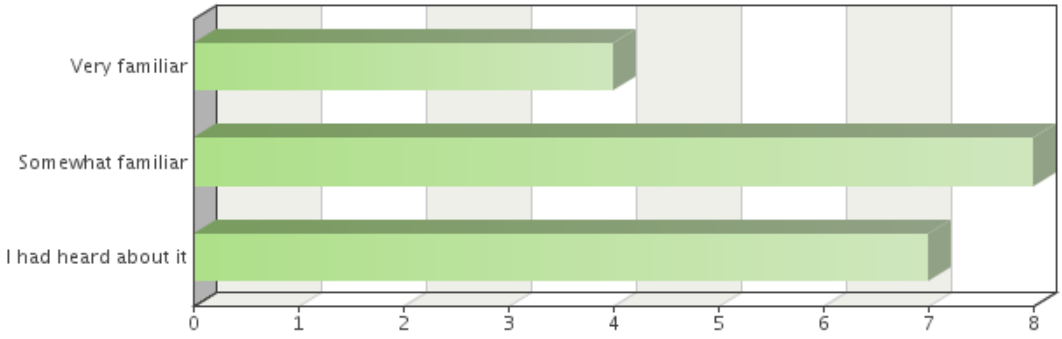
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Very familiar	4	21.05%	21.05%
Somewhat familiar	10	52.63%	52.63%
I had heard about it	5	26.32%	26.32%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 5

How familiar are you with domain-specificmodeling and domain-specific language?



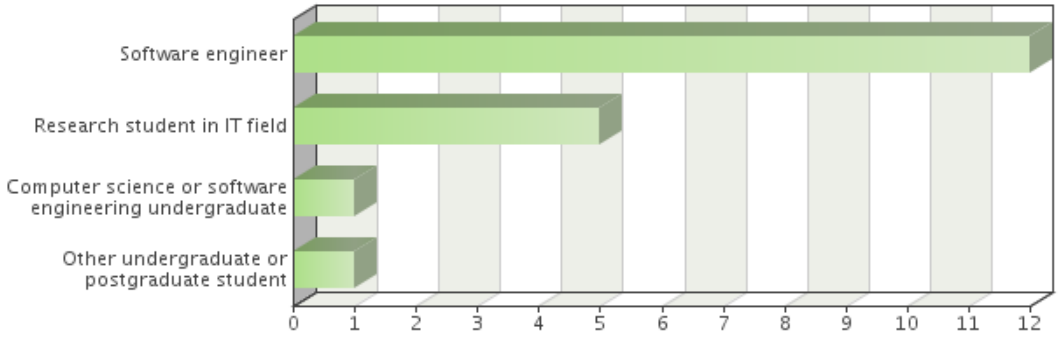
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Very familiar	4	21.05%	21.05%
Somewhat familiar	8	42.11%	42.11%
I had heard about it	7	36.84%	36.84%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 6

What best describes your area?



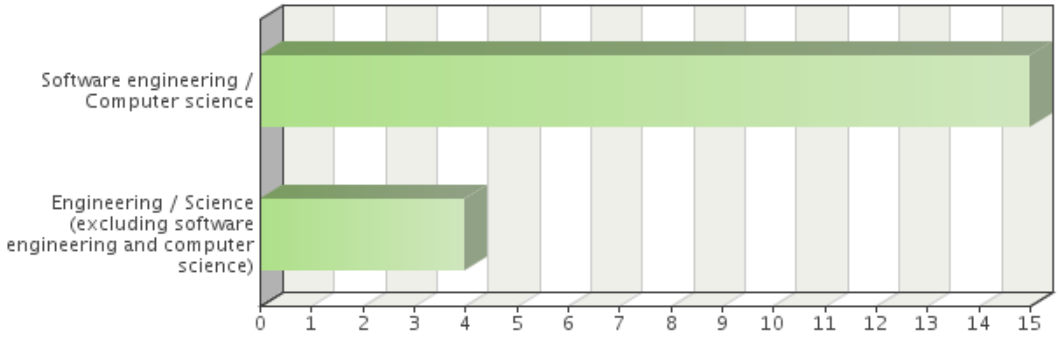
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Software engineer	12	63.16%	63.16%
Research student in IT field	5	26.32%	26.32%
Computer science or software engineering undergraduate	1	5.26%	5.26%
Other undergraduate or postgraduate student	1	5.26%	5.26%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 7

What is your educational background ?



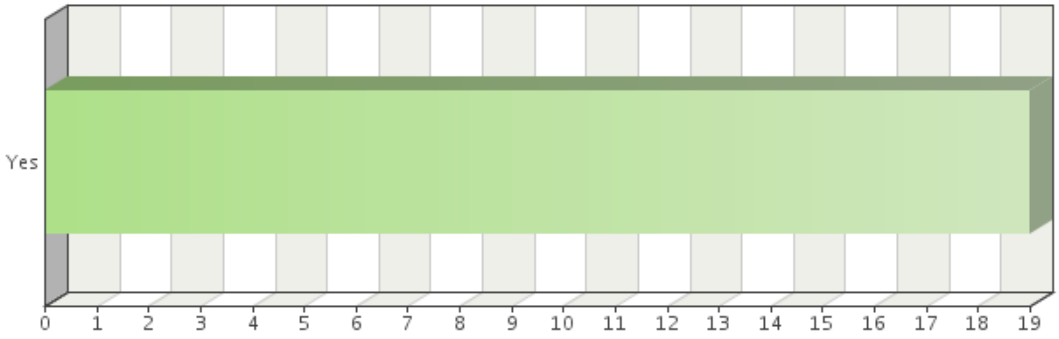
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Software engineering / Computer science	15	78.95%	78.95%
Engineering / Science (excluding software engineering and computer science)	4	21.05%	21.05%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 8

Have you completed the assigned task?



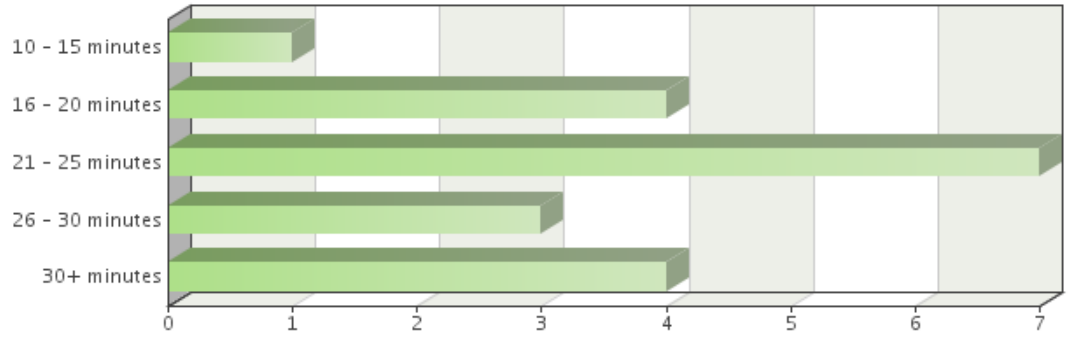
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Yes	19	100%	100%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 9

How long did it take you to complete the task?



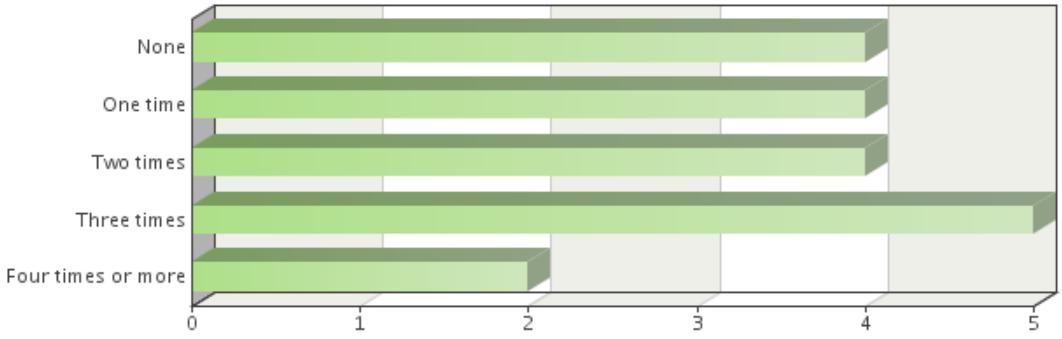
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
10 - 15 minutes	1	5.26%	5.26%
16 - 20 minutes	4	21.05%	21.05%
21 - 25 minutes	7	36.84%	36.84%
26 - 30 minutes	3	15.79%	15.79%
30+ minutes	4	21.05%	21.05%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 10

How many times have you asked for support?



Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
None	4	21.05%	21.05%
One time	4	21.05%	21.05%
Two times	4	21.05%	21.05%
Three times	5	26.32%	26.32%
Four times or more	2	10.53%	10.53%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

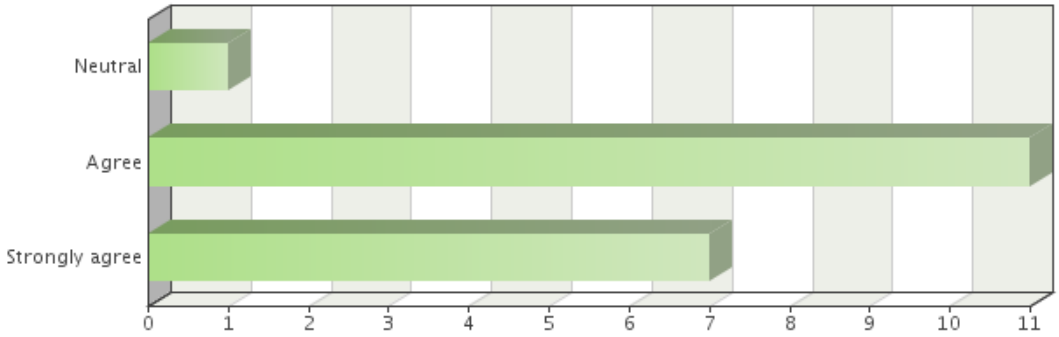
Question 11

Which phase did you stop at?

No data to report

Question 12

You would like to use the tool in your future project.



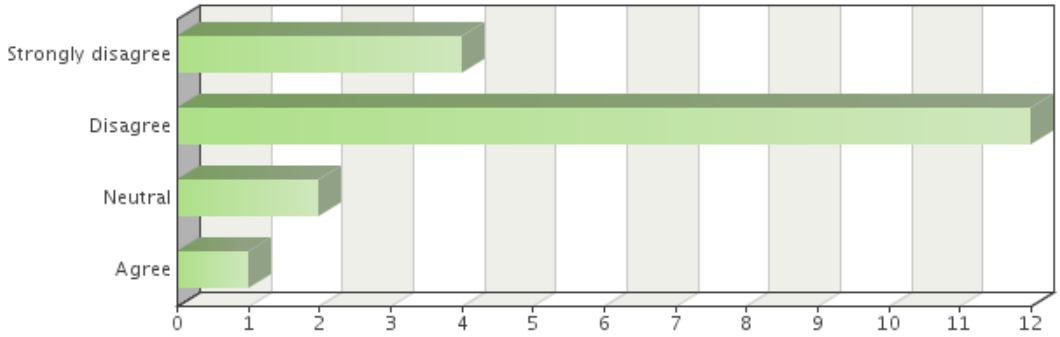
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Neutral	1	5.26%	5.26%
Agree	11	57.89%	57.89%
Strongly agree	7	36.84%	36.84%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 13

You found the tool unnecessarily complex.



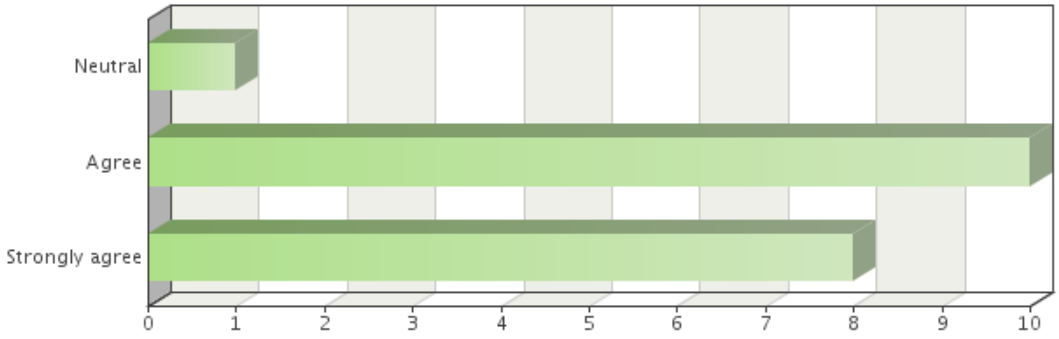
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Strongly disagree	4	21.05%	21.05%
Disagree	12	63.16%	63.16%
Neutral	2	10.53%	10.53%
Agree	1	5.26%	5.26%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 14

You found the tool was easy to use.



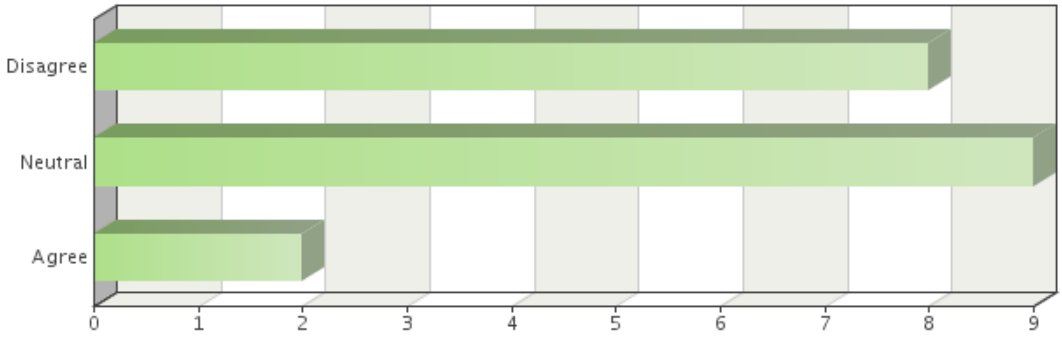
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Neutral	1	5.26%	5.26%
Agree	10	52.63%	52.63%
Strongly agree	8	42.11%	42.11%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 15

You would need support to be able to use the tool.



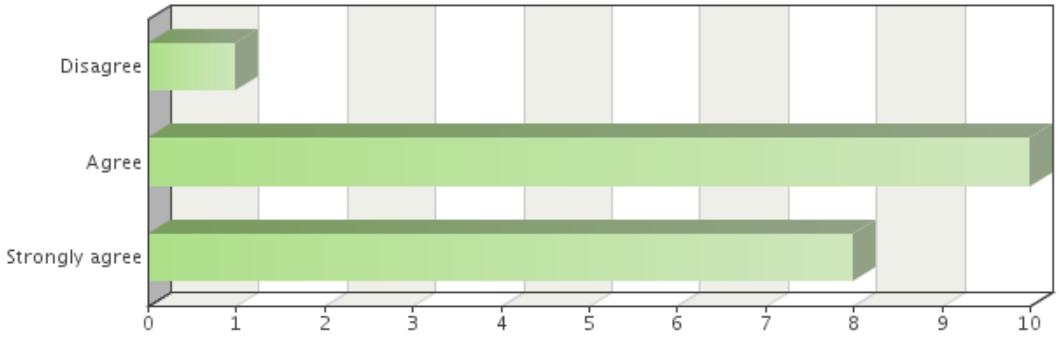
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Disagree	8	42.11%	42.11%
Neutral	9	47.37%	47.37%
Agree	2	10.53%	10.53%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 16

You found the various features of the tool were well integrated.



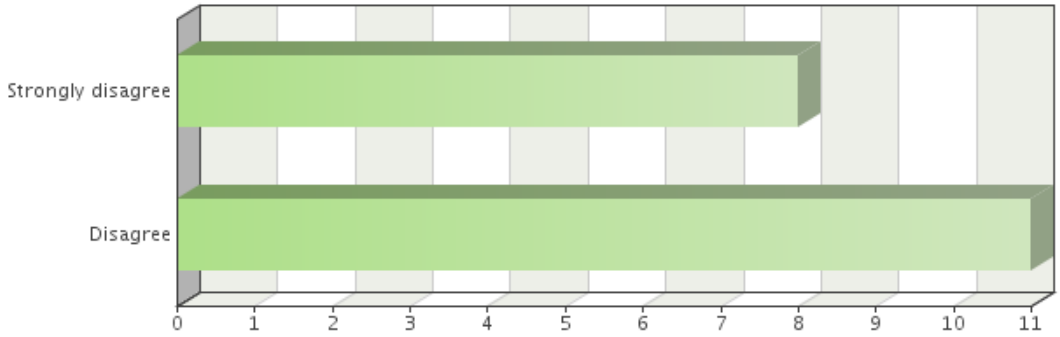
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Disagree	1	5.26%	5.26%
Agree	10	52.63%	52.63%
Strongly agree	8	42.11%	42.11%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 17

You found there was too much inconsistency in the tool.



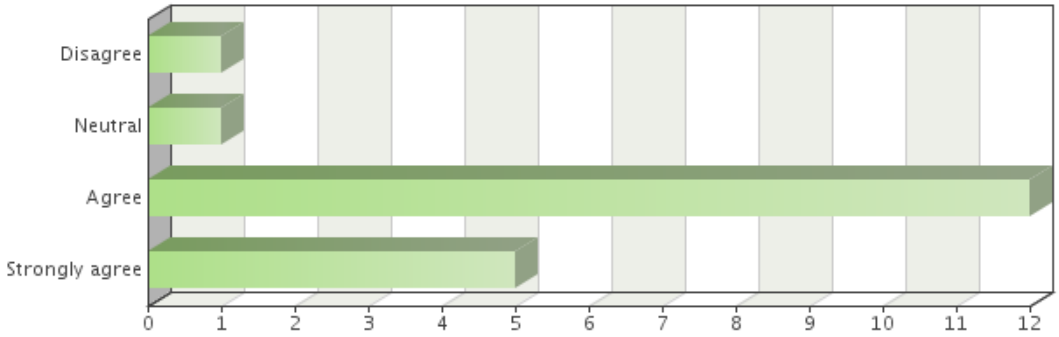
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Strongly disagree	8	42.11%	42.11%
Disagree	11	57.89%	57.89%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 18

You would image that most people would learn to use the tool very quickly.



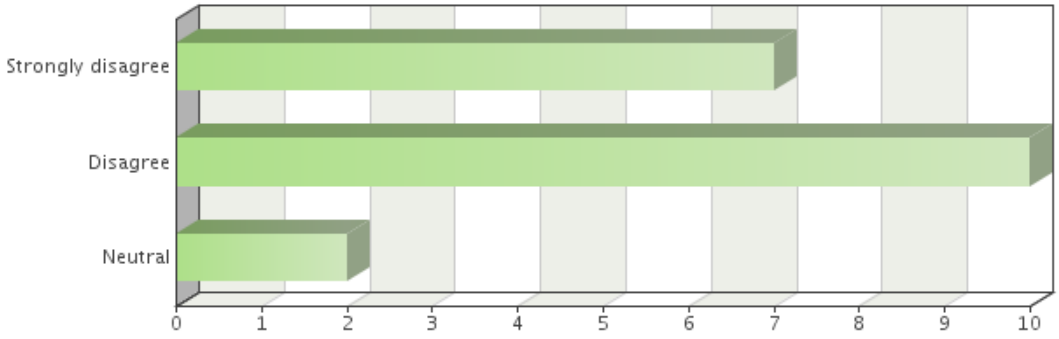
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Disagree	1	5.26%	5.26%
Neutral	1	5.26%	5.26%
Agree	12	63.16%	63.16%
Strongly agree	5	26.32%	26.32%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 19

You found the tool very cumbersome to use.



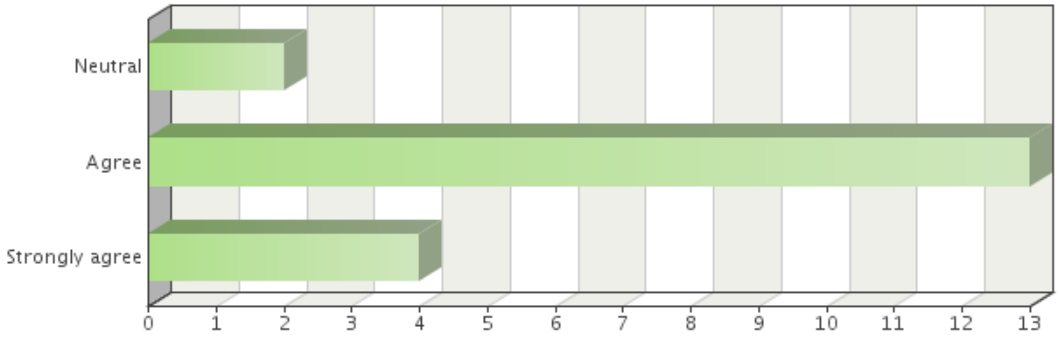
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Strongly disagree	7	36.84%	36.84%
Disagree	10	52.63%	52.63%
Neutral	2	10.53%	10.53%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 20

You felt very confident using the tool.



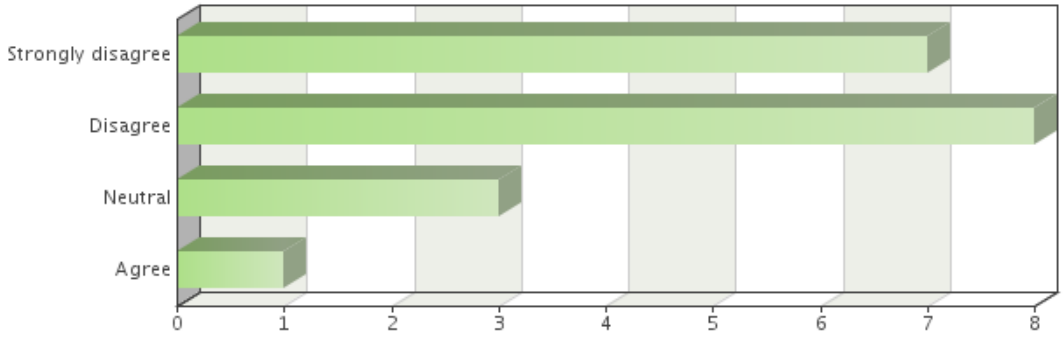
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Neutral	2	10.53%	10.53%
Agree	13	68.42%	68.42%
Strongly agree	4	21.05%	21.05%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 21

You needed to learn a lot of things before you could get going with the tool.



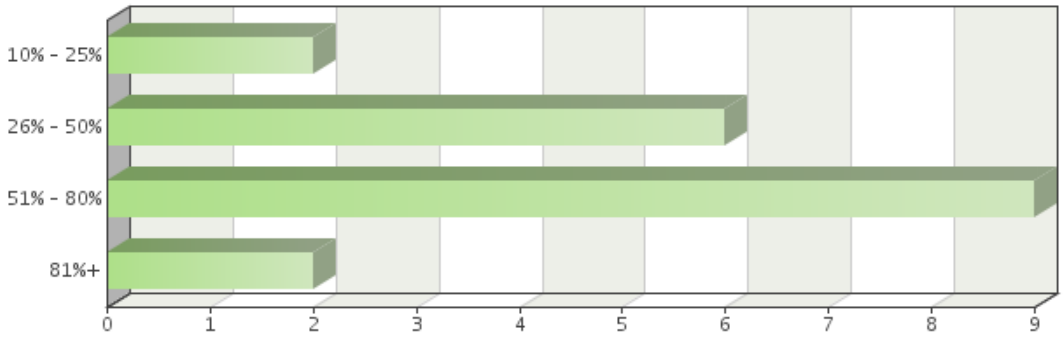
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Strongly disagree	7	36.84%	36.84%
Disagree	8	42.11%	42.11%
Neutral	3	15.79%	15.79%
Agree	1	5.26%	5.26%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 22

In your opinion, comparing to a third generation language (e.g. Java) you are familiar with, how much would a typical endpoint development effort be reduced by using the tool?



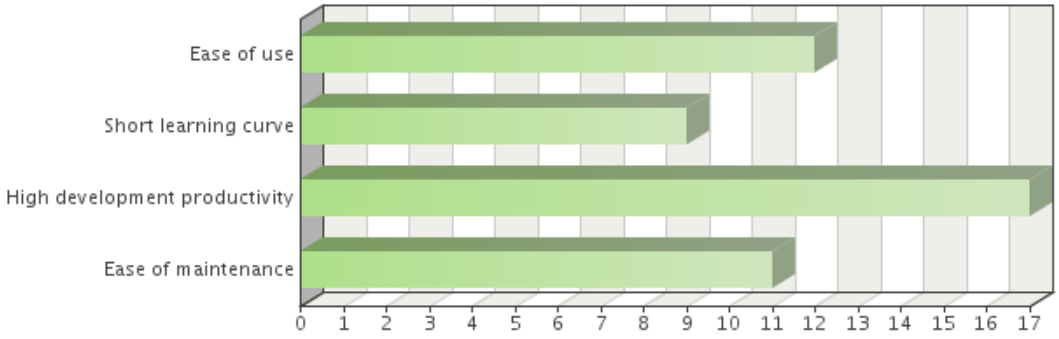
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
10% - 25%	2	10.53%	10.53%
26% - 50%	6	31.58%	31.58%
51% - 80%	9	47.37%	47.37%
81%+	2	10.53%	10.53%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 23

What would be your main motivations for you to use the tool? (may have multiple selections)



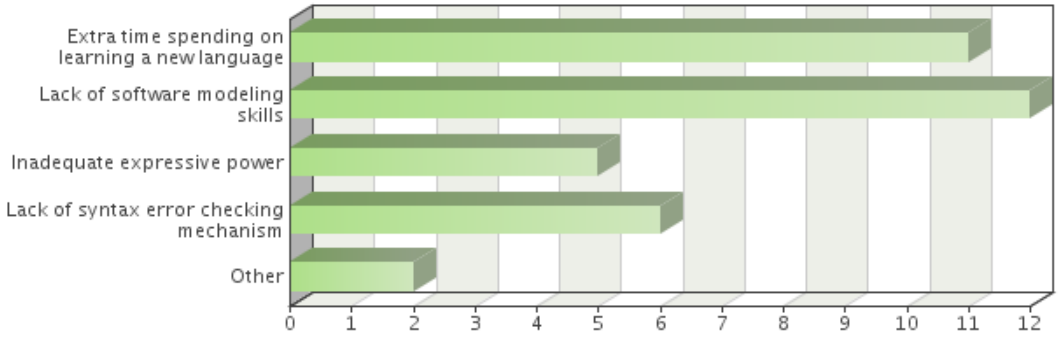
Frequency table

Choices	Absolute frequency	Relative frequency by choice	Relative frequency	Adjusted relative frequency
Ease of use	12	24.49%	63.16%	63.16%
Short learning curve	9	18.37%	47.37%	47.37%
High development productivity	17	34.69%	89.47%	89.47%
Ease of maintenance	11	22.45%	57.89%	57.89%
Sum:	49	100%	-	-
Not answered:	0	-	0%	-

Total answered: 19

Question 24

What would be your main concerns, which could prevent you from using the tool? (may have multiple selections)



Frequency table

Choices	Absolute frequency	Relative frequency by choice	Relative frequency	Adjusted relative frequency
Extra time spending on learning a new language	11	30.56%	57.89%	57.89%
Lack of software modeling skills	12	33.33%	63.16%	63.16%
Inadequate expressive power	5	13.89%	26.32%	26.32%
Lack of syntax error checking mechanism	6	16.67%	31.58%	31.58%
Other	2	5.56%	10.53%	10.53%
Sum:	36	100%	-	-
Not answered:	0	-	0%	-

Total answered: 19

Question 25

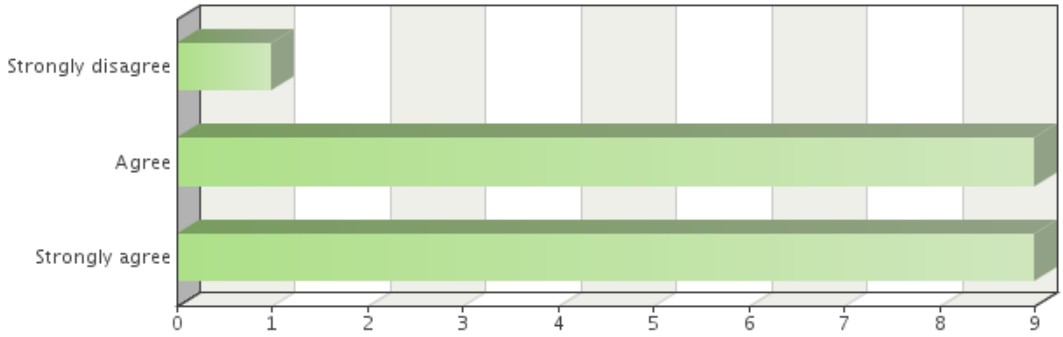
Is there anything the tool does not let you do that you would like to?

Question 26

Your comment on overall usability of the TeeVML tool.

Question 27

Endpoint signature is easily modelled by the tool.



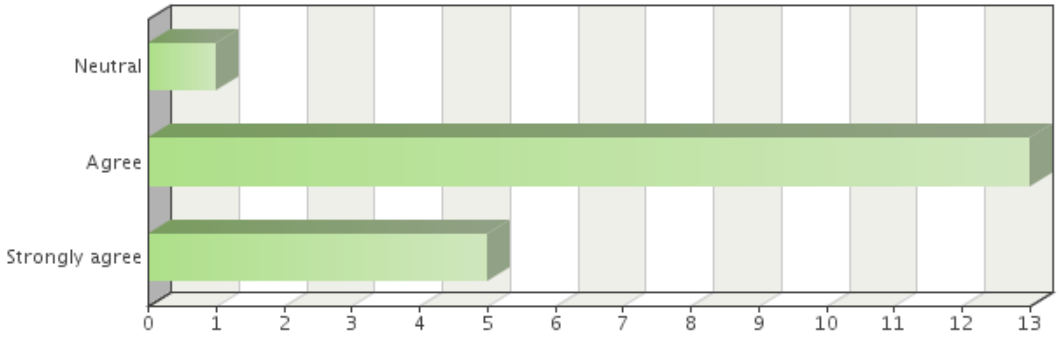
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Strongly disagree	1	5.26%	5.26%
Agree	9	47.37%	47.37%
Strongly agree	9	47.37%	47.37%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 28

It is easy to visually see various parts and relationships of a message signature.



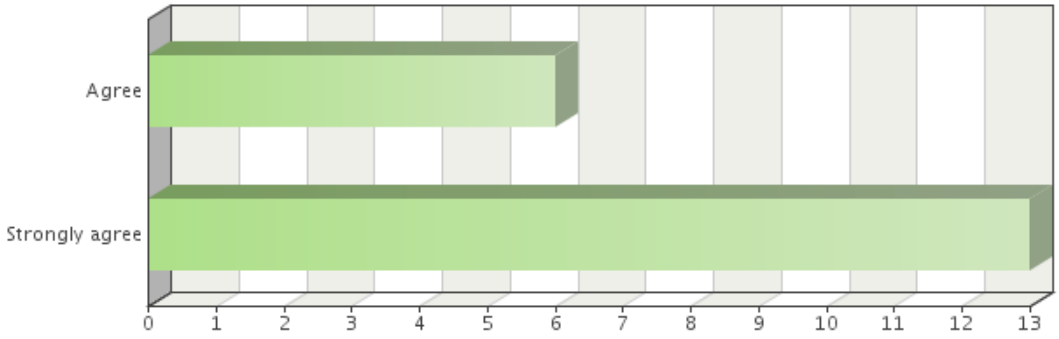
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Neutral	1	5.26%	5.26%
Agree	13	68.42%	68.42%
Strongly agree	5	26.32%	26.32%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 29

It is easy to make changes to message signature model.



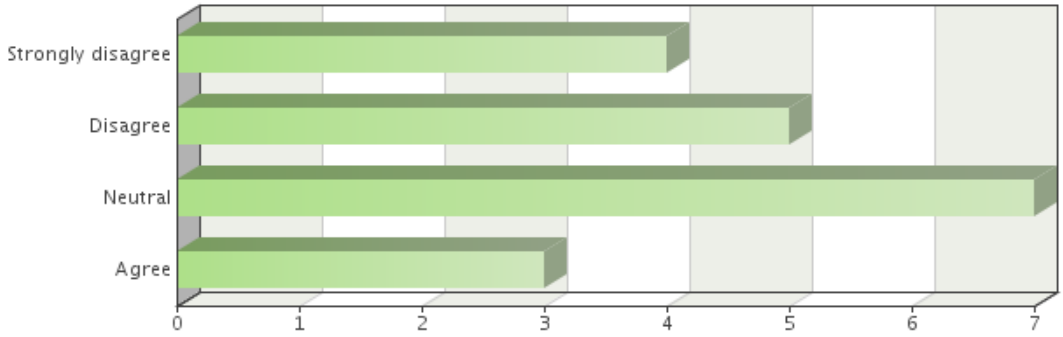
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Agree	6	31.58%	31.58%
Strongly agree	13	68.42%	68.42%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 30

It is easy to make errors or mistakes during message signature definition.



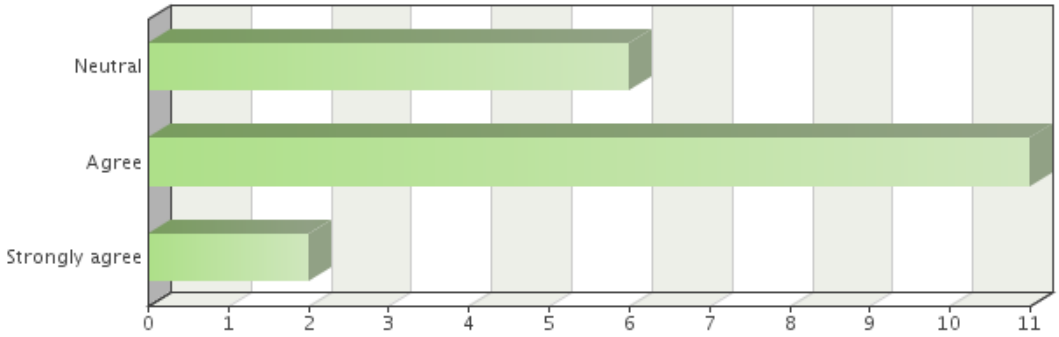
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Strongly disagree	4	21.05%	21.05%
Disagree	5	26.32%	26.32%
Neutral	7	36.84%	36.84%
Agree	3	15.79%	15.79%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 31

It is capable of defining all types of message signatures you have seen.



Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Neutral	6	31.58%	31.58%
Agree	11	57.89%	57.89%
Strongly agree	2	10.53%	10.53%
Sum:	19	100%	100%
Not answered:	0	0%	-

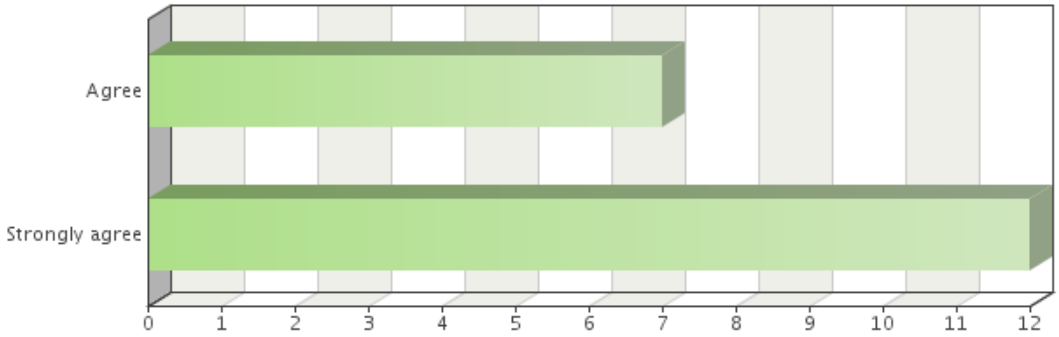
Total answered: 19

Question 32

Are there any message signature notations that should be made clearer for the user? How?

Question 33

Endpoint protocol is easily modelled by the tool.



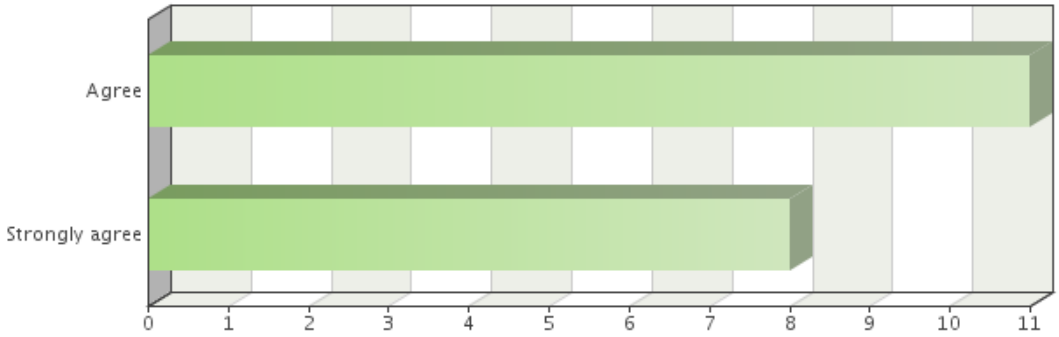
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Agree	7	36.84%	36.84%
Strongly agree	12	63.16%	63.16%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 34

It is easy to visually see all valid service requests and their dependencies on other factors.



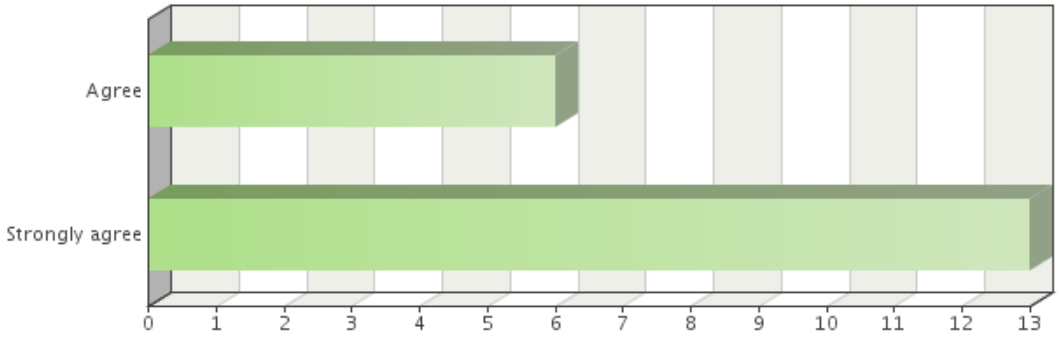
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Agree	11	57.89%	57.89%
Strongly agree	8	42.11%	42.11%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 35

It is easy to make changes to interactive protocol model.



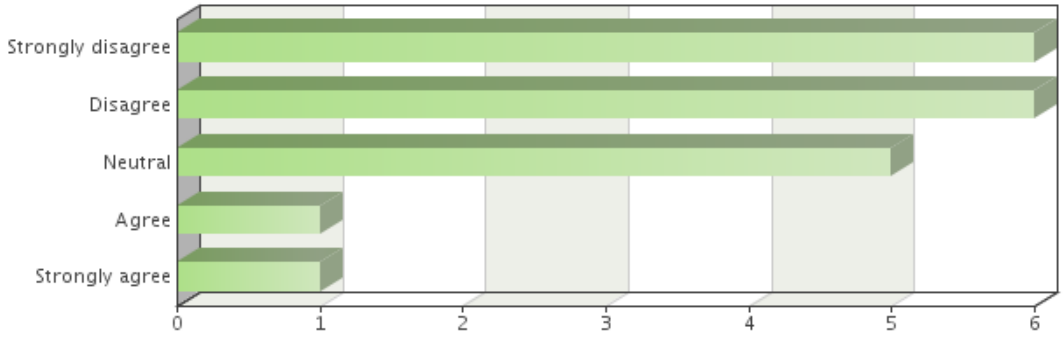
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Agree	6	31.58%	31.58%
Strongly agree	13	68.42%	68.42%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 36

It is easy to make errors or mistakes during interactive protocol definition.



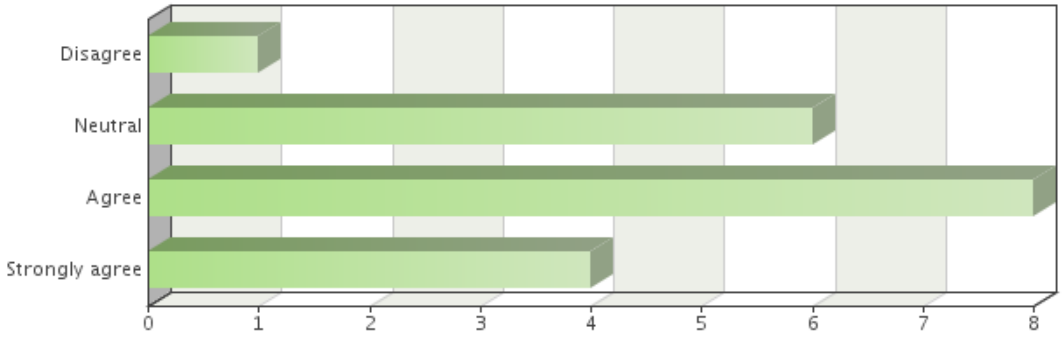
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Strongly disagree	6	31.58%	31.58%
Disagree	6	31.58%	31.58%
Neutral	5	26.32%	26.32%
Agree	1	5.26%	5.26%
Strongly agree	1	5.26%	5.26%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 37

It is capable of defining all interactive protocol scenarios you have seen.



Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Disagree	1	5.26%	5.26%
Neutral	6	31.58%	31.58%
Agree	8	42.11%	42.11%
Strongly agree	4	21.05%	21.05%
Sum:	19	100%	100%
Not answered:	0	0%	-

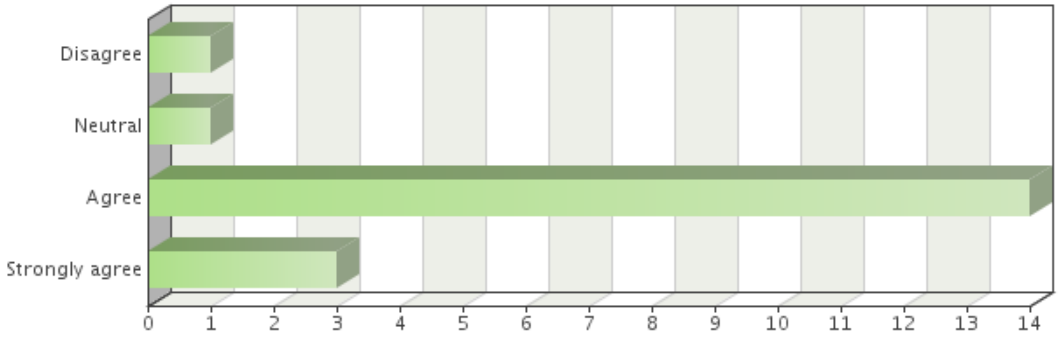
Total answered: 19

Question 38

Are there any interactive protocol notations that should be made clearer for the user? How?

Question 39

Endpoint interactive behavior is easily modelled by the tool.



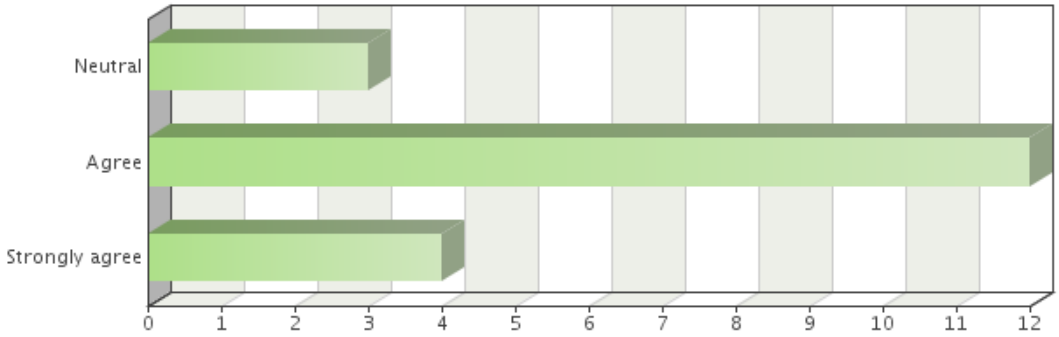
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Disagree	1	5.26%	5.26%
Neutral	1	5.26%	5.26%
Agree	14	73.68%	73.68%
Strongly agree	3	15.79%	15.79%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 40

It is easy to visually see all inputs/outputs, data store manipulations and behavior logic processes.



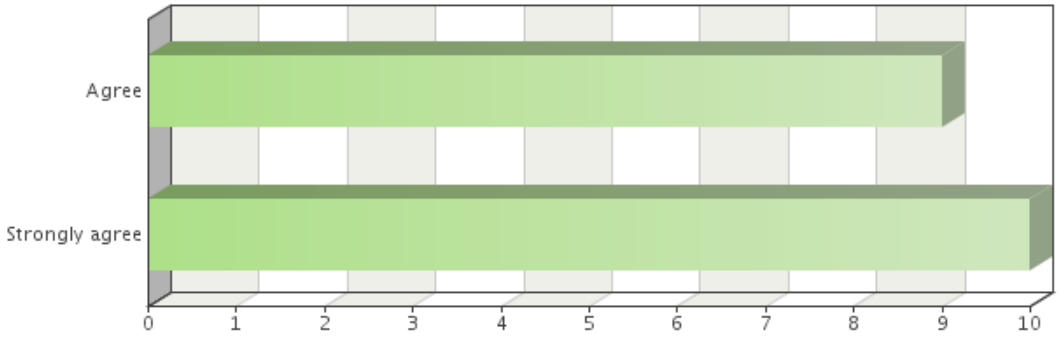
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Neutral	3	15.79%	15.79%
Agree	12	63.16%	63.16%
Strongly agree	4	21.05%	21.05%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 41

It is easy to make changes to interactive behavior model.



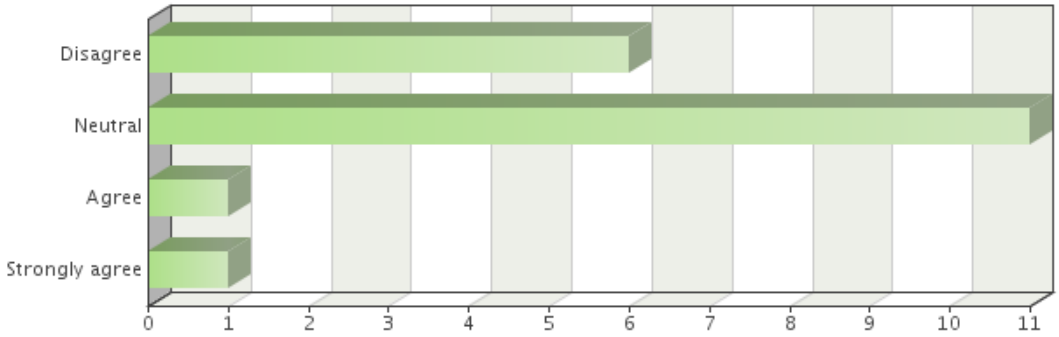
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Agree	9	47.37%	47.37%
Strongly agree	10	52.63%	52.63%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 42

It is easy to make errors or mistakes during interactive behavior definition.



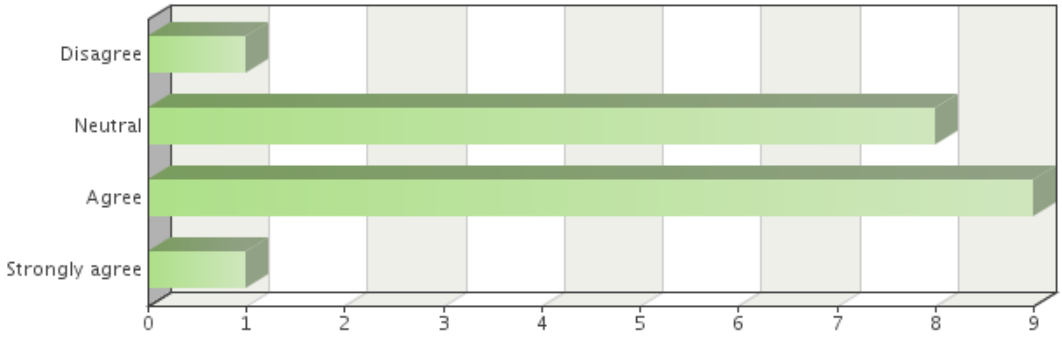
Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Disagree	6	31.58%	31.58%
Neutral	11	57.89%	57.89%
Agree	1	5.26%	5.26%
Strongly agree	1	5.26%	5.26%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 43

The tool has sufficient behavioural expressive power for creating behavior model with accurate outputs?



Frequency table

Choices	Absolute frequency	Relative frequency	Adjusted relative frequency
Disagree	1	5.26%	5.26%
Neutral	8	42.11%	42.11%
Agree	9	47.37%	47.37%
Strongly agree	1	5.26%	5.26%
Sum:	19	100%	100%
Not answered:	0	0%	-

Total answered: 19

Question 44

Are there any interactive behavior notations that should be made clearer for the user? How?

Question 45

Your comment on usability of each interface layer sub-language of the TeeVML tool.

Question 46

Thank you for taking part in this survey. If you have any questions or issues, please don't hesitate to contact me. Email: jianliu@swin.edu.au Phone: 0451845630