Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Computer Science

Model Transformation Approach to Automated Model Driven Development

Nguyen Viet Cuong

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor

Ph.D. Programme: Electrical Engineering and Information Technology

Branch of study: Information Science and Computer Engineering

Supervisor : doc. Ing. Karel Richta, CSc.

Co-supervisor : doc. Ing. Vojtěch Merunka, Ph.D.

Prague, February 2015

Thesis Supervisor:

DOC. ING. KAREL RICHTA, CSC.

Department of Computer Science

Faculty of Electrical Engineering

Czech Technical University in Prague

Karlovo nám. 13, 121 35 Praha 2

Czech Republic.

Thesis Co-Supervisor:

DOC. ING. VOJTĚCH MERUNKA, PH.D.

Faculty of Economics and Management

Czech University of Life Sciences Prague

Kamýcká 129, 16521 Praha 6

Czech Republic.

Acknowledgments

I would like to express my gratitude toward my supervisors, doc. Ing. Karel Richta, CSc. and doc. Ing. Vojtěch Merunka, Ph.D., without whose assistance and support this thesis would not have been possible.

I would also like to thank my parents for their love, encouragements and faith in me, thank you Mom and Dad for giving me the chance to pursue my study in another country. I appreciate the help from my close ones and tireless support from my brother for taking care of my parents for such a long time when I am away from my hometown.

I would like to thank my wife for giving me the strength to complete the work every time I think about giving up. Many thanks also go to all my friends and colleagues that helped me throughout this project.

Abstract

One of the contemporary challenges of software evolution is to adapt a software system to the changing of requirements and demands from users and environments. An ultimate goal is to encapsulate these requirements into a high-level abstraction, giving the ability to achieve large-scale adaptation of the underlying software implementation. Model-Driven Engineering (MDE) is one of the enabling techniques that supports this objective. In MDE, the effective creation of models and their transformation are core activities to enable the conversion of source models to target models in order to change model structures or translate models to other software artifacts. The main goal is to provide automation and enable the automated development of a system from its corresponding models. There are several approaches on this matter from high level. However, there is still absence of clear methodology and results on how to apply MDE for a specific domain with specific requirements such as the web domain. This research brings contribution toward the solution to automated model development by providing an overview of existing approaches and introducing a novel approach in the emerging field of web applications and services.

To cope with current trend in the growing of complexity of web services as programmatic backbones of modern distributed and cloud architecture, we present an approach using domain specific language for modeling of web services as the solution to the challenge in scalability of web service modeling and development. We analyze the current state of the problem domain and implement a domain specific language called Simple Web Service Modeling to support automated model-driven development of such web services. This approach is the solution to the problem in web service development of software-as-service systems that require the support for tenant-specific architecture.

In the domain of web application quality assurance, we build a modeling language for model driven testing of web application that focuses on automation and regression testing. Our techniques are based on building abstractions of web pages and modeling state-machine-based test behavior using Web Testing Modeling Language - a domain specific language that we developed for web page modeling. This methodology and techniques aim at helping software developers as well as testers to become more productive and reduce the time-to-market, while maintaining high standards of web application. The proposing techniques is the answer to the lack of concrete methods and toolset in applying model driven development to specific areas such as web application testing and services. The results of this work can be applied to practical purposes with the methodological support to integrate into existing software development practices.

Contents

List of Figures

Chapter 1

Introduction

Along with the growing software market in recent years, business requirements are changing more rapidly which leads to the continuously growing in the complexity of enterprise applications. This makes it hard to find the case nowadays which we can deliver a software system with the assumption that one-size-fits-all [61]. The complex nature of software systems created more challenges in software development and maintenance. It is well-known from the work of Dijkstra [20] that the techniques addressing the complexity of software development include the principles of abstraction, information hiding, encapsulation and decomposition. Model Driven Engineering (MDE) addresses the complexity problem with the approach to raise the abstraction level by focusing on the creation of models, aiming to increase productivity by maximizing compatibility between systems, simplifying the process of design and promoting communication between individuals and teams working on the system.

The movement towards software development with the use of models has been increased rapidly over the past several years. Organizations are increasingly seizing the opportunity to move their intellectual property and business logic from source code into models, allowing them to focus on the important aspects of their systems. The introduction of models has opened up new possibilities for creating, analyzing, and manipulating systems through various types of tools. Model driven approaches enable architects to achieve a solution where communication ability is promoted while creating artifacts that become parts of the overall solution.

Recent trend in the software development strategy envisions cloud computing and distributed systems continue to gain more mainstream adoption as more companies move into the cloud. With mobile experiences gradually gain space against the desktop experience, cloud computing continues to accelerate and become more and more significant [34]. Big data may be competing with cloud computing for the tech news headlines, but many providers and businesses are now starting to see the value in combining the two. “Big data as a service” seems like one of the most practical options for big data analytics, as it is scalable and within the reach of any organization, no matter its size or resources. These cloud providers are also overcoming the technical barrier by transforming Hadoop from an open source platform to an enterprise-ready service [3]. All of these reasons force cloud and distributed systems to constantly grow in their complexities with external factors are becoming more unpredictable [64]. From this standpoint, Model-driven Engineering methodologies have been applied as a solution for better reaction to the market trend and aims to increase efficiency and bring more agility to the development life cycle. However, with the regularity of the different applicable domains in web applications, it is unattainable to finalize a method or approach that could fit in any situation. In the next section we look at some of the problems and key challenges in this field.

1.1 Key challenges

Computer-aided software engineering (CASE) was a popular effort to apply set of methods and tools to help developers in the process of analyzing and developing systems with the help of general purpose and graphical representations. This concept aims to create the software systems with high-quality, defect-free and maintainable. However, there are limitations to this approach, there are difficulties in linking CASE tools from different vendors since each of them were constrained to a different notations with different code structures and data classifications. Another challenge in this context is to obtain automation in software development. The major obstacle in achieving such automation exists due to the lack of effective modeling and transformation technologies [21]. General purpose modeling often provides a set of general elements applicable to any situation. This leads to abstractions that may not be needed in every domain, hence adds redundancy and obstacle in effective usage. To address the issue, there is a need for an approach to software development that can focus on higher level specifications of programs in Domain-Specific Modeling Languages, offering greater degrees of automation, and embrace effectively the domain knowledge.

The challenge in applying domain specific language in modeling is to define correct standards and tool sets that can integrate effectively. There have been standards defined by the Object Management Group from a high-level. The challenge, however, still remains in various phases of the process to provide not only the appropriate set of technologies and tools but also best practice methods that could react to the frequent change in business requirement within various phases of model evolution.

Model Driven Architecture (MDA) aims to separate application structure - Platform Independent Model (PIM) from its functionality - Platform Specific Model (PSM). The mapping between these models is realized by model transformation. The problem of model transformation based on Meta-Object Facility (MOF) can, then, be stated in the following way: "*Given a source model 'm1' described by a meta-model 'MM1' we define an automatic process making it possible to obtain a model 'm2' conforming to a meta-model 'MM2'*" [17]. Model transformations require specialized support in order to realize their full potential. There are still open issues in their foundations, semantics, structuring mechanisms that demand further research and study. Model transformations also require methodological support to integrate into existing software development practices. Further more, MDA provides the concepts from a high level, there is an absence of clear methodology and results on how to apply MDA for a specific domain with specific requirements such as the web domain. In areas such as model driven development of web services, current methods often focus on the usage of a generic modeling language such as UML, which leads to complex class diagrams and obstacle in achieving automation. This creates several issues in software reuse, development speed and cost-effectiveness, which need to be resolved.

1.2 Contributions of the thesis

This research brings a contribution to automated model-driven development by firstly analyzing and studying overview of several model transformation approaches, which emerged from practical case studies. This includes the taxonomy study which can help in giving an overview of the techniques and their best-suited application domains. With a more comprehensive overview in the domain of interest, this can help a developer in making decision of choosing the approach that is best suited for his requirements.

Main result of this work is the introduction of an approach to the development of web services by using model driven techniques with domain specific language. As a result, a DSL for modeling of web services called SWSM (Simple Web Service Modeling) is developed and introduced. To demonstrate this approach, a case study of web service development from modeling to code generation is also illustrated with the associated techniques. This approach aims at improving productivity and maintainability of web services by raising the level of abstraction from source code in a general purpose language to high-level, domain-specific models such that developers can concentrate on application logic rather than the complexity of low-level implementation details. This design addresses the underlying challenges by providing support for software reuse and development speed via simple syntax, better code readability and easier program integration.

As a solution for quality assurance of web applications, we build a framework for model driven testing of web application that focuses on automation and regression testing. Our techniques are based on building abstractions of web pages and modeling state-machine-based test behavior using Webpage Testing Modeling Language (WTML) - a domain specific language that we developed for web page modeling. These techniques aim to reduce the time-to-market and maintain high standards of the application by identifying in advance possible faults with automated test case generation and execution. This is the answer to the lack of concrete methods and toolset in applying model driven development to specific areas such as web application testing and services.

1.3 Structure of the thesis

The thesis is organized as follows. Chapter 2 introduces basic definitions, terminology and background. Chapter 3 summarizes and gives overview on the state of the art in model transformation, model driven development and previous related results. In this chapter, we also describe several existing approaches and some results from our case studies. Chapter 4 addresses the challenge in development of fast-growing web platforms and Platform-as-a-Service architecture by introducing our approach to the development of web services using domain specific language and model driven techniques. We present our effort to implement SWSM as a DSL to support modeling and automation. Chapter 5 demonstrates our approach on quality assurance of web application, which focuses on automation and regression testing by implementing a framework for model driven testing. We present our techniques in modeling web pages and state-machine-based test behavior using WTML - a DSL that we developed for this purpose. Chapter 6 concludes and outlines the future work.

Chapter 2

Background

2.1 Model Driven Engineering

A model is a simplified representation of an aspect of the world for a specific purpose. Nowadays, in many complex systems, a lot of aspects need to be considered from architectural to dynamic behaviors, functionalities and user interfaces. In Model-Driven Engineering, all aspects are put together and presented as models. The design process can be described as the weaving of all these aspects into a detailed design model. Model-driven methods aim at automating this weaving process. Model-driven development (MDD) is a software development methodology which focuses on creating models, or abstractions of something that describes the elements of a system. It is meant to increase productivity by maximizing compatibility between systems, simplifying the process of design and promoting communication between individuals and teams working on the system [74].

The promise of MDA [31] is to allow definition of machine-readable applications and data models that enable long-term flexibility with regards to implementation, integration, maintenance, testing and simulation. MDA could be seen as a framework that defines the architecture for models. Based on this architecture, models represent a part of the system's functionality, structure or behavior. Moreover a consistent structure of these models is defined.

MDA defines two main modeling structures, namely Platform Independent Models (PIMs) and Platform Specific Models (PSMs). PIMs provide formal specifications of the structure and/or functionality of a given system, leaving out technical particulars about that system. PSM then deals with such implementation-specific concerns. This viewpoint of models is very important and beneficent for many reasons. Such approach makes it easier to reproduce implementations of the same model into different platforms, when they are referring to the same initial structure with no platform-specific semantics. Validation of such model is also easier then when platform-specific concepts are embodied into the implementation. Given such consistency, it may be possible under certain circumstances to automatically transform a PIM into different target PSMs, using some mappings and patterns and attaching some platform-dependent information into the process. PIMs, PSMs and mappings are based on metamodel concepts, usually expressed in some core OMG technologies such as Unified Modeling Language (UML), Meta Object Facility (MOF) [33] or Common Warehouse Metamodel (CWM) [30]. A metamodel is actually a model that describes another model. This layered architecture is also defined by MOF as seen in Fig 2.1.

2.2 Essential concepts in modeling and meta-modeling

2.2.1 Model and meta-model

A model can be defined as a simplification of the subject, and its purpose is to answer some particular questions aimed towards the subject or simply something we want to reason about. A model can give us the view of attributes such as functionality, time constrains, security etc. In the development process, models are developed through extensive communication among product managers, designers, and members of the development team. As the models approach completion, they enable the development of software and systems.

To work with each model, we need to have information about which exact properties and the relations it may hold. In other words, we need to know the structure of the model. This information is expressed in the metamodel, where a metamodel is a model that makes

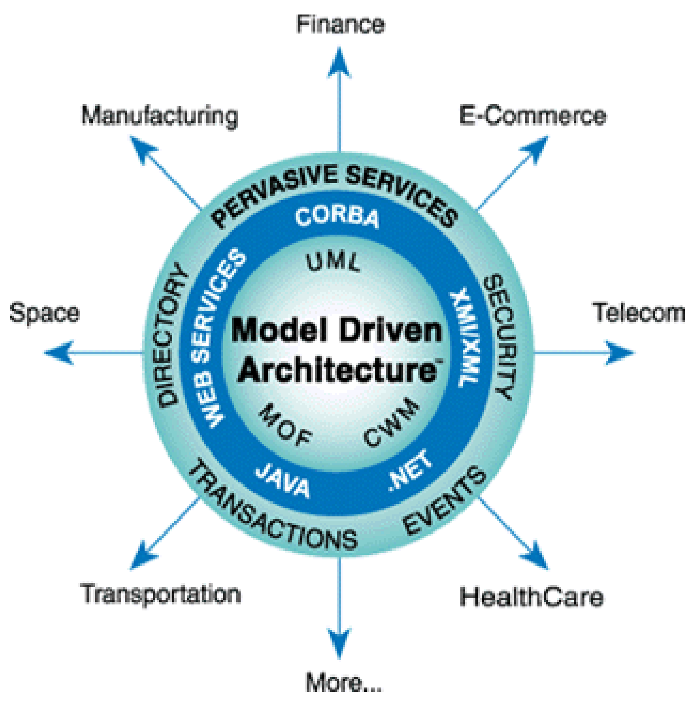


Figure 2.1: OMG's Model Driven Architecture (source: omg.org).

statements about what can be expressed in valid models. A metamodel can be considered as yet another abstraction, highlighting properties of the model itself. A model shows a simplification of a subject. A meta-model offers the vocabulary for formulating reasonings on top of a given model. While both are models, they are different in intent and interpretation [73]. A model conforms to its metamodel in the way that a computer program conforms to the grammar of the programming language in which it is written [74]. Model-driven engineering with metamodels within model-driven principles is an attempt on leveraging these concepts to bring productivity to software development.

2.2.2 Model transformation

Model transformations can be considered as components that take models as input. There is a variety of kinds of model transformation, which differ in their inputs, outputs and also in the way they are expressed. A model transformation usually specifies which models are acceptable as input, and if appropriate what models it may produce as output, by specifying the meta-model to which a model must conform. The source and target models can be expressed either in the same modeling languages or in different formats.

Model transformations and languages hence could be classified according to several attributes, some of the major classifications as seen in the work of Czarnecki et al. and Mens and Van Gorp [15][54] are:

- *Number and type of inputs and outputs.* In principle a model transformation may have many inputs and outputs of various types. The only absolute limitation is that a model transformation will take at least one model as input. However, a model transformation that did not produce any model as output would more commonly be called a model analysis or model query.

- *Endogenous versus exogenous.* In order to transform models, these models need to be expressed in some modeling language (e.g., UML for design models, a domain specific language for modeling or programming languages for source code models). The syntax and semantics of the modeling language itself is expressed by a meta-model. Endogenous transformations are transformations between models expressed in the same language. Exogenous transformations are transformations between models expressed using different languages [54]. For example, in a process conforming to the OMG Model Driven Architecture, a platform-independent model might be transformed into a platform-specific model by an exogenous model transformation.

- *Horizontal versus vertical transformations.* A horizontal transformation is a transformation where the source and target models reside at the same abstraction level. Typical examples are refactoring (an endogenous transformation) and migration (an exogenous transformation). A vertical transformation is a transformation where the source and target models reside at different abstraction levels. A typical example is refinement, where a specification is gradually refined into a full-fledged implementation [54].

- *Unidirectional versus bidirectional.* A unidirectional model transformation has only one mode of execution: that is, it always takes the same type of input and produces the same type of output. Unidirectional model transformations are useful in compilation-like situations, where any output model is read-only. The relevant notion of consistency is then very simple: the input model is consistent with the model that the transformation would produce as output, only. For a bidirectional model transformation, the same type of model can sometimes be input and other times be output. Bidirectional transformations are necessary in situations where people are working on more than one model and the models must be kept consistent. Then a change to either model might necessitate a change to the other, in order to maintain consistency between the models. Because each model can incorporate information which is not reflected in the other, there may be many models which are consistent with a given model.

- *Syntactical versus semantic transformations.* A final distinction can be made between model transformations that merely transform the syntax, and more sophisticated transformations that also take the semantics of the model into account. As an example of syntactical transformation, consider a parser that transforms the concrete syntax of a program (resp. model) in some programming (resp. modeling language) into an abstract syntax. The abstract syntax is then used as the internal representation of the program (resp. model) on which more complex semantic transformations (e.g. refactoring or optimization) can be applied. Also when we want to import our export our models in a specific format, a syntactical transformation is needed [54].

A model transformation can be implemented in a general purpose programming language or by a domain specific model transformation language. In some model transformation languages, a model transformation is itself a model, hence this model also conforms to a meta-model which is part of the model transformation language's definition.

2.3 Unified Modeling Language

Unified Modeling Language (UML) is a standardized, general-purpose modeling language in the field of software engineering used for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.

The Unified Modeling Language includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems. In theory models can be defined in any language, however according to Object Management Group's standards, models should be defined in a MOF-compliant language, UML is one of such languages. This argument is supported by the ability to facilitate and improve reuse, as well as being able to conform and exchange information with other tools from the OMG family. UML is well-known for its extensive graphical notation and diagramming techniques. However, not all complicated designs can be simplified and fully expressed by pictures. There are cases where additional information needs to be captured in a different way. For this reason the UML includes the Object Constraint Language (OCL), a textual language that allows a UML modeler to specify additional constrains and other requirements that sometimes graphical models are just not enough good for.

2.4 Object Constraint Language

In the OMG's MDA, precise modeling and behavior of action, execution, query and transformation on models are essential. OCL is one of the approaches that can be used for this purpose. We often see OCL appears in an UML diagram or in the supporting documentation describing a diagram such as business rule definitions. However, this does not mean that OCL is strictly entitled with UML. We can also use OCL on non-UML diagram for the same purpose. In terms of language category, OCL is a declarative language for describing rules that apply to a UML model. OCL was developed at IBM and now is a part of UML standard. Initially, OCL was only a formal specification language extension to UML. Now, as we mentioned, OCL can be used with any MOF OMG's meta-model, including UML. In terms of operations and functionality, OCL supports three (out of five) required relational algebra operations. Union, Difference, and Select are all supported by operations defined on the OCL collection types. However, the operation Product (or Cartesian Product) and Project are not supported, and cannot be supported as they directly require a facility for structured aggregation or a notion of tuples [52].

UML combined with OCL enables the resolution for many of the tasks that are required for MDA. OCL was originally viewed as a way to introduce constrains or to restrict certain values in a model. Nevertheless, OCL can also be used to support query expressions, derived values, conditions, business rules etc. OCL can express concepts that are not supported by UML diagrams, hence making models at all levels more precise. OCL can also support transformation tools and code generation as a key component to MDA.

2.5 Domain specific language

General purpose modeling languages like UML are like multipurpose knives, they can be used for many objectives from cutting to peeling or even drilling. However, imagine using a knife to drill a hole is clearly not effective. The basic design principle of a domain specific language (DSL) is targeted to a particular kind of problem, rather than a general purpose language that aimed at any kind of software problem. It is a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique.

In software development and domain engineering, a domain-specific language is a programming language or specification language dedicated to a particular problem in the development of a range of software systems. It can be a particular representation technique, and/or a particular solution technique to a specific software problem domain. The concept is not new, special-purpose programming languages and all kinds of modeling/specification languages have always existed, but the term has become more popular due to the rise of domain-specific modeling. Using domain specific language can be a solution to several problems encountered in various software development aspects. A DSL can reduce the cost implied in development and maintenance of software systems [18].

Domain-specific languages have (with declared syntax or grammars) very specific goals in design and implementation. A domain-specific language can be either a visual diagramming languages, such as those created by the Generic Eclipse Modeling System, programmatic abstractions, such as the Eclipse Modeling Framework, or textual languages. For instance, the command line utility *grep* in Unix systems has a regular expression syntax, which matches patterns in lines of text. The *sed* utility defines a syntax for matching and replacing regular expressions. Often, these tiny languages can be used together inside a shell to perform more complex programming tasks [75].

Recent movement within the MDA initiatives is introducing a concept of domain specific modeling that utilizes customized modeling languages to raise the level of abstraction but at the same time narrow downs the design space. This allow developers to work directly with the domain concepts of the problem space and provide automation support for software development.

2.6 Role of domain specific language in MDA

Domain specific language in MDA contains languages for creating models. The process of modeling software systems in a specific domain is often referred to as Domain Specific Modeling (DSM) and the languages used in for this purpose are often referred to as Domain Specific Modeling Languages (DSML).

The DSM philosophy uses the concept of narrowly defined modeling languages is contrasted with larger standardized modeling languages, such as UML. These general languages provide abstractions that may not be needed in every domain, this could add redundancy and confusion to domain experts. The principle of DSML utilizing notations that relate directly to a familiar domain not only helps flatten learning curves but also facilitates the communication between a broader range of experts, such as domain experts, system engineers and software architects. In addition, the ability of DSM to synthesize artifacts from high-level models to low-level implementation artifacts, simplifies the activities in software development such as developing, testing and debugging [49].

The key challenge in applying DSM is to define useful standards that enable tools and models to work together portably and effectively [21]. Existing de facto standards include the Object Management Group's Model Driven Architecture [32], Query/View/Transformations (QVT) [29] and the Meta Object Facilities (MOF) [33]. These standards can also form the basis for domain-specific modeling tools. There are several attempts in building the infrastructures and tools for DSM. Some of which are the Generic Modeling Environment (GME), ATLAS Model Management Architecture, Eclipse Modeling Framework (EMF) [49]. In history, initial success stories from industry adoption of DSM have been reported, one of which is the story about Saturn's multi-million dollar cost savings associated with timelier reconfiguration of an automotive assembly line driven by seven domain-specific models [50]. There are also various projects from companies in industry, such as BMW, formerly Nokia (now belongs to Microsoft), Honeywell that have successfully adopted DSM within their software development processes.

Chapter 3

Previous work and case studies

3.1 Overview of model transformation approaches

Model transformation overview is useful for users in the domain of interest, it can help a developer make a decision of choosing the appropriate approach that is best suited for his requirements. Having a broad perspective on current state-of-the-art also helps us identify the advantages and weaknesses of different methods when building the domain specific language for model driven development of web services in the following chapters. Based on the study of current model transformation technologies, the following sections depict the classification and overview on the previous work and methods in model driven development approaches.

3.1.1 Graph transformation

In many cases, (meta-) models are represented in UML formalism. As a result, the models can be viewed as graphs. It is therefore natural to consider the use of graph grammars to express model transformation. Graph transformation [45] approaches are very well founded theoretically. They favor matching and replacement. They are based on syntactic graph rules that consist of finding a Left Hand Side graph and replacing it by a Right Hand Side graph. This approach has the power of a clear operational idea, which enhances rule specification. The complexity of this approach stems from its non-determinism in scheduling and application strategy, which requires careful consideration of termination of the transformation process and the sequence of rule application [17].

Early work involving graph transformation and models largely centered on their use in defining the semantics of different modeling diagram types, such as the continuing work of Kuske et al.[45]. More recent work by Kuster et al. [46] has defined a more general model transformation approach using graph transformation as the underlying mechanism, allowing them to draw upon some of the properties of graph transformations in a model transformation context. Heckel et al. [35] have continued this work, reasoning about confluence with typed attributed graphs. [48] have proposed model transformation approaches which are essentially based upon simplified views of graph transformations, as is Agrawal et. al's more mature GReAT system [1].

Although graph transformations have several interesting properties when applying to model transformations, it is still not used widely in practical situation due to the complexity and lack of structuring mechanisms. Solutions based on the graph transformation paradigm therefore have limited real-world usage [72]. Nevertheless, graph transformations can be considered as foundation theory that can be applied when implementing other types of transformation.

3.1.2 API approach

This type of transformation is based firstly on Meta Object Facility specification. MOF is used in many modeling tools to create model repositories. After that Application Programming Interfaces (API) are generated for each supported meta-model. These interfaces are used to describe the model transformation process by means of programs written in an imperative language: Java, C++, etc. This approach provides the user with a set of interfaces used to describe the transformation process as a series of instructions that allow the generation of a target model from a corresponding source model. The use of APIs to describe a transformation process is a powerful solution because programming languages generally have good performance at runtime. Basically, the entire procedure must be performed by the user.

This user is in charge of the organization and description of all stages, explicitly in terms of imperative statements [17].

3.1.3 XSLT approach

Along with XML technology, XML Metadata Interchange (XMI) enables the exchange of meta-models as a standard. There is a need for bridging between XML processing and other form of data and a language for that purpose is in demand. XSL stands for EXtensible Stylesheet Language and XSLT stands for XSL Transformations. As models are described in XML format, it appears that EXtensible Stylesheet Language Transformation (XSLT) is a convenient solution for model transformation. XSLT is an appropriate standard for XML document transformation, but suffers from limitations in realizing model transformation. Moreover, XSLT data types are limited; this restricts the scope of information that must be computed during the transformation process. In a DTD, the syntax and the semantics of an XML documents are fixed, and transformation rules therefore have to deal with both [17].

The main weakness of XSLT lies in the fact that it was adequate for the simple transformations but has serious shortcomings for more advanced transformations. Recently, a formal proof was constructed that XSLT is Turing complete. However it took several years before that was proven and in practical usages the limits in XSLT make it harder to conveniently apply this approach. A final issue which makes expressing model transformations in XSLT less than ideal is that XML documents are represented as a tree structure; models are, in the general case, naturally representable as graphs. Although graphs can be represented by trees with link references between nodes, the difference in representation can lead to an unnatural representation of many types of model transformations [72].

3.1.4 Declarative approach

In declarative approach, the relationship between concepts in the source and the target meta-model is defined by patterns. The transformation is defined by a set of rules. A rule lays forth a pattern of source model concepts, which is then transformed into a set of elements in the target model. The sequence of the various stages of the transformation process is controlled by the user, thanks to operators that allow the carrying out of explicit transformation rules invocation. The implementation is realized by an inference engine [17]. However, one of the disadvantages of this approach is the significant amount of work burden the developer with specifying all the constraints supporting the transformation.

3.1.5 Imperative approach

Imperative approach as similar to imperative programming works in the paradigm that describes the transformation in terms of statements that change the program states. Imperative approach defines a sequence of commands to perform. An example of this approach is Transformation Rule Language (TRL). This language [29] is in essence a standard rule-based imperative language specialized for UML-esque model transformations. This comes in several forms: concepts such as 'transformation rule' are raised to first-class status; some of the information recorded in the new first class elements is used for additional purposes e.g. to create tracing information; extra syntax is provided for e.g. accessing the stereotype of a UML model element. Rules consist of a crude signature (comprising the types of the source and target model elements) and an imperative body. The syntax and semantics of actions are essentially that of the Object Constraint Language (OCL) [28] augmented with side-effects and a small handful of necessary control structures. The benefit of such an approach is its relative familiarity to users, and the knowledge that largely imperative solutions traditionally lead to efficient implementations. However TRL is only capable of expressing unidirectional stateless transformations, due to the imperative nature of rule actions [72].

3.1.6 The hybrid approach

In a declarative approach, a transformation is defined by a set of relations between the concepts of the source and target models, as described in their meta-models. The implementation is realized by an inference engine, which allows the application of the transformation to generate the target model. In an imperative approach, a transformation is described by a set of algorithms as functions or procedures that explicitly describe the sequence of transformation applications. Hybrid approaches combine the declarative and imperative approaches as used in [60]. The declarative approach is generally used in the definition and selection of the transformations which can be applied, while the imperative approach is well adapted to describing the transformation strategy by a control flow of execution rules, and hence to executing the transformation [17]. In hybrid languages, transformation rules mix the syntax and the semantics of the concepts they handle. ATLAS Transformation Language (ATL) [60] as an example implements imperative bodies inside declarative shell to specify transformation.

3.2 Domain specific modeling language approaches

In order to implement a domain specific modeling language, some essential design principles can be applied. There have been research on how to effectively design a software system in general that could be useful also for domain specific languages. Peffers et al. in [63] proposed a methodology where several elements of design processes are identified. This process includes six steps: problem identification and motivation, definition of the objectives for a solution, design and development, demonstration, evaluation, and communication. Several useful case studies were demonstrated in various application domains. In an attempt to produce a more concrete guideline with specific steps, Hevner et al. [38] presented several important guidelines in design science:

1. Producing a viable artifact
2. Tackling a relevant problem
3. Evaluating the proposed design solution
4. Contributing clear and verifiable results to research

These items are important for our study to evaluate several approaches in domain specific modeling and understand the challenge of creating a high quality modeling framework.

Inline with the principle of producing of viable artifacts, DSL approaches have been used not only in software but also many other industries such as industrial machine control, embedded systems, mobile application specifications, and web application development. MetaEdit+ is one of the tool that focusses on meta-modeling and code generation [40]. By raising the level of abstraction using meta models, it aims to increase productivity and maintain good level of code reuse. Meta-modeling has a similar expressiveness to the grammar-based approach, this increases the compatibility of different systems. One drawback from this approach is that complex operations resulting from rule compositions cannot be easily realized by using this tool. Luoma et al. [51] presented suggestions for developing a family of applications with domain specific modeling tools, based on experiences with MetaEdit+ and applying DSMLs in different domains. Lessons learned from this work can be used for the development of a modeling tool with support for DSML and code generation as well as building a domain specific framework.

DSMLs can be used to solve problems in the software development process. Software architecture modeling is one of the aspects that was attempted to tackle in the work of Kaul et al. [41]. In this approach, Patterns-oriented Software Architecture Modeling Language (POSAML) is developed as a visual modeling language for middleware provisioning. Several non-functional or systemic concerns must be addressed when provisioning (configuring, optimizing, and validating) distributed applications on their middleware platforms. Practitioners generally manage this problem with the manual configuration of XML files, which can be very time-consuming, straining, and error-prone work. As a solution, the POSAML is provided, which raises the level of abstraction for provisioning middleware. As a visual aid, the POSAML introduces the "middleware building block" which is implemented with software patterns, and offers a technology-independent solution to middleware provisioning [43].

In electronic and computer hardware industry, ProcGraph [39] is developed as a DSML in the domain of programmable logic controllers (PLCs). ProcGraph consists of three visual tools, each describing a particular aspect of software to be produced. The procedural control entities diagram shows the system's composition as well as the relationships between conceptual components. The state transition diagram describes the behavior of conceptual components. The entity dependency diagram shows different dependencies between the conceptual components also it represents a mixture of the first two diagrams. Automatic code generation is used to transform a model from a higher level of abstraction to produce a PLC programming code, which is later imported to interpret the produced code and generate the executable code for PLCs. Data acquisition is a demanding process that includes highly specialized equipment [43].

There has been work attempting to study and analyze the advantages and disadvantages in the development of domain-specific modeling languages. Sprinkle et al. [69] presented an overview and describes several disadvantages with over-engineering of DSMLs and possible coupling of models. To reduce the risk of developing over-engineered DSMLs, there is a need to focus on the extensibility and changeability of grammar rules and vocabularies. This enables resulting models and the language definition to be evolved interactively starting from a small core language.

Amyot et al. [4] presented an evaluation of development tools for domain specific language. Several development tools for DSMLs are analyzed such as the Generic Modeling Environment (GME), Rational Software Architect, XMF-Mosaic and Eclipse Modeling Framework (EMF). Among them, the support for model evolution can be seen in GME and EMF. Another criterion is integration with other languages. The tools that fulfill this criterion achieve integration either by building on UML or by technological integration via the Eclipse platform. It remains unclear how gaps between similar but slightly different concepts and structures are bridged in these solutions.

Several attempts on study of the evolution and migration ability of domain specific modeling language have also been carried out. Herrmannsdoerfer et al. in [37] examined the GMF framework and presented a detailed study regarding the evolution of languages and models. It identifies the need for language evolution by operators. This can be supported within the formal techniques of algebraic graph transformation by providing corresponding rules for the different operators that are proposed, whereas an equivalent solution in the GMF framework requires considerable effort.

Sprinkle and Karsai proposed a visual language for specifying domain model evolutions in [70]. While this language is also inspired by graph transformations, it aims to create domain-specific visual languages that are defined by meta-modeling. To define transformation, this approach utilize a fundamental set of operators [11]. Another approach utilizing meta model concepts can be seen in [6]. This approach presented a concept for automatically deriving transformations of statements from evolutions of domain-specific languages.

Empirical research in software engineering is a challenging discipline. The main disadvantage of DSLs is the cost of their development, requiring both domain and language development expertise. This is one of the reasons why DSLs are still not widely adopted in many emerging domains [44]. There is also the lack of guidelines and experienced reports on DSL development. As more and more systems are moving into cloud-based architecture recently, there is a shortage of guidelines for helping DSL developers in the web domain. Chapter 4 and 5 present our approaches to this area and introduces the suitable implementations for working with web services and web application testing.

3.3 Case studies

In this section, we look at several case studies as background information to model driven development process. This is aimed to provide useful information in building the domain specific language for model driven development in the following chapters. There exist various techniques to define and perform model transformations. Some of these techniques provide transformation languages to define transformation rules and their application, which can be either graphical or textual, either imperative or declarative.

We first looked at a formal approach using graph theory and triple graph grammars to specify model transformation. This provides a higher level of structuring and specification, allows us to define the complete behavior of a transformation and its process in order for effective verification and reuse of transformations. We apply this formalism in the construction of transformation rules from the domain specific language representing state machine models to Java programing language in Chapter 5.

3.3.1 Formal specification of model transformation

Specification of models and model transformations play an essential role in model-driven software development. It is important to have higher level of structuring and specification, to define the complete behavior of a transformation and its process in order for effective verification and reuse of transformations. In this sub-section, we describe a formal approach to specify model transformation using graph grammar and the ability to support bidirectional transformation with triple graph grammar.

Graphs and graph transformations are used in a variety of use cases for specifying, analyzing and optimizing systems. From the work of Ehrig et al. [23], using graphs to specify model transformations is proven to work as an adequate basis to specify model transformations between different domain-specific modeling languages. If effectively constructed, this approach can support the specification for visual, formal and bidirectional transformation.

For exogenous model transformations, triple graph grammars has proven to be a well-suited formalism [22] and they were successfully applied in several domains. Let us define the concepts needed for the construction of model transformations:

Definition 1. Graph: *A graph $G = (G_V, G_E, s_G, t_G)$ consists of a set G_V of nodes, a set G_E of edges, and two functions s_G, $t_G : G_E \to G_V$, which are the source and the target function that define edges by connecting nodes.*

Definition 2. Morphism: *Given graphs G, H a graph morphism $f : G \to H$, where $f = (f_V, f_E)$ consists of two functions $f_V : G_V \to H_V$ and $f_E : G_E \to H_E$ that preserve the source and the target function, i.e. $f_V \circ s_G = s_H \circ f_E$ and $f_V \circ t_G = t_H \circ f_E$.*

A graph morphism f is injective if both functions f_V, f_E are injective [23]. Graphs and graph morphisms define the category Graph. In the modeling world, these definitions of graphs can be used to represent various artifacts such as class diagrams. Graphs and diagrams are very useful means to describe complex structures and systems and to model concepts and ideas in a direct and intuitive way. Let us now define the *typing* concept in order to describe the structure of such artifacts.

Definition 3. Typed graph: *Given a distinguished graph TG, called type graph, a typed graph G = (G, $type_G$) consists of a graph G = (V, E, s, t) together with a type morphism $type_G : G \rightarrow TG$ from G to its type graph TG. The distinguished graph TG is a type graph. A tuple (G, $type_G$) of a graph G together with a graph morphism $type_G : G \rightarrow TG$ is called a typed graph.*

Given typed graphs G = (G, $type_G$) and H = (H, $type_H$), a typed graph morphism f is a graph morphism $f : G \rightarrow H$, such that $type_H \circ f = type_G$.

This concept plays a part in the creation of meta-models, in which their models are conformed to. This concepts can also be used in object-oriented modeling graphs to describe two levels: the type level - given by a class diagram and the instance level - given by all valid object diagrams. This idea can be described more generally by the concept of typed graphs, where a fixed type graph TG serves as abstract representation of the class diagram [5]. To describe the conditions which a graph needs to fulfill in order to belong to a type, we need to define graph grammar. A graph grammar specifies a graph language as set of all those graphs that can be created by applying transformation rules starting with a given start graph.

Definition 4. Typed Graph Rule: *A typed graph rule $p = L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ consists of three typed graphs L: the left-hand side graph, K: the gluing graph, R: the right-hand side graph, and two injective typed graph morphisms l and r.*

Definition 5. Typed Graph Grammar*: A typed graph grammar GG = (TG, S, R) consists of a type graph TG , a start graph S and set of graph rules R. If a rule p is applicable to a graph G via a morphism $m : L \rightarrow G$, called match, the transformation $G \overset{p}{\Rightarrow} H$ is defined by two pushouts or double-pushout (DPO):*

$$\begin{tikzcd}
L \arrow[d, "m"'] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"'] & R \arrow[d, "m*"] \\
G & D \arrow[l, "l*"] \arrow[r, "r*"'] & H
\end{tikzcd}$$

The typed graph language L of GG is defined by L = {G | ∃ typed graph transformation S ⇒*G}.

As seen in Fig. 3.1, to introduce associations in a class diagram, the rule for inserting association between two existing classes is expressed as:

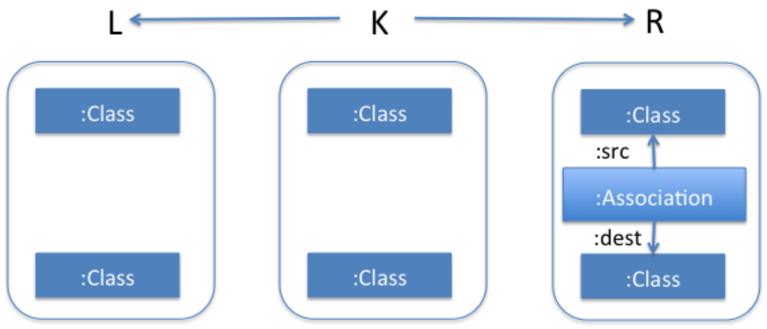


Figure 3.1: Graph rule to insert association between two classes

Informally, assume a given graph G, a graph rule p: $L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ and a graph morphism *m: $L \rightarrow G$,* the double pushout is carried out in two phases:

1. *Pushout complement*: The context graph is obtained "deleting" from G all elements images of elements in L but not of elements in K.

2. *Pushout fill*: The final graph is obtained "adding" to context graph all elements which don't have a pre-image in K.

For the association insertion rule as in Fig. 3.1, when applying the rule to a source class diagram having a match, this results in two classes having the new association between them inserted.

In order to define a transformation in a declarative way and still execute the transformations in both directions, Triple Graph Grammars (TGGs) can be introduced as a technique for model transformation. Let us define the concepts:

Definition 6. Triple Graph and Triple Graph Morphism: *Three graphs SG, CG, and TG, called source, connection, and target graph, together with two graph morphisms s_G*

: $CG \rightarrow SG$ and $t_G : CG \rightarrow TG$ form a triple graph $G = (SG \overset{s_G}{\leftarrow} CG \overset{t_G}{\rightarrow} TG)$. If SG, CG, and TG are empty graphs, we say that G is empty.

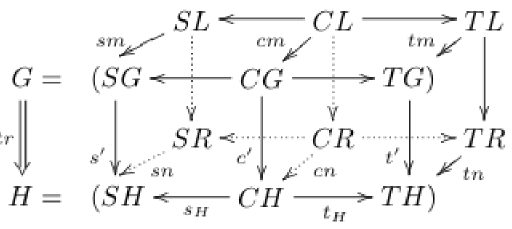
Triple Graph Morphism: *A triple graph morphism $m = (s, c, t) : G \rightarrow H$ between two triple graphs $G = (SG \overset{s_G}{\leftarrow} CG \overset{t_G}{\rightarrow} TG)$ and $H = (SH \overset{s_H}{\leftarrow} CH \overset{t_H}{\rightarrow} TH)$ consists of three graph morphisms $s : SG \rightarrow SH$, $c : CG \rightarrow CH$ and $t : TG \rightarrow TH$ such that $s \circ s_G = s_H \circ c$ and $t \circ t_G = t_H \circ c$. Triple graph morphism m is injective, if morphisms s, c and t are injective.*

Triple graphs and triple graph morphisms form the category TripleGraphs. From the definitions of triple graph and triple graph morphism, we can now define the grammar that is made up from triple graphs:

Definition 7. Triple Graph Grammar: *A triple rule $tr = L \overset{tr}{\rightarrow} R$ consists of triple graphs L and R and an injective triple graph morphisms tr. A triple graph grammar TGG = (TG, S, TR) consists of a triple type graph TG, a triple start graph S and triple rules TR typed over TG.*

$$\begin{tikzcd}
L & = & (SL & CL \arrow[l, "s_L"'] \arrow[r, "t_L"] & TL) \\
R \arrow[from=u, "tr"'] & = & (SR \arrow[from=u, "s"'] & CR \arrow[from=u, "c"'] \arrow[l, "s_R"] \arrow[r, "t_R"'] & TR) \arrow[from=u, "t"']
\end{tikzcd}$$

From triple rule tr, we can also define triple graph transformation step (TGTstep): *Given a triple rule $tr = (s, c, t) : L \rightarrow R$, a triple graph G and a triple graph morphism $m = (sm, cm, tm) : L \rightarrow G$, called triple match m, a triple graph transformation step (TGTstep) $G \overset{tr,m}{\Rightarrow} H$ from G to a triple graph H is given by three objects SH, CH and TH in category Graph with induced morphisms $s_H : CH \rightarrow SH$ and $t_H : CH \rightarrow TH$. Morphism n = (sn, cn, tn) is called comatch.*



$$\begin{tikzcd}
 & & SL \arrow[dl, "sm"'] \arrow[dd, dotted] & & CL \arrow[ll] \arrow[rr] \arrow[dl, "cm"'] \arrow[dd, dotted] & & TL \arrow[dl, "tm"'] \arrow[dd] \\
G = \arrow[dd, Rightarrow, "tr"'] & (SG \arrow[dd, "s'"'] & & CG \arrow[ll] \arrow[rr] \arrow[dd, "c'"'] & & TG) \arrow[dd, "t'"'] & \\
 & & SR \arrow[dl, dotted, "sn"] & & CR \arrow[ll, dotted] \arrow[rr, dotted] \arrow[dl, dotted, "cn"] & & TR \arrow[dl, "tn"] \\
H = & (SH & & CH \arrow[ll, "s_H"] \arrow[rr, "t_H"'] & & TH) &
\end{tikzcd}$$

As the result, we obtain a triple graph morphism d : G → H with d = (s´,c´,t´) called transformation morphism. A sequence of triple graph transformation steps is called triple transformation sequence (TGT-sequence) [22]. A triple graph grammar TGG = (S, TR) consists of a triple start graph S and a set TR of triple rules. The triple graph language L of triple graph grammar is defined by:

$$L = \{G \mid \exists \textit{ triple graph transformation } S \Rightarrow^* G\}.$$

Triple graph grammar can be used to define the relation between two types of models as well as to transform a model of one type into another, to compute the correspondence between two existing models, or to maintain the consistency between the two types of models as defined by the TGGs. When one of the models is changed, the other one can be change accordingly, which means that the transformations or synchronizations can be applied incrementally [42].

Definition 8. Direct triple transformation: *Given a triple rule tr =* $L \overset{tr}{\rightarrow} R$ *consists of triple graphs L and R. A direct triple transformation* G $\overset{tr,m}{\Rightarrow}$ H *of a triple graph G via a triple rule tr and a match m : L → G is given by the pushout:*

$$\begin{tikzcd} L \arrow[r, "tr"] \arrow[d, "m"'] & R \arrow[d, "n"] \\ G \arrow[r, "f"'] & H \end{tikzcd}$$

From a triple rule, we can derive a source rule tr_S and a target rule tr_T, which specify the changes done by this rule in the source and target components, respectively. Additionally, we can derive the forward rule tr_F which describes the changes done by the rule to the connection and target. Similarly, the backward rule tr_B describes the changes done by the rule to the connection and source parts. In this process, the source rule creates a source model, which can then be transformed by the forward rules into the corresponding target model. This means that the forward rules define the actual model transformation from source to target models. In the backward direction, the target rules create the target model, which can then be transformed into a source model applying the backward rules. Thus, the backward rules define the backward model transformation from target to source models [27]. This way, we can specify the transformation in both direction when needed. Let us specify the derived rules in more detail:

Definition 9. Derived Triple Rules: *From each triple rule $tr = L \to R$ we have the following source, forward, target and backward rules:*

$$\begin{tikzcd}
SL \arrow[d, "s"'] & \emptyset \arrow[l] \arrow[r] \arrow[d] & \emptyset \arrow[d] \\
SR & \emptyset \arrow[l] \arrow[r] & \emptyset
\end{tikzcd}$$

source rule tr_S

$$\begin{tikzcd}
\emptyset \arrow[d] & \emptyset \arrow[l] \arrow[r] \arrow[d] & \text{TL} \arrow[d, "t"] \\
\emptyset & \emptyset \arrow[l] \arrow[r] & \text{TR}
\end{tikzcd}$$

target rule tr_T

$$\begin{tikzcd}
SL \arrow[d, "id"'] & CL \arrow[l, "s \circ s_L"'] \arrow[r, "t_L"] \arrow[d, "c"'] & \text{TL} \arrow[d, "t"] \\
SR & CR \arrow[l, "s_R"] \arrow[r, "t_R"'] & \text{TR}
\end{tikzcd}$$

forward rule tr_F

$$\begin{tikzcd}
SL \arrow[d, "s"'] & CL \arrow[l, "s_L"'] \arrow[r, "t \circ t_L"] \arrow[d, "c"'] & \text{TL} \arrow[d, "id"] \\
SR & CR \arrow[l, "s_R"] \arrow[r, "t_R"'] & \text{TR}
\end{tikzcd}$$

backward rule tr_B

From the the triple rules TR, we can define the language VL = {G|$\emptyset \Rightarrow^*$G via TR} of triple graphs. Given a triple graph $G = (SG \overset{s_G}{\leftarrow} CG \overset{t_G}{\rightarrow} TG)$, we define the projection proj$_T$(G) to the target is triple graph G$_T$ = $(\emptyset \overset{\emptyset}{\leftarrow} \emptyset \overset{\emptyset}{\rightarrow} TG)$ and the projection proj$_S$(G) to the source is triple graph G$_S$ = $(SG \overset{\emptyset}{\leftarrow} \emptyset \overset{\emptyset}{\rightarrow} \emptyset)$. This shows that each TGT-sequence can be decomposed in transformation sequences by corresponding source and forward rules and vice versa, which provides the support for incremental changes [23]. Assume proj$_X$ is a projection defined by restriction to one of the triple components where X∈{S, C, T}, we can derive the source and target languages. Source language VL$_S$ is derived by projection to the triple components: VL$_S$ = proj$_S$(VL). Target language VL$_T$ is derived by VL$_T$ = proj$_T$(VL).

Model transformations from elements of the source language VL$_{S0}$ to elements of the target language VL$_{T0}$ can be defined on the basis of forward rules. In the opposite direction, they can be defined using backward rules hence make it possible to define backward transformations from target to source graphs and altogether form the bidirectional model transformations. To specify the consistency of source elements, let us define the following:

Definition 10. Match and Source Consistency: *Let tr_S^* and tr_F^* be sequences of source rules tri_S and forward rules tri_F, which are derived from the same triple rules tri for i = 1,...,n. Let further $G_{00} \overset{tr_S^*}{\Rightarrow} G_{n0} \overset{tr_F^*}{\Rightarrow} G_{nn}$ be a TGT-sequence with (mi_S,ni_S) being match and comatch of tri_S (respectively (mi,ni) for tri_F) then match consistency of $G_{00} \overset{tr_S^*}{\Rightarrow} G_{n0} \overset{tr_F^*}{\Rightarrow} G_{nn}$ means that the S-component of the match mi is uniquely determined by the comatch ni_S (i = 1,...,n).*

A TGT-sequence $G_{n0} \overset{tr_F^*}{\Rightarrow} G_{nn}$ is source consistent, if there is a match consistent sequence $\emptyset \overset{tr_S^*}{\Rightarrow} G_{n0} \overset{tr_F^*}{\Rightarrow} G_{nn}$.

Note that by source consistency the application of the forward rules is controlled by the source sequence, which generates the given source model.

Theorem 1: Canonical Decomposition and Composition:

1. Decomposition: *For each TGT-sequence* $\mathrm{G}_0 \overset{tr^*}{\Rightarrow} \mathrm{G}_n$ *there is a canonical match consistent TGT-sequence* $G_0 = \mathrm{G}_{00} \overset{tr_S^*}{\Rightarrow} \mathrm{G}_{n0} \overset{tr_F^*}{\Rightarrow} \mathrm{G}_{nn} = G_n$ *using corresponding source rules tr_S^* and forward rules tr_F^*.*

2. Composition: *For each match consistent transformation sequence* $G_0 = \mathrm{G}_{00} \overset{tr_S^*}{\Rightarrow} \mathrm{G}_{n0} \overset{tr_F^*}{\Rightarrow} \mathrm{G}_{nn} = G_n$ *using corresponding source rules* tr_S^* *and forward rules* tr_F^*, *there is a canonical transformation sequence* $\mathrm{G}_0 \overset{tr^*}{\Rightarrow} \mathrm{G}_n$.

3. Bijective Correspondence: *Composition and Decomposition are inverse to each other.*

The proof of this theorem is presented in the work of Ehrig et al. [22]. This allows us to infer the bijective correspondence between decomposition and composition. The main significance it brings is the equivalence of forward and backward TGT-sequences which can be derived from the same general TGT- sequence.

Definition 11. Model Transformation using forward rule: *A model transformation sequence* $(G_S, G_1 \overset{tr_F^*}{\Rightarrow} G_n, G_T)$ *consists of a source graph* G_S, *a target graph* G_T, *and a source consistent forward TGT-sequence* $G_1 \overset{tr_F^*}{\Rightarrow} G_n$ *with* $G_S = proj_S(G_1)$ *and* $G_T = proj_T(G_n)$.

A model transformation $MT : VL_{S0} \Rrightarrow VL_{T0}$ *is defined by model transformation sequences* $(G_S, G_1 \overset{tr_F^*}{\Rightarrow} G_n, G_T)$ *with* $G_S \in VL_{S0}$ *and* $G_T \in VL_{T0}$.

Using the same pattern, backward transformation based on backward rule can be defined analogically. To verify the transformation, we need to make sure that the transformation is justified. Let us define the correctness and completeness of the transformation:

Definition 12. Syntactical Correctness and Completeness: *Given a model transformation* $MT : \mathrm{VL}_S \Rrightarrow \mathrm{VL}_T$. *We say that:*

- MT is syntactically correct, *if for each model transformation sequence* $(\mathrm{G}_S, \mathrm{G}_1 \overset{tr_F^*}{\Rightarrow} \mathrm{G}_n, \mathrm{G}_T)$ *there is* $G = (\mathrm{G}_S \leftarrow \mathrm{G}_C \rightarrow \mathrm{G}_T) \in VL$.

- MT is complete, *if for each source model* $G_S \in VL_S$ *there is a model transformation sequence* $(\mathrm{G}_S, \mathrm{G}_1 \overset{tr_F^*}{\Rightarrow} \mathrm{G}_n, \mathrm{G}_T)$

Theorem 2. Correctness of results: *A model transformation* $MT : \mathrm{VL}_S \Rrightarrow \mathrm{VL}_T$ *based on forward rules is syntactically correct and complete.*

Proof:

1. *Syntactical Correctness*: From Definition 11 of model transformation using forward rule, with a model transformation sequence $(G_S, G_1 \overset{tr_F^*}{\Rightarrow} G_n, G_T)$ we have source consistent forward TGT-sequence $G_1 \overset{tr_F^*}{\Rightarrow} G_n$, this means there is a match consistent sequent $\emptyset \overset{tr_S^*}{\Rightarrow} G_1 \overset{tr_F^*}{\Rightarrow} G_n$. Applying theorem 1 with composition we imply that $\emptyset \overset{tr^*}{\Rightarrow} G_n$, this means for $G = G_n$ there is $G \in VL$ with $G = (G_S \leftarrow G_C \rightarrow G_T)$ hence $G_S \in VL_S$, $G_T \in VL_T$.

2. *Completeness*: Since we have a source model $G_S \in VL_S$, there exits $G = (G_S \leftarrow G_C \rightarrow G_T)$ where $G \in VL$. This implies that there exists a TGT-sequence $\emptyset \overset{tr^*}{\Rightarrow} G$. Applying the first part of theorem 1 with decomposition we have a match consistent sequence $\emptyset \overset{tr_S^*}{\Rightarrow} G_1 \overset{tr_F^*}{\Rightarrow} G_n$. This hence defines a model transformation sequence $(G_S, G_1 \overset{tr_F^*}{\Rightarrow} G_n, G_T)$ using $G = (G_S \leftarrow G_C \rightarrow G_T)$.

Taking both 1. and 2. into account, we have completed the proof of Theorem 3. This guaranteed the correct and complete results of TGG-model transformation. This approach hence allows us to formally define higher level of structuring and specification of model transformation as well as specify the complete behavior of a transformation in both directions using triple rules.

Summary

In this sub-section, the basic concepts of graph and triple graph grammars are formalized in a theoretical way. Triple graph grammars have been applied and implemented as a formal basis for model transformations. This allows us to specify models and model transformations effectively that provide the support for bidirectional transformations, the ability in automatic derivation of forward, backward and several other transformations out of just one specified set of rules for the integrated model defined by a triple of graphs.

The advantage of triple graph grammars comes from the fact that the relation between the two models cannot only be defined, but the definition can be made operational so that one model can be transformed into the other in either direction. This allows incremental change propagations between two models. This change propagation is also bidirectional and is especially useful for iterative software engineering processes where a model evolves continuously.

In the situation where a model transformation is required to be reversible to translate information back to source models, the formalism using triple graph grammars is a promising approach. For example, a transformation of a domain-specific model to some formal model for the purpose of validation should be reversible to transform back analysis results stemming from the formal model [22].

This formalism can be seen on a number of work in various tracks such as adding nested application conditions [27], specifying equivalent model transformation based on plain rules from one with forward rules [23], on the fly construction of model transformation [24]. This serves as the background theory for our further work. Our contribution to this domain is to use graph transformations to manipulate models and use generators to produce source code. We apply triple graph grammar to the transformation of finite state machine in the domain specific language for testing of web applications. Our approach utilizes this theory and constructs the transformation between the state machine modeling language and Java codes. We describe this technique in section 5, Chapter 5.

3.3.2 Agile Development of PIMs in Model Driven Architecture

As the second case study, we looked at model driven development process from the agile perspective. A good understanding of the agile modeling process can help in the design decision when constructing the domain specific language for model driven development of web services in the next chapters. In this process, automation is achieved by applying model transformations to Platform Independent Models (PIMs). This sub-section looked at the development process from software engineering and project management perspective. We introduce an approach to developing PIMs by using UML, OCL and promotes agility by applying agile modeling for this purpose. A practical example will be demonstrated.

In the modeling process, efforts are being put into filling the gaps between requirements analysis and actual implementation. This gap is widely considered as the main source leading to poor quality applications that are hard to maintain and reuse in the future. Specialized methods from Model-driven Engineering perspective have been introduced and widely applied, making the situation better and increasing efficiency of the whole process. In this section, we point out the advantages that are brought from applying formal engineering principles in practice. Particularly we describe an approach of developing a PIMs by using UML and OCL while promote agility by applying agile modeling principles. We aim to use a combination of UML and OCL to build PIMs. These PIMs need to be enough precise, consistent, full of information, and enable a more complete generation of PSMs. In our approach we utilize plain UML for the class, use case, and state chart diagrams. However, any modeling language can be used at this stage, such as domain specific modeling languages designed for a particular application domain. OCL is used to formalize integrity constraints and other system related conditions. OCL for querying models at instance-level is used to verify models in advance, therefore brings more agility to the development process.

Our case study

We combine MDA with Agile Modeling (AM) in this approach. Agile Modeling defines a collection of core and supplementary principles that when applied on a software development project set the stage for a collection of modeling practices [13]. The most important principles include [65]:

- Assume Simplicity.
- Embrace Change.
- Enabling the Next Effort is Your Secondary Goal.
- Maximize Stakeholder Investment.
- Incremental Change.
- Model With a Purpose.

- Multiple Models.
- Rapid Feedback.

MDA and AM principles are used in the process of creation of PIMs. An important concept to understand about AM is that it is not a complete software process. AM's focus is on effective modeling and documentation. In general, development of PIMs can be a long process, in order to be agile, it need to be developed in incremental, rapid cycles. This results in small incremental deliverables with each deliverable building on previous functionality. Each deliverable is thoroughly tested to ensure model quality is maintained. The first step in this process is to capture domain information, in which we should avoid having too complex patterns too soon or over-architect the system. One important principle in this process is "Enabling the Next Effort is Your Secondary Goal". To enable it, we need to make sure that not only the outcome models is of good quality, but also create enough documentation so that the people work on the next effort can be effective.

Our experiment case is a web application as an information system. It is a simple system of a conference management (conference information system) where authors can submit their papers and edit their registration information. The conference system can also manage multiple conferences. With agile methodology, we start from a simple model (assume simplicity) and make the model reusable. For this purpose we need to create a generic PIM which can be adapted to the specific software system when needed. The following components need to be developed:

- Use cases
- State model (state diagrams)
- Class diagram : Entity and relationship (their attributes and integrity constraints (with OCL))
- System behavior model (operations)

One of the important principles is to introduce the definition a "black box" in the system behavior before proceeding to a logical design. This decision has a clear impact during analysis since operations required to specify the system response to the external events are not assigned to classes but recorded in an artificial type. In this way, responsibilities are not assigned to objects during analysis [65]. This utilizes the agile philosophy "embrace changes". In the development process, towards a reusable model for different project, we have to include different kind of stakeholders and contractors. Depending on particular project, stakeholders can have different background knowledge and cultures. To support consistency between all stakeholders, principles of agile modeling should be applied. This includes "assume simplicity, embrace change, incremental change" and "maximizing stake holder investment".

1) Use cases

The use case model consists of the definition of use case diagrams. This provides a view on the ways users can interact with the system. In our experiment, use case diagram of a simple conference information system can be simplified as follow:

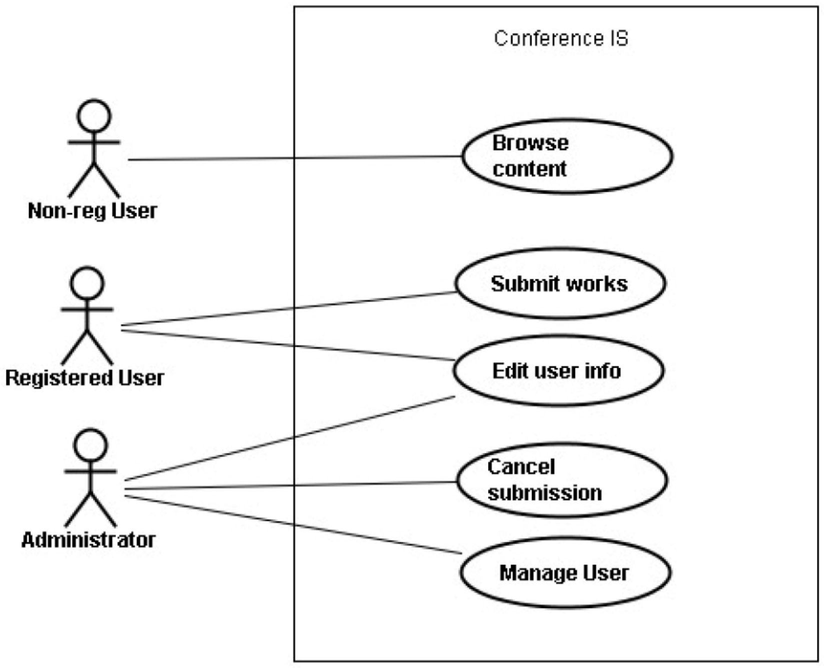


Figure 3.2: Conference IS sample use cases

It is important to start modeling with a simple system, through different iterations, more precise and complex features will be added via communication across multi functional members in the team. This includes business holders, project management, operations and other involved parties. In our case study, more user roles and behavior will be added along each iterations.

Model constraints can be specified with OCL. An example is unique properties: each user has an unique ID.

```
context User::UniqueUserId():Boolean
body: User.allInstances()->isUnique(userId)
```

2) Statechart diagram

When developing PIMs, statechart diagrams show the events that lead to changes in object states and specify the system response to these events. The state model includes a statechart diagram for each object with an important dynamic behavior. At the PIM level, statechart diagrams can be defined using a protocol state machine. For simplicity, we can assume an example of statechart diagrams for a paper could be started with:

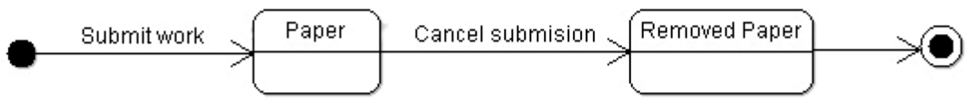


Figure 3.3: Starting with simple paper statechart diagram

Applying agile modeling to this section, our approach aims to follow the principles “assume simplicity, embrace change” and “rapid feedback”. It is essential to start with a simple model, this allows the statechart diagram to be reviewed fast with rapid feedback. In the weaving process of all aspects together, “rapid feedback” also plays a more important role in the building-up of models. The process involves communication with various stake holders. In this communication, models need to be simple with a single objective as in “modeling with a purpose”. Any change to the model will be reviewed via the incremental modeling phases. In this case study, the statechart diagram can be later enriched with various states emerged from feedback such as “on hold”, “waiting for review”, “camera ready submitted” etc.. Through many iterations, both with design and communications, the model will gradually reach the complete state.

3) Class diagram and operations

The approach includes applying OCL in specifying constrains, derivation and query. Consider the case of UML diagram as follow:

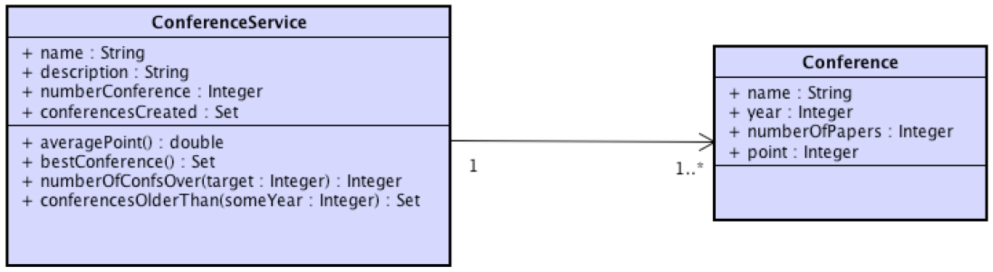

ConferenceService
+ name : String
+ description : String
+ numberConference : Integer
+ conferencesCreated : Set
+ averagePoint() : double
+ bestConference() : Set
+ numberOfConfsOver(target : Integer) : Integer
+ conferencesOlderThan(someYear : Integer) : Set
1
1..*
Conference
+ name : String
+ year : Integer
+ numberOfPapers : Integer
+ point : Integer

Figure 3.4: Conference class models

This is a class model diagram of a conference service from a basic starting point. There are a number of conferences. These conferences are documented with how many papers they have received and some information about them. In agile modeling, the principles apply to this phase includes “model with a purpose” by focusing on the useful features first, e.g. we only introduce properties of the class diagrams that we think will be used at the moment. By starting with a simple set of models, we follow “assume simplicity”, hence allows us to embrace change more efficiently. In order to be more agile, there is a need to validate the correctness of the model upon each iteration, this enables us to utilize test-first modeling as a combination of test driven and model driven development.

In order to facilitate testings, we need to query values from model to assert the correctness and validity of constraints or business logic. OCL can be used to query and derive from instance-model for this purpose. Let us define some of the operation needed in our case study. Let us assume that 2 points will be calculated for each paper submitted to a conference. Applying the techniques with OCL model querying from the previous section, several properties of the model can be achieved, the attribute point can be derived in OCL as follow:

```
context Conference::point: Integer
derive: 5*numberOfPapers
```

To get the number of conferences recorded in the service, we can derive as follow:

```
context ConferenceService::numberOfConferences :Integer
derive: conferencesCreated->size()
```

As we start from the most simple models, the agile principles of “embrace change” and “incremental change” can to be applied by feedback and validation at the end of each iteration. For this purpose, we need to run and test the models in order to verify the design. In the case study, our approach includes using OCL to query data at instance-level, starting form the simple ones first. One convenient way to query data with OCL is to use the OCL *select()* query. Assuming we have to query for the number of conferences that have the number of submitted paper greater than a certain value:

```
context ConferenceService:: numberOfConfsOverTarget(target:Integer):Integer
body:conferencesCreated->select(numberOfPapers>target)->size()
```

The OCL query above should return an integer value as the number of conferences having numbers of submitted paper greater than a value as target. If we need to query a collection, OCL can return the result also as a collection:

```
context ConferenceService:: conferencesOlderThan(someYear:Integer):Set(Player)
body: conferencesCreated->select(year<someYear)
```

This function returns a Set as result of the query. We know that some aggregate functions are often used in a query language such as Structured Query Language. As we see from previous section, in OCL this is not quite straight-ahead, but these functions still can be implemented. A query to find the average value of a collection can be implemented with OCL as follow:

```
context ConferenceService::averagePoint(): Real
body: conferencesCreated.point->sum()/conferencesCreated->size()
```

The equivalent query in SQL could simply be constructed as: *SELECT AVG(point) FROM Conference*. We see that most Database Management Systems (DBMS) provide many built-in aggregate functions and they can be used quite conveniently for cases like this. With OCL, however we have to manually define our functions for such purpose. In the case of finding the average value, we had to get the sum of all points and divided by the number of conferences we have. This function returns the average point of all conferences in the ConferenceService as a real number. In the case of finding the maximal value, our sample is to find the best conferences which are the conferences that score the highest number of points. Some calculations need to be performed:

```
context ConferenceService::bestConference():Set(Conference)
body: conferencesCreated->select(
          point = conferencesCreated->sortedBy(point)->last().point)
```

This OCL snippet required a sorting of all conferences according to their points. After that, we performed a projection to the list of all conferences with the condition where the number of points equals to the maximal number of points from that sorted list. This approach on querying instance-model for testing can promote agility by verifying the models agains specifications on each iteration hence reduce potential errors. The principle of developing working software via test-first approach in agile model driven development can be supported by OCL instance-model querying. This process continues after each iteration, allow changes to be introduced to the models faster.

In the case study, the development of PIMs was more agile by starting with a simple set of models, applying incremental changes via feedback from various stake holders in the project.

Communication is made important in the modeling process to increase the possibility for various parties to be involved early in the model design. The process was more efficient with fail-fast character by applying test driven development to modeling. The development of PIMs is completed after successful verification the models to the requirements and business logics that was added or changed through out many phases of the project. Lesson learned in this case study can be applied to the design of domain specific language for model driven development such that it needs to be simple enough to enhance readability and able to embrace agile principles by supporting incremental changes and involvement of more stake holders.

To support agile model driven development via test-first approach, OCL can be used to query data for testing at instance-model level. In general, OCL is quite handy in supporting simple queries despite the fact that we had to manually implement some of the aggregate functions and manually manipulate data. In term of expressiveness, OCL lacks the direct support for Product and Project operators [2], however, OCL can be used as a good tool for navigation, simple queries for deriving values and helpers to support testing in agile modeling.

In the MDA, a PIM describes a software system that supports some business independently of the implementation technology. There are approaches to bring more context to PIMs to extend the traditional notion of PIMs in the MDA by capturing the common aspects of a class of applications instead of modeling a particular system. The approach proposed in this section can also be implemented in this context. As the proposed guidelines are generic enough, the approach for agile development of PIMs could also be applied using the same principles on domain specific modeling languages (rather than UML) that capture a class of systems in a particular domain such as Simple Web Service Modeling Language (SWSM) which is introduced in Chapter 4.

3.4 Summary

In this chapter, we analyzed and covered several techniques for model driven development from our case studies. We outlined a formal specification of model transformation based on triple graph grammars, results from this formalism is later applied to the design of a domain specific language for modeling and testing of web applications in the following chapters.

We made an attempt to apply agile principles to the modeling process with a case study. Important principles to follow include: assume simplicity in starting models, incremental changes via short modeling iterations, focus on the important features by modeling with a purpose. This allows us to be efficient when coping with frequent changes and maximize stake holders' influence in the development process via rapid feedback. The work of the author on agile development of PIMs has been cited as references in US Patent No. 8516435 and No. 8495559. This study provides us with lessons to apply in our design of the domain specific language approach in the following chapters.

From practical viewpoints, presenting some layer of abstraction will increase effectiveness by making code (and spent time for preparing it) more useful and easily adaptable in the future. On the other hand, changes in requirements, platforms and sometimes even personnel require fast altering of existing artifacts at minimal costs. Hence, the need of an automatization process is in place and indispensable. In addition to that, to cope with the fast-changing environment, there is a need also for techniques that can be aligned with agile principles and potentially become an actual standard in the variety of existing tools available in recent emerging domains such as the web domain. In the next chapters, we introduce our approach using Domain specific language for modeling and exogenous transformation utilizing hybrid templating techniques for code generation.

Chapter 4

Domain specific language for modeling and transformation

4.1 Preliminaries

Cloud computing and distributed systems continue to gain more mainstream adoption as more companies move into the cloud. With mobile gradually taking over the desktop experience, cloud architecture continues to accelerate and provides more impact to the evolution of software [34]. Model-driven Engineering methodologies have been applied (as a solution) for better reaction to market trends and aims to increase efficiency as well as bring more agility to the development life-cycle of cloud and distributed systems. However, since there are many different applicable domains in web applications and distributed systems, the challenge is to introduce an approach that can effectively support automation in model-driven development of that application domain. In this chapter, we present our effort towards the solution for this issue by analyzing the concepts involved in key aspects of web service design and introduces an approach to the development of web services by using model-driven techniques with domain specific language. As a result, a DSL for modeling of web services named SWSM (Simple Web Service Modeling) was developed and introduced. To demonstrate this approach, a case study of web service development from modeling to code generation is also illustrated with the associated techniques. This chapter is structured as follows: In the next section, we review some knowledge of web services and domain specific language as background information. The subsequent section discusses the current state of web services development using model-driven techniques. We also highlight the features of the DSL that we aim achieve when designing a new DSL for modeling of web services. In the next section, we introduce SWSM - our designed DSL for modeling and development of web services and how to apply it at a specific point during design phase. In the last section, we present some conclusions on web service development using SWSM and its applications.

4.2 Background

4.2.1 Web services

With the growing demands in recent years, distributed computing and e-business processing systems are made possible by adopting a new paradigm of Service-Oriented Computing (SOC). SOC enables building agile networks of collaborating business applications across distributed locations and promote service oriented architecture. Web services are the current most important technologies based on the idea of SOC, and are self-describing, self-contained, modular components that can be published, located, and invoked across the web via standard interfaces and protocols [78].

The key technologies of Web services include XML (Extensible Markup Language), UDDI (Web Services Description Language, Universal Description, Discovery and Integration) and SOAP (Simple Object Access Protocol)[14]. Currently the development of web services in MDD involves using UML to specify services precisely and in a technology-independent manner. However, UML is not the optimal way for modeling of web services; the efficiency could be improved by using a specific language to address the detailed nature of web services. Introducing a new DSL can set up the stage for automatic generation of a part of the XML and code, such as Java code, that implements the services. It also makes it easier to re-target the services to use different web services implementation technologies when required.

4.2.2 Current Approaches in Web Service Development

Currently development of web services fall into two main categories associated with the order in which models are developed: top-down and bottom-up. In top-down approach we first design the abstraction and description of web services, after that we add more detail implementation and business logic to it, in this process, modeling is a crucial part. A good web service starts with a good design. The UML approach for this has some drawbacks, UML is a tool for generic design, it is not straight forward to address easily all the aspect of web service. Besides, creating XML/WSDL is a complicated process with a lot of detail information. In contradiction to that, modeling process at the first place intended to abstract away unnecessary details and makes it easier to understand the system. Hence, there is a need to create a better mechanism to solely support the design of web services. Using a dedicated DSL for this purpose can turn into a promising approach in this situation.

In bottom-up development, the design process starts with a given prototype or presentations of a class, other web service artifacts are generated from the given prototype. This means part of the implementation must be designed at the first steps. This approach means changes made from the first steps will propagate and require changes on all model artifacts.

Aligning with principles of MDD, top-down approach is considered more suitable and is one of the solutions to better support software reuse. Our approach using model specific language in top-down paradigm can be the feasible choice for development of web services.

4.2.3 Related work

Model-driven development of web services is still evolving to address the problem of complexity, integration, fast-changing technology in the software industry. Model-driven development of web-services is addressed in the work of Benguria et al.[7]. This approach focused on building platform independent models for service oriented architecture. The solution provides a platform independent meta-model for Service Oriented Architecture (SOA) and a set of transformations that link the meta-model with specific platforms following the Model Driven Architecture (MDA) approach. There are also existing UML-based approaches to modeling services. UML collaboration diagrams have been used extensively to model behavioral-aspects such as service collaboration and compositions in the work of Bezivin et al. [8]. In this approach, the Platform-Independent Model is created using UML. This PIM is transformed using Atlas Transformation Language to generate PSMs based on three target platforms: Java, Web Service and Java Web Service Developer Pack (JWSDP). This approach showed that UML profiles allow the extension of the UML meta-model. However, UML profiles make the creation of transformation rules difficult.

The support for software as a service and services modeling have been also addressed by providing lightweight extensions to UML through Profiles, these approaches can be seen in the work of Frankel and Parodi [26] and Bordbar et al. [9]. UML-profiles for services and SOA are proposed by the approach of Heckel [36]. This effort developed a suitable syntax for this domain by sketching a UML profile for SOA based on UML 1.x standards with a direct mapping between WSDL 1.1 elements and their model elements. Once the profile is properly defined, its semantics can be given in terms of a graph transformation. This approach has an advantage of UML generality, it can be used to model just about any type of application, running on any type and combination of hardware, operating system, programming language, and network. However, since UML is large and complex, using multiple models/diagrams makes it difficult to keep them consistent with each other and more code has to be added manually. There are also other efforts to provide domain specific languages for modeling of web services and service-oriented architecture. A qualitative, explorative study that provides an initial analysis of a number of such approaches through a series of three prototyping experiments in which each experiment has developed, analyzed, and compared a set of DSLs for process-driven SOAs can be seen in the work of Oberortner [62]. Maximilien et al. developed a DSL for Web APIs and Services Mashups [53]. This effort describes a domain-specific language that unifies the most common service models and facilitates service composition and integration into end-user-oriented Web applications. A number of interesting design issues for DSLs are discussed including analysis on levels of abstraction, the need for simple and natural syntax and code generation. On the track of non-UML-based modeling approaches, there are efforts aiming to aid modeling of services. The Web Services Modeling Framework (WSMF) from Fensel & Bussler [25] defines conceptual entities for service modeling. It is an effort to build the Web Service Modeling Framework (WSMF) that provides the appropriate conceptual model for developing and describing web services and their composition (complex web services). Its philosophy is based on the following principle: maximal de-coupling complemented by a scalable mediation service. Web-Service Modeling Ontology (WSMO) [47] provides a conceptual framework and a formal language for semantically describing all relevant aspects of Web services in order to facilitate the automation of discovering, combining and invoking electronic services over the Web. It has its foundations in WSMF but it defines a formal ontology to semantically describe web services. The Web Services Modeling Language (WSML) [16] provides a formal syntax and semantics for the WSMO based on different logical formalisms.

4.3 Challenges and Current Limitations

Advances on programming languages still cannot cover all aspects of the fast-growing complexity of web platforms. In a wide range of systems, especially distributed ones, more and more middle-ware frameworks are developed in languages such as Java or .Net, which contain thousands of classes and methods as well as their dependencies. This requires considerable effort to port systems to newer platforms when using these programing languages [34]. Therefore, general programming languages cannot be considered as first-class languages to describe system-wide and non-functional aspects of a system. There is a need to raise the level of abstraction while still providing specific domain attributes for modeling of such systems.

With mobile technology adoption continuing to gain momentum, in the next few years more cloud based and software-as-service (SaaS) systems will grow. As more systems migrate

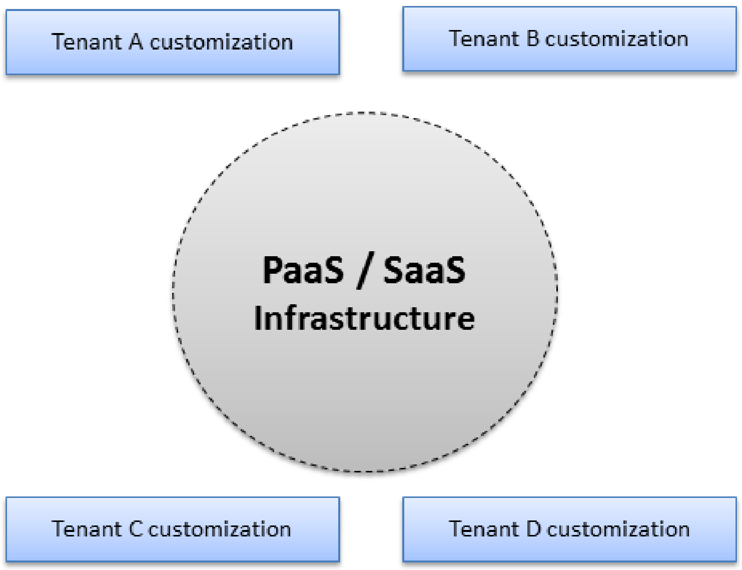


Figure 4.1: Multi-tenancy architecture service sharing

to the cloud, there is a big space for web services to continue gaining popularity. SaaS, and recently Platform-as-a-Service (PaaS), as different layers of cloud computing, require different approach to web service development and deployment. In these infrastructures, the so-called multi-tenancy becomes an essential factor. The multi-tenant architecture (as depicted in Fig. 4.1) ensures the customization of tenant-specific requirements while sharing the same code-base and other common resources. In this figure, four customizations of different tenants are built based on the shared service implementation and infrastructure. Web services in multi-tenant platforms need a way of abstracting away the configuration and make it possible for every part of the service to be customized for a specific tenant. These platforms are often built from the meta-data driven solution. This therefore means that the application logic can be based on meta-data which later can be customized [34].

The challenge in this architecture is to adopt or develop a modeling language at the appropriate abstraction level to separate the logical models from its technical aspects. This detaches the definition of service architectures independently from the used specific platforms. A modeling language raising the level of abstraction allows us to reuse models and keeps platform-specific artifacts at a separated tier in the development workflow. Having a modeling language based on services aspects with the ability to set aside technical concerns and still be able to tackle a problem in a specific platform is hard to come across.

There are existing general purposes modeling languages such as UML. UML is a standard for software system modeling. It is able to represent various kinds of software systems, from embedded software to enterprise applications. To allow this flexibility, UML provides a set of general elements applicable to any situation such as classes or relationships [7]. However, in the SaaS or PaaS architecture, the class systems in UML often forces systems to be represented or surrounded by classes, this could make the models difficult to understand and use. In addition UML provides facilities to specialize UML for a specific domain as so-called UML profiles. But frequently, these mechanisms are not able to represent the semantics behind the domain concepts [7]. The challenge therefore remains in defining a domain specific language that can be suitable for the modeling and development of this infrastructure. In the case of modeling web services, the creation of a high-level DSL turns into a necessity for software reuse, higher development speed and better cost-effectiveness.

4.4 Features of a DSL as a modeling language

Introducing a new DSL with the support for modeling at a good abstraction level is crucial. This DSL can later be used for automatic generation of the model artifacts and code that implement the services. In theory, a general modeling language could also be used for this purpose but an appropriately designed DSL will perform the same job much more effectively. We define a set of features that are essential to the DSL design in model-driven development of web services. All of these features should be considered during the creation of a DSL to ensure the quality of the language.

Effectiveness: The language needs to be able to deliver useable output without having to re-tailor based on specific use case while being easy to read and to understand. This means that the language is able to bring up good solution on specific domain and focus on solving the particular range of problems. Effectiveness also needs to guarantee the unambiguity feature of language expressions and capability to describe the problem as a whole from a higher level.

Automation and Liveness: As the modeling language can raise the level of abstraction away from programming codes by directly using domain concepts, an important aspect is the ability to generate final artifacts from these high-level specifications. This automation transformation has to fit the requirements of the specific domain. Liveness feature ensures changes from models described by the language are propagated to the next phase of development automatically.

Support Integration: The DSL has to be able to provide support via tools and platforms. The DSL needs to be able to integrate with other parts of the development process. This means that the language is used for editing, debugging, compiling and transformation as well as integrated together with other languages and platform without any heavy effort.

When designing and implementing DSLs as executable languages, there is a need to choose the most suitable implementation approach. Related work from Mernik et al. [55] identifies different implementation patterns, all with different characteristics. These patterns provide another perspective to consider when making the design decision of DSL. These options can be broken down to the following categories:

- *As interpreter*: In this method, DSL constructs are recognized and interpreted using a standard fetch-decode-execute cycle. With this pattern no transformation takes place, the model is directly executable.

- *As compiler/application generator*: DSL constructs are translated to base language constructs and library calls. People are mostly talking about code generation when pointing at this implementation pattern.

- *Using pre-processor*: DSL constructs are translated to constructs in an existing language (the base language). Static analysis is limited to that done by the base language processor.

- *Embedding design*: DSL constructs are embedded in an existing GPL (the host language) by defining new abstract data types and operators. A basic example is application libraries. This type of DSL is mostly called an internal DSL. The good side of this is that grammar, parsers and tools are immediately available. However, the challenge with an embedded DSL is to tactfully design the language so that the syntax is within the confines of what the host language allows, yet is as expressive and concise.

- *Using extensible compiler/interpreter*: A GPL compiler/interpreter is extended with domain-specific optimization rules and/or domain-specific code generation. While interpreters are usually relatively easy to extend, extending compilers is hard unless they were designed with extension in mind.

- *Commercial off-the-shelf*: existing tools and/or notations are applied to a specific domain. In this approach, it is not needed to define new DSL, editor and DSL implementation, one only needs to make use of a Model Driven Software Factory. One example is using the Mendix Model-Driven Enterprise Application Platform targeted at the domain of Service-Oriented Business Applications.

- *Hybrid*: a combination of the above approaches.

The choice of the approach is very important because it can make a big difference in the total effort to be invested in DSL development. With the success of open source projects like Xtext, development of DSL is made affordable and the development is focused on building the grammar, while support for static analysis and validation of models are possible out of the box.

We aim to maintain the set of features defined in this section while designing SWSM. This allows us to provide automatic transformation, agility and integration to the development cycle. This ensures that the process of model-driven development of web services using SWSM is efficient.

4.5 Model Driven Development of Web Service using SWSM

Web service technologies depend on the use of XML, SOAP, WSDL. These standards are important, but they do not effectively support automation of code evolution at different phases in the multi-tenancy development cycle. A DSL for modeling web services is therefore useful because it can effectively support automation in model-driven development. In the process of designing a suitable DSL for this purpose, we consider some valuable lessons described in the work of Wile [77]:

Lesson T2: You are almost never designing a programming language. Most DSL designers come from language design backgrounds. There the admirable principles of orthogonality and economy of form are not necessarily well-applied to DSL design. Especially in catering to the pre-existing jargon and notations of the domain, one must be careful not to embellish or over-generalize the language.

Lesson T2 Corollary: Design only what is necessary. Learn to recognize your tendency to over-design.

Keeping this principle as an effective approach in designing, we made an effort to design SWSM as a modeling language for web services at the appropriate abstraction level. As a proof of concept, this language aims to increase the efficiency of the development process by letting user focus only on the modeling of essential aspects that comprise web service.

The syntax needs to be simple yet as expressive and concise. The possible set of simplified syntax diagrams for the components of this DSL can be depicted as in Fig. 4.2.

To describe the service as an aggregation of several ports, the keyword *webservice* is used for modeling web services. Following is the syntax diagram for this model declaration (Fig. 4.3).

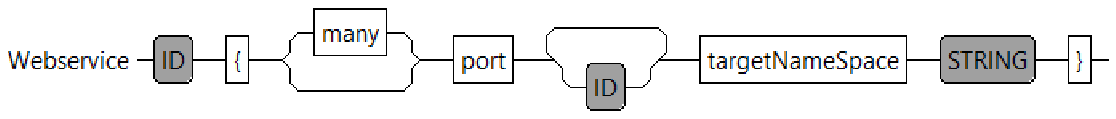


Figure 4.3: Web services syntax with SWSM

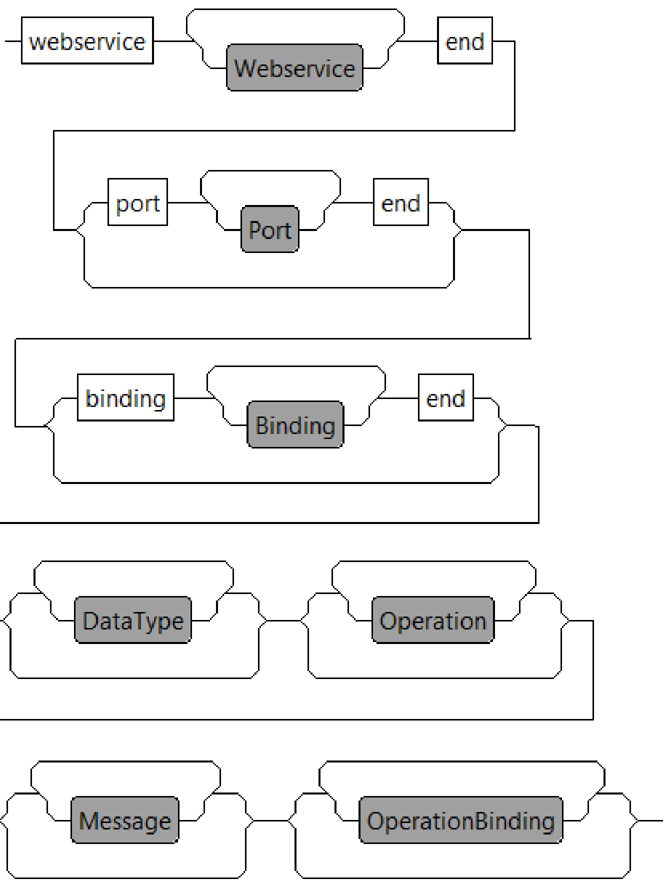


Figure 4.2: Simplified syntax diagram of web services with SWSM

The semantics of the language expressions starts with the web service definition followed by its name. There could be a number of ports associated with this web service and this mapping is described by the *port* keyword followed by a string identifier of a port. It is worth to mention that ID is a term representing the name (identification) of an element. Value of the target namespace is a string followed by *targetNamespace* keyword. This enables developers to specify the relationship between a port and a particular web service. In most cases the association is a one-to-many mapping. The syntax diagram of ports can be depicted as in Fig. 4.4.

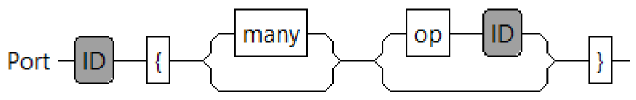


Figure 4.4: Port syntax with SWSM

A port identified by name (ID) consists of one or many operations, each operation is then defined by the set of input and output. This semantics can be seen in the syntax diagram of an operation in Fig. 4.5.

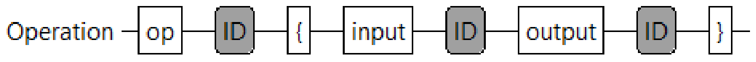


Figure 4.5: Operation syntax with SWSM

Each input and output of an operation is of the type *message*. The keywords *input* and *output* make the semantics of an operation signature easy to comprehend by describing the parameters for the operation with its returning type. The *message* element defines the data elements of an operation. Each message can consist of one or more fields (parts). These fields play the role of the parameters of a function call as in a traditional programming language. All modeled fields form the method signature for each operation.

Given a collection of operations $O_1 \ldots O_n$ with associated input and output messages, we define the mapping to web services and ports:

- ▶ One or more operations ($O_1 \ldots O_n$) are mapped to a port P1 to describe one function

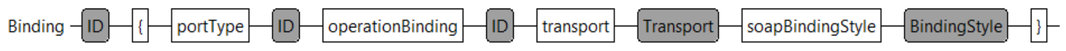


Figure 4.6: Binding syntax with SWSM

of a web service.

- ▶ P1 defines the connection point to a web service, one or more web services ($W_1 \ldots W_n$) are modeled within an SWSM file.

The message format and protocol details for a web service are modeled via *binding*. A binding is identified by its name (ID on the diagram), the mapping to a *port* is described by port attribute. The binding style is represented by *bindingStyle* attribute. Value of *transport* attribute has the direct semantics of defining which transportation protocol to use. For example, in the case of HTTP, we can simple assign "*http*" to transport. This is more convenient than the approach currently used in WSDL where "http://schemas.xmlsoap.org/soap/http" is assigned. To define the operations that the port exposes, the mapping *operationBinding* is used. For each operation binding, the corresponding SOAP action is described with its encoding type of input and output.

The best way of illustrating the syntax is to start modeling web services in a case-study. The first step in model-driven development of web services is designing the models. The output of this phase are models that conform to a web service meta-model, which can be represented in a textual format complying to the grammar of a DSL. Model artifacts are later used as input for the generation process. One of the important influencing factors is that any changes in the models will propagate changes in other stages. SWSM has a mechanism to support change propagation. To start modeling web services with SWSM, the process begins with representing the principal elements of a web service in the modeling language:

- ▶ *Types*: used to define the abstract elements in the description of the web service. They can be simple or complex type. Identified by the keyword *type*.

- ▶ *Messages*: are units of information exchanged between the web service and the customer application (logically they are input, output messages and sometimes also fault messages). Each operation provided by a Web service is described by at most, one input message and one output message. These messages relate to the parameters of the operation. In SWSM messages are identified by the *message* keyword.

- *Interfaces* (or portTypes in WSDL1.0): they constitute aggregations of operations provided by the service. In SWSM interfaces start with the keyword *interface*.
- *Bindings*: they specify in particular the protocol used to invoke the methods of an interface. In SWSM bindings start with the keyword *binding*.
- *Services* and ports: the service can constitute an aggregation of ports. A *port* is an endpoint enabling the access to an interface through an URI address. Services are identified by the keyword *webservice*. We can define multiple web services within a single design.

Utilizing MDD principles, web service development using SWSM can be decomposed into four steps:

1. Modeling the web service using SWSM.
2. Enhancement and automatic validation of web service models.
3. Generating Java code using built-in code generation feature of SWSM.
4. Code refinement, refactoring and testing.

To demonstrate the modeling syntaxes, we can see how SWSM is used to represent various essential elements of web services in a case study. To model the service as an aggregation of several ports, the keyword *webservice* is used. The code bellow shows how the keyword *webservice* is used to define a service called *DictionaryService*:

```
webservice DictionaryService {
port LookUpPort
```

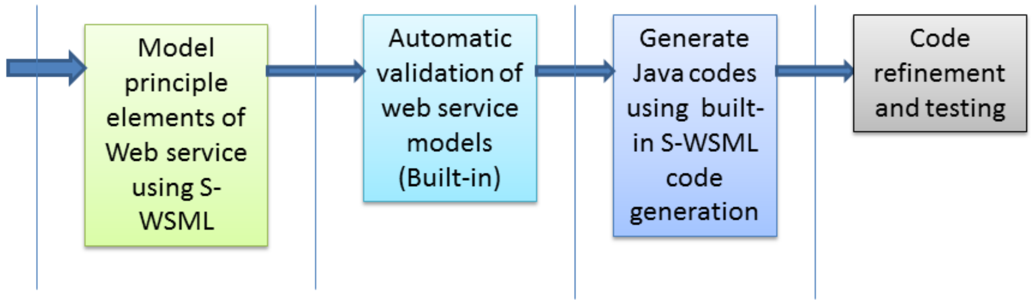

Model principle elements of Web service using S-WSML
Automatic validation of web service models (Built-in)
Generate Java codes using built-in S-WSML code generation
Code refinement and testing

Figure 4.7: Development of web services with SWSM

```
targetNameSpace "http://ws.mydictionary.net/lookUp" }
end
```

The semantics of the *webservice* block indicates a web service, which can consist of one or many ports. This can be seen from the syntax diagram described above. However, on this dictionary look-up example, there is only one port named *LookUpPort* declared. In the next step, the ports associated to the service also need to be defined:

```
port  LookUpPort {
op LookUp
}
end
```

In a port, there are operations involved, *LookUp* operation is the one in this case. A port is associated with an *interface* by the association *binding*. The meta-class *interface* constitutes an aggregation of several operations.

```
binding LookupBinding {
portType  LookUpPort
operationBinding  OpBinding
transport http   soapBindingStyle rpc
} end
```

Each operation (identified by the keyword *op*) consists of *message*(s) which defines its inputs and outputs. A *message* naturally refers to a meta-class indicating its simple or complex type:

```
op LookUp {
input Input
output Output
}
message Input {
username : String
word: String
}
message Output {
meaning : String
wordType : String
related : Integer
}
```

In addition, we can also define the action associated with an operation by using the keyword *opBinding*:

```
opBinding OpBinding {
soapAction "http://me.com/action"
inputSoapBody literal
outputSoapBody literal
}
```

For the HTTP protocol binding of SOAP, the value of soapAction is required. For other SOAP protocol bindings, this value could be omitted. The *inputSoapBody* and *outputSoap-Body* keywords indicate whether the message parts are encoded using some encoding rules.

Putting all the pieces together gives us the information needed to model a simple web service. These models are later used as input for code generation. SWSM makes it possible to design web services by using simple and fast syntaxes. In contrast to other approaches, SWSM is uncomplicated, rapid and easy to adapt. The syntax used in SWSM is simple and more intuitive in comparison to the complex structure of UML. The order in which aspects of a web service are defined is the same as the logical order, when we design a web service. This makes the designing process more natural and perceptive, hence allows agile principles to be applied easily. Using SWSM enables us to focus only on the essential aspects of the web service. This approach promotes model-driven development principles and makes the web service development process more efficient [57].

All of the SWSM language infrastructures can be packed as a plug-in for the Eclipse integrated development environment. This includes a text editor with autosuggestion and validation capabilities. This enables the development phase to be carried out seamlessly. We also built a code generation feature (Java language) based on a code template engine and embedded it into SWSM. Textual models created using SWSM are used as input for the generation of Java web services. Code generation can be executed right within the editor.

The role of MDA in this development process is to raise the level of abstraction in which we develop systems. This is aimed to improve productivity similarly as when we moved from assembly language to third-generation languages. At first, third-generation language compilers did not produce code as optimal as hand-crafted machine code. Over time, however, the productivity increase justified the changeover, especially as computers speeded up and compiler technology improved [26]. SWSM is similarly used at a different level of abstraction to third-generation programming languages, to tackle overall productivity.

4.6 Summary

MDD approach can be applied to web services in order to increase the resilience of implementations, as web services technologies change and evolve. This chapter brings up the design theory and methodology for implementing and utilizing a domain specific language for model-driven development of web services. Adopting domain specific languages, such as the one we introduce, can increase productivity and ease the burden of development of web services as the backbone on SOA systems. SWSM also reduces the cost implied in maintaining the systems and provides a solution to software reuse.

SWSM was written at a good abstraction level. This improves code readability and makes program integration easier. SWSM enables users without experience in programming at a higher level to focus only on knowledge of their concerned domain. Hence under this approach, it is possible for different stakeholders such as business experts and IT experts to model web services during early stages of web services design. This enhance the agile modeling capability in maximizing involvement of stake holders. Another advantage of SWSM for modeling is the ability to generate more verification on the syntax and semantics than a general modeling language. This can reduce errors on the testing or debugging process. However, we also need to point out that this approach has several drawbacks. There is an extended learning curve for a new language, even though SWSM as a domain specific language proves to be a lot easier to learn than a general programming language. Additionally, as a general language is adopted by more people, it could be more feasible to find staff capable of solving the problem using their language knowledge. There are also spaces for improvement in the syntaxes of SWSM.

In practice, approach using SWSM can be applied to the web service development process in various environments. As members of our team are working with companies in the top global Fortune 500 dealing with large-scale web services for financial services and telecommunication industries, the outlined approach has started to gain adoption in equity research department of the investment bank and initially has been applied successfully.

Chapter 5

Model driven web testing framework with DSL

5.1 Motivation

As more and more systems move to the cloud, the importance of web applications have increased. Web applications need more strict requirements in order to support higher availability. The techniques in quality assurance of these applications hence become essential, therefore the role of testing for web application becomes more significant. The reasons for validating the correctness of web application in model driven development include:

- Web-based applications and pages are error-prone: Web-based applications are often more fragmented due to its nature of having many components. These components are developed by humans and susceptible to errors. In some situations, web application specifications need to support complex business logic which makes it hard to specify correctly from the beginning. The need to support multiple concurrent access in the nature of web application also makes the implementation more error-prone.
- Web applications need high availability: More strict up-time criteria are demanded in web applications nowadays, this requires the specification to be followed more strictly,

hence testing needs to be more intensive. Web-based applications and pages provide a bridge of sharing applications and information between various locations. Changes to one system may affect many other downstream ones. This leads to the need of testing to be performed at many levels to assure the system is precise and resource-efficient.

- The process of test development is time consuming: Web applications and pages are usually built based on a variety of technologies. Developing test cases for this type of system often needs repetitive work, not to mention the execution of regression testing, thus more automation is needed in this process in order to reduce time-to-market.

Model-driven test development can be applied in order to obtain better automation of software testing. This approach can reduce the repetitive cycle of test development and execution. In the web domain, the challenge however remains in the creation of models and the complexity of configuring, launching, and testing big number of valid configuration and testing cases. This chapter proposes a solution towards this challenge with an approach using Domain Specific Language for model driven testing of web application. Our techniques are based on building abstractions of web pages and modeling state-machine-based test behavior using domain specific language. This methodology and techniques aim at helping software developers as well as testers to become more productive and reduce the time-to-market, while maintaining high standards of web application quality. This section also outlines the theoretical ideas and analysis from lessons learned during the real industry implementation of the framework.

5.2 Model based testing

We analyze model-based testing from the model driven development perspective. Model-based testing can be considered as one component in model-based design. In the computer world, it is used for designing and optionally also executing artifacts to perform software testing or system testing. Models can be used to represent the desired behavior of a System Under Test (SUT), or to represent testing strategies and a test environment.

A model describing a SUT is usually an abstract, partial presentation of the SUT's desired behavior. Test cases derived from such a model are functional tests on the same level of abstraction as the model. These test cases are collectively known as an abstract test suite. An abstract test suite cannot be directly executed against an SUT because the suite is on the different level of abstraction. An executable test suite needs to be derived from a corresponding abstract test suite. The executable test suite can communicate directly with the system under test. This is achieved by mapping the abstract test cases to concrete test cases suitable for execution. In some model-based testing environments, models contain enough information to generate executable test suites directly. In others, elements in the abstract test suite must be mapped to specific statements or method calls in the software to create a concrete test suite.

Tests can be derived from models in different ways. Because testing is usually experimental and based on heuristics, there is no known single best approach for test derivation. It is common to consolidate all test derivation related parameters into a package that is often known as ”test requirements”, ”test purpose” or even ”use cases”. This package can contain information about those parts of a model that should be focused on, or the conditions for finishing testing (test stopping criteria) [76].

Currently, testing usually comprises between 30% and 70% of all software development projects [58]. Hence, a good testing methodology and toolset will enable software developers and testers to become more productive and reduce the time-to-market, while maintaining high standards of software quality.

The purpose of the model-driven testing in the web domain is to provide a framework that helps developers to perform the following tasks:

- Create models of web applications or pages: This enable developers to create the abstraction of the components. Developers can later use the model created as a skeleton for the test project. In this way, the test plan can be reviewed and simulated to discover problems in the implementation or model before the actual code is ready for test.

- ▶ Model behaviors: The behaviors and interactions of the web application and pages are modeled using the modeling language to later support test case generation. These behavior models simulate the features of the web application.

- ▶ Generate test cases for the web components: The tools generate tests using data from the component (page) models and the behavior models. It is often a good practice to have the test cases that cover all required test specifications.

- ▶ Test execution: The generated tests can be later either manually or automatically executed by some triggers. This test execution automatically compares the observed results with the results predicted by the model. Thus, developers can walk through an unit test case to examine each test interaction and identify where the test failed.

In the each phase (task) of the process, there are sill challenges need to be resolved in order to achieve a more efficient process in model-based testing of web applications. In the next section, we analyze some of the challenges in this area.

5.3 Challenges

The process of web application development starts with concepts, mock-ups and requirements. After that, following many iterations, more and more mature prototypes are gradually created towards a working solution. Testing needs to be performed at every iteration in this process. This nature makes testing web applications a routine task from designing the tests to tests execution and report. When maintaining such systems, any additional change to the system also requires the execution of a complete regression test. Therefore, there is a need to build a testing platform that can automate this testing process from development to execution.

There exist many model-based testing approaches and tools which vary significantly in their specific designs, testing target, tool support, and evaluation strategies. In the web domain, there is a noticeable increase in the number of model-driven testing techniques in recent years. The challenge in this area is firstly to have a good design of a modeling language that used to represent the system. Secondly, there is the challenge for effectively defining the process of test case generation and evaluation. There are several aspects of a model-driven testing technique that need to be considered:

- Effective Modeling Language: The modeling language used to model the system for testing, as a domain specific language, should bring up good solution on the web domain while being easy to read and to understand. This language needs to be effective and designed with agility support to ensure that models can adapt to changes seamlessly.
- Automation: This is an important aspect in model driven development, it is the ability to generate final artifacts from high-level specifications. Automation also enable test case generation and execution mechanism to perform easily without manual refinements.
- Tool Support: The tool chain and platform support is essential for any approach. This allows the integration with other parts of the development process. This means that the platform should provide tools for editing, debugging, compiling and transformation. It should also be able to be integrated together with other languages and platforms without a lot of effort.

Although there exist several techniques with different designs, they usually don't provide adequate results in every applicable domain [58]. There are also challenges in other aspects of the modeling process: on one hand, the model has to be written in a notation powerful enough to describe any elements of the web page. On the other hand, it has to be abstract enough to ease the process of model creation and promote software reuse.

In the next section we introduce our approach and analyze how these challenges can be addressed using our designed DSL for model-bases testing of web applications.

5.4 Approach on using DSL for web page modeling

Our approach is based on the principle of raising the level of abstraction by modeling web pages and describe their behaviors using the theory of State Machines. In order to check the conformance between the application and the model, the automated process for generating test cases from the model is used. Our approach uses DSL to develop the testing model together with the functional web page model development. We aim at introducing a DSL and the tool set that fit for this purpose.

In this approach, designing a new DSL with the support for modeling at a good abstraction level is crucial. This DSL can later be used for automatic generation of the model artifacts and code that implement the services. There are three essential requirements to the DSL design that we aim to achieve during the creation of a DSL to ensure the quality of the language. Firstly, the language needs to be effective, while being easy to read and to understand. Secondly, as the modeling language can raise the level of abstraction away from programming code by directly using domain concepts, automation needs to be achieved to generate final artifacts from these high-level specifications. This automatic transformation at the same time has to fit the requirements of the specific domain. Finally, the DSL has to be able to provide support via tools and platforms. The DSL needs to be able to integrate with other parts of the development process. This means that the language is used for editing, debugging, compiling and transformation. It should also be able to be integrated together with other languages and platforms without a lot of effort.

The starting point for a DSL for web page modeling is an abstraction of a web page. This abstraction model comprises the effective elements that are involved in the testing process and, optionally, the behavior of the transitions to be simulated and validated during the test execution. The following diagram (Fig. 5.1) depicts the simplified syntax rules of a page model:

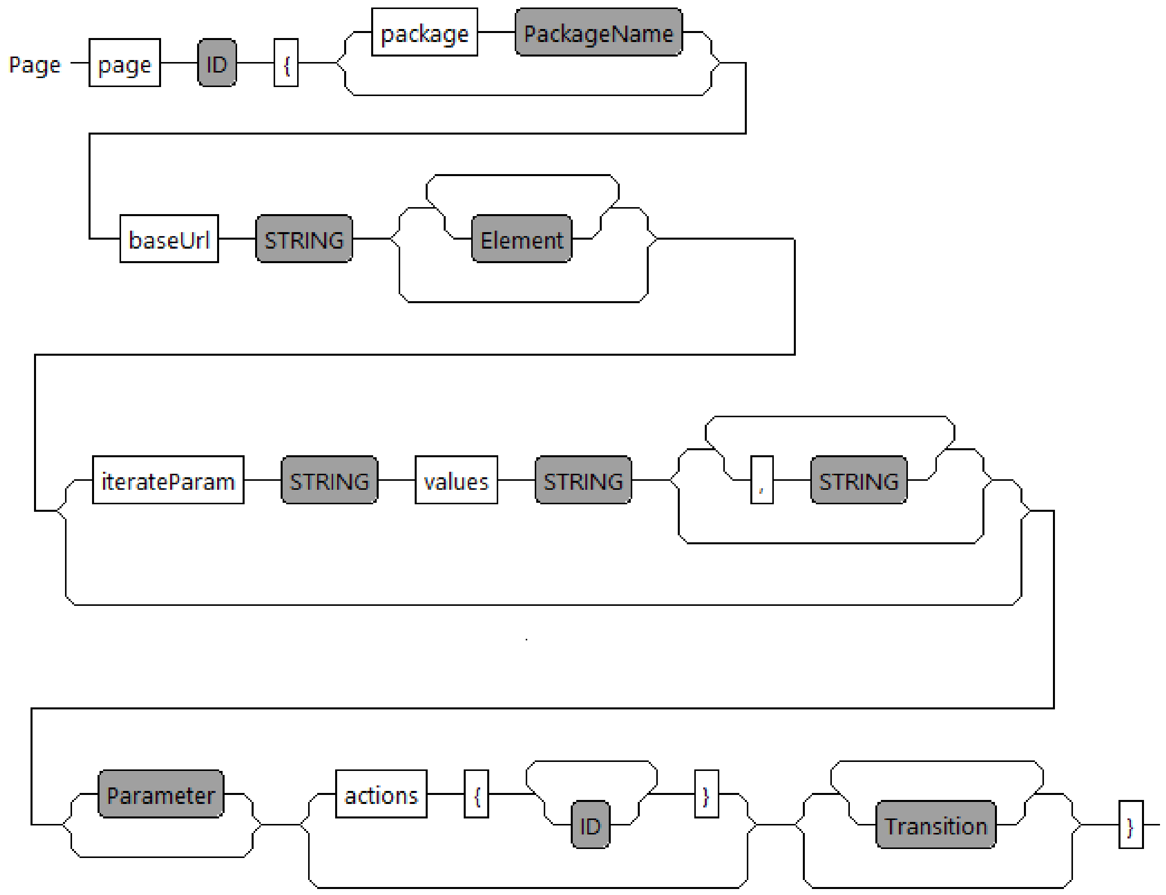


Figure 5.1: Simplified syntax of a Page in WTML

The semantics of the language expressions starts with the page definition identified by its name (ID). In order to have package information for code generation, a package name can be optionally declared. Base URL is then assigned to each page. This gives us the possibility for customization of the parameters for the URL. Main information for a page are the elements. A page can have arbitrary number of elements. In order to to query elements in a web page, we identified it with the XPath expression. The syntax for an element can be seen as in Fig. 5.2.

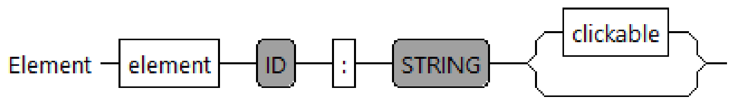


Figure 5.2: Syntax of an Element model in a page

Each element starts with the keyword *element* followed by its name (ID). We then use a string literal to store its XPath expression. An element can optionally be clickable, this can be declared by the keyword *clickable*.

We then define the parameters of the page. A page can have any number of parameters. Each parameter starts with the keyword *param* as in Fig. 5.3.

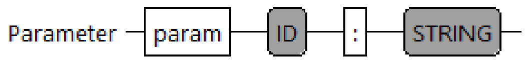


Figure 5.3: Syntax of parameters in a page

Another type of parameter can be seen in the next block in a page as in Fig. 5.1 is the set of parameters to later be used in the code generation process to repeatedly test against. This is defined by the keyword *iterateParam.* The parameters for iteration are comma-separated. This makes the focus on regression testing easy, since repetitive tests using different parameter configurations can be achieved with this declaration.

The last components are the actions and transitions. These are the optional components to define the actions and transition between pages. This can be used later when we want to use state machine to model the test cycle of the whole web platform. This is described in the next section.

To demonstrate the simplicity of the model creation process in this approach, we can see how simple it it to write a textual model of a web component from a web application in a case study.

```
page RatingPage{
   baseUrl "http://testhost/www/test"
   element content "//*[@id='content']"
   element user "//*[@id='user']"
   element submit "//*[@id='submit']" clickable
   iterateParam itemID "12,13,14,15,16"
   param action "add"
}
```

This eight-line-of-code model at this abstraction level allows us to be very flexible on building the elements and logic needed for the test. At this level code reused is heavily promoted. This can be reused on many pages yet enables us to generate large amount of codes for test automation. Our benchmark pointed out that 90 lines of concise Java code were generated from this. This means we saved a significant amount of time that was otherwise supposed to be spent on test development. Overall, even if we take into account the time spent on developing and learning a new DSL such as WTML, this can still potentially provide a good productivity gain in test development.

5.5 Modeling of page behavior using state machine

In order to model the behavior of the web page, we use state machine to model the actions and transitions between states of the page. With lessons learned from the successful application of state machine in different fields such as: definition of programming and modeling languages, modeling e-commerce and web services, we design a behavior model that is simple enough to model the states and transactions associated with various types of events but yet abstract enough to be able to address problems in a wide range of web page modeling challenges.

Overall, the behavior model can be depicted as:

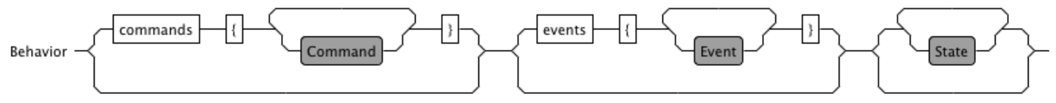


Figure 5.4: Model of behavior of a page

Our behavior model consists of a set of commands that users can perform on the page, a set of events that can happen on the page, and a set of states that can be activated from the set of input events. This allows us to describe the actions on the page that trigger different events, the set of commands that users are able to interact with on the page and the possible state transitions.

In each state, we define the actions that can be performed within the state scope, the transitions that can be triggered from the current state:

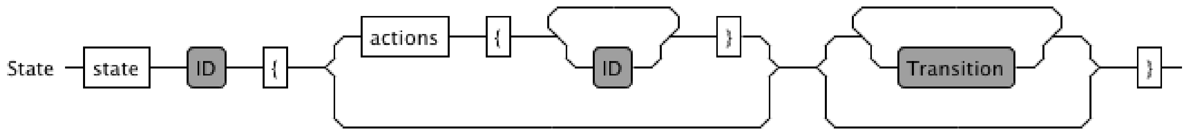


Figure 5.5: State model of a page

The state model starts with the keyword *state* followed by its name (ID). Each state can consist of one or many actions and transitions. The state model does not have to know about how the actions are defined or implemented, it just have to know about the set of command IDs in each action. This way, we separate the state from other implementation aspects of the page, making it possible to abstract away the specific details and making code reuse more easy.

The next component of the behavior model are events. Events can be simply modeled as:

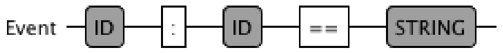


Figure 5.6: Events in a page model

In this model, each event is defined by its name (ID). We solve the modeling and triggering of events by creating a model that can describe how the event could be generated. This is done by having the command ID (the second ID block) followed after the event name (defined by the first ID block). This command is executed in order for the event to be fired. Finally, an assertion string could also be added in case we want to assert the outcome of a command

and fire the event accordingly.

Following the page model from previous section, we define the model of a command which interact with the elements on the page as following:

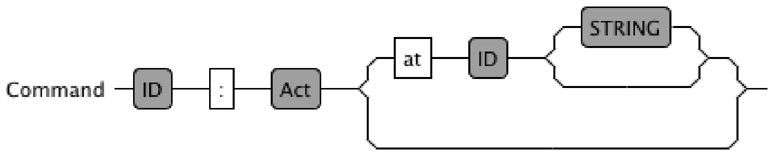


Figure 5.7: Model of commands in each state

The command is defined by its name (ID), followed by the literal ":" and the type of action that the command is executed. Example action type of the page could be TYPE, CLICK, SUBMIT etc. These options are defined as an enum of type *Act.* Optionally we can define the location where the command is aimed at by the keyword *at* followed by the element ID that was defined on the page as described on the previous section.

The last block on a state definition are transitions. Transitions are defined inside each state as we can see previously on the model of state. Since transitions are defined for each state, we just need to supply the event ID and the target state ID that this event triggers to.

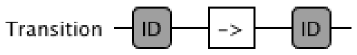


Figure 5.8: Model of transitions defined in each state

The first block is the event ID followed by the symbol "->" and the new state ID that this event triggered to. Transitions complete the model of states that a page can be in.

To implement the transformation from state machine to programming codes, one approach is to utilize the theory from triple graph grammars. In object oriented languages, we can realize states as classes, transitions then can be described as methods inside these classes. Given the syntax graph G_S of the models, code generation can be done via the transformation from G_S to the syntax graph G_T of the target programming language. The mapping between both graphs via a correspondence graph G_C forms the triple graph grammar.

There have been attempts to build the syntax graph of Java as the target programming language in various ways [71]. With the help of Java Development Tools (JDT) inside Eclipse IDE, we can achieve this by using the given API from the plugin. There have also been work to construct type graph model for Java programs as seen in [66]. Graph rules and grammars allow us to define the transformation as seen on Chapter 4. It is important to have the mapping from the syntax graph of the state machine model to the syntax graph of Java code. Let us look at a code example in Java for a sample state with the transition methods:

```
...
public class HomeState implements State {
    @Transition(target = FormSubmitedState.class)
    public void submitForm(TestPageModel testPage) {
        testPage.typeUserName();
        testPage.typePassword();
        testPage.clickSubmitButton();
    }
    @Transition(target = ContextMenuState.class)
    public void gotoContextMenu(TestPageModel testPage) {
        testpage.clickContextMenuButton();
    }
}
```

In this example, the *State* interface is a marker interface to indicate a state object. The *@Transition* annotation defines the target state and the transition conditions. Each state is defined by a class, the transition between states can be described by the methods inside that class. Within the transition method, we can execute actions on the page hence make it possible for the test cases to be executed. This can be represented as a syntax graph as in Fig. 5.9.

With this representation and the syntax graph of the models, we have both source and

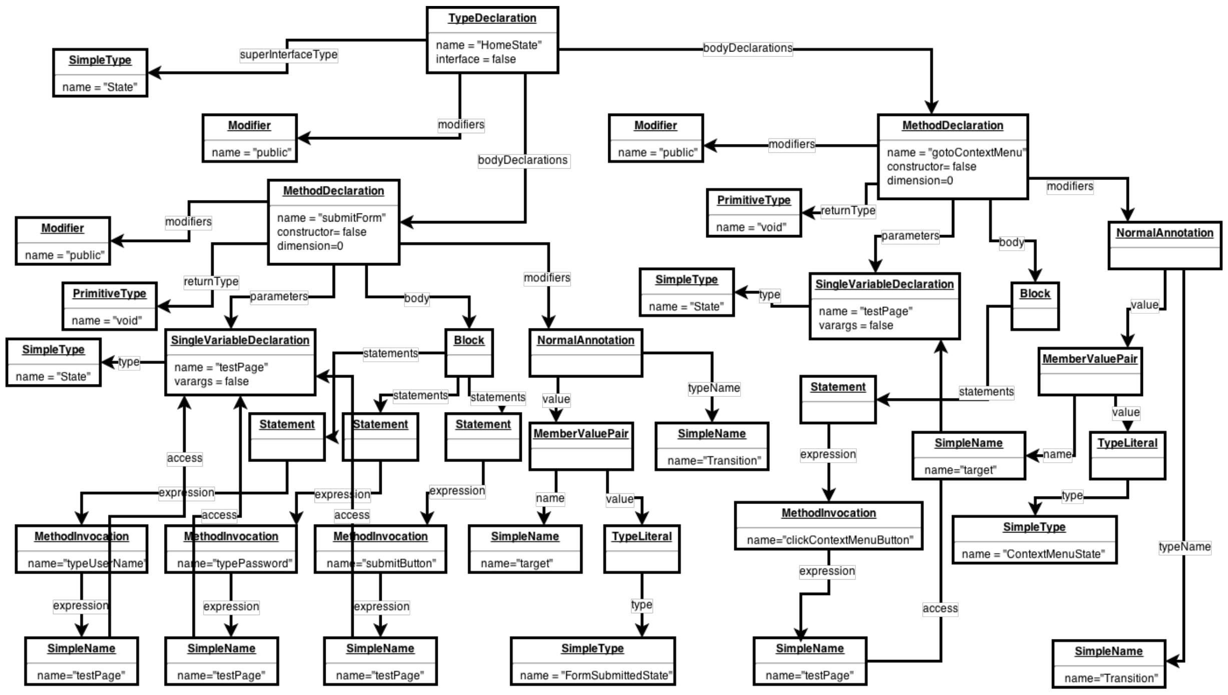


Figure 5.9: Represented syntax graph of the sample source code

target models represented by graphs. Hence, we can apply graph transformation to build the connection between models and codes. We need to derive the mappings between G_S and G_T that can provide the elements in which will be needed for defining the transformation. In this sample, we define the mapping of the state transition with the associated Java code. We can collect the necessary attributes that need to be translated based on the attributes from the graph. The mapping between a state transition and Java codes can be seen as in Fig. 5.10.

In this mapping, directly mapped elements are denoted by dashed lines. The target state “ContextMenuState” and other non-related elements from all graphs are omitted for simplicity. The reference from a transition to its target can be derived in the Java syntax graph by graph pattern matching without attribute comparison. In this case, the node SimpleType with name = “ContextMenuState” can be connected to TypeDeclaration by an additional edge.

As proven in section 1, chapter 4, with triple graph grammar we can define rules that can support the adding, removing of states and their transitions, which can be used to propagate

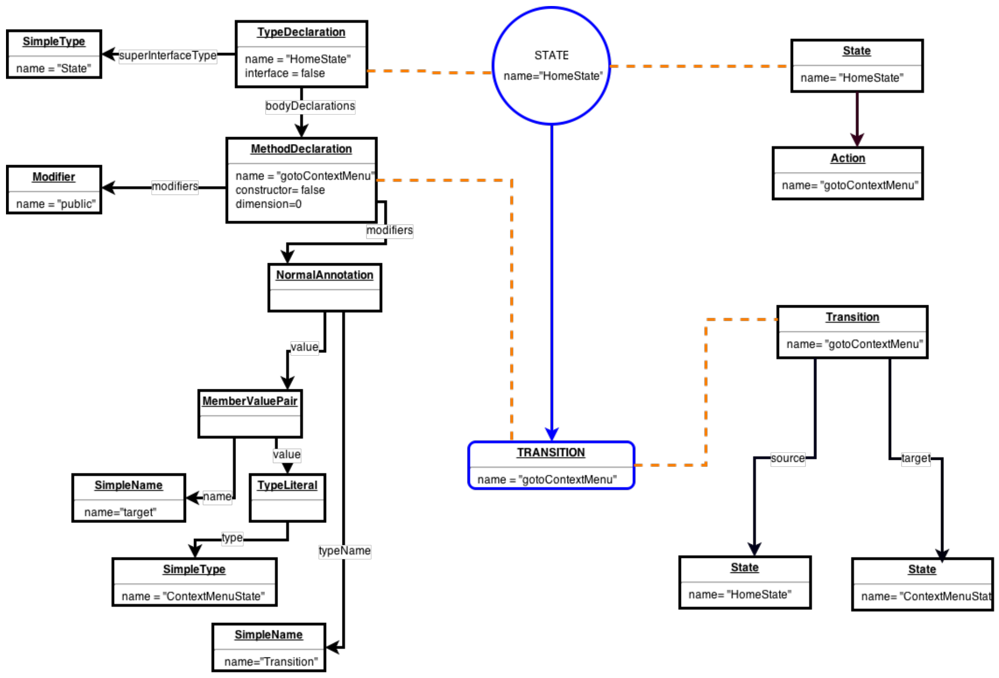


Figure 5.10: Mapping a sample state transition between programming language (left) and WTML (right)

changes from G_S to G_T. Recall from Def. 4.: *A typed graph rule* $p = L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ *consists of three typed graphs L: the left-hand side graph, K: the gluing graph, R: the right-hand side graph, and two injective typed graph morphisms l and r.*

In theory, if a rule p is applicable to a graph G via a morphism m : L → G, called match, the transformation $G \overset{p}{\Rightarrow} H$ is defined by two pushouts or double-pushout (DPO)*:*

$$\begin{tikzcd} L \arrow[d, "m"'] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"'] & R \arrow[d, "m*"] \\ G & D \arrow[l, "l*"] \arrow[r, "r*"'] & H \end{tikzcd}$$

In our case, a sample rule for adding of transition between two states can be seen as in Fig. 5.11. In this scenario, we can see that in each component we have elements that origin from G_S, G_T and the corresponding graph. The LHS of the rule represents the graph having two states in which the transition between them will be added. The RHS of the rule shows

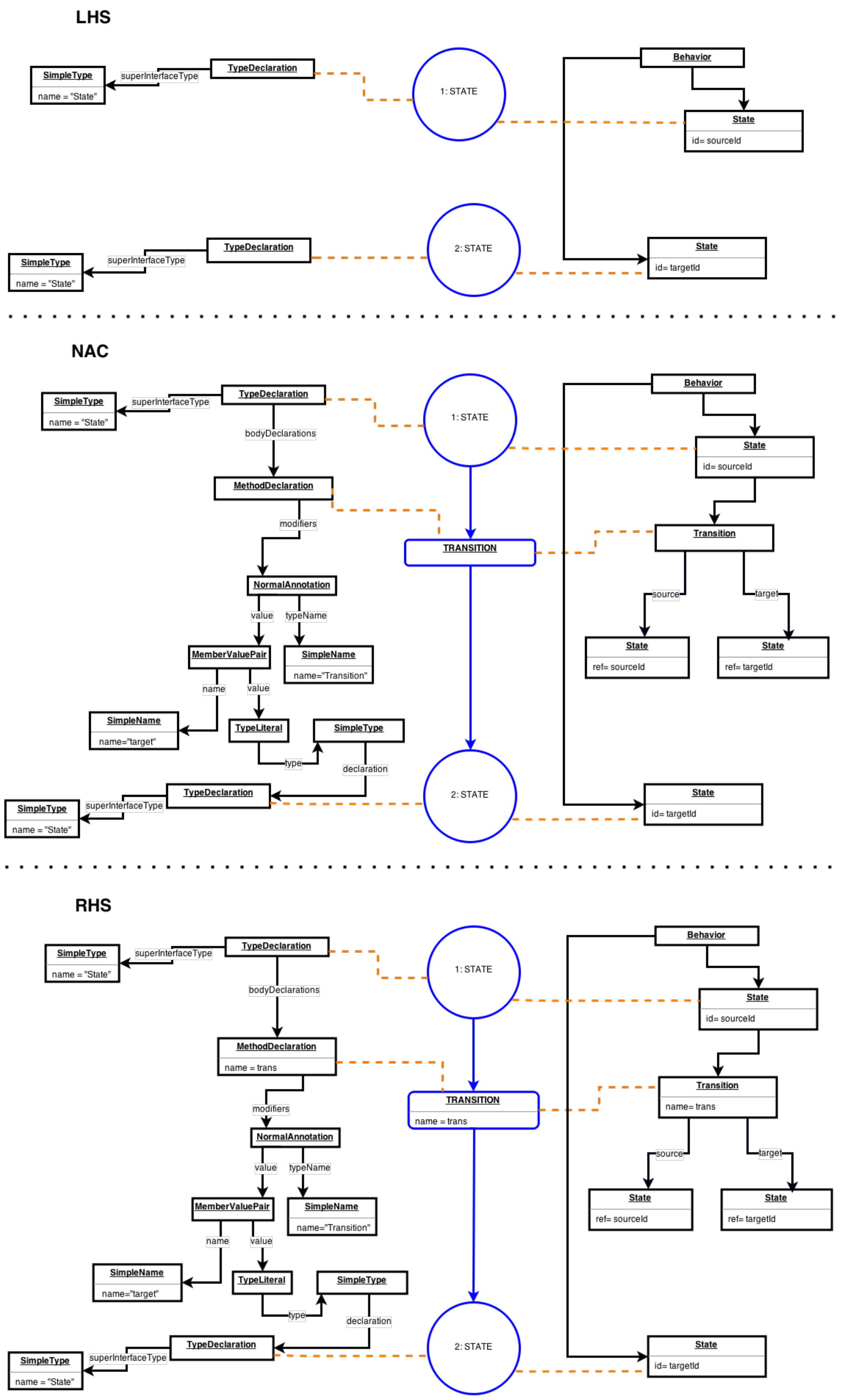


Figure 5.11: A rule for adding transition between two states

the complete graph after a new transition from one state to another state was added. This approach preserves the semantics of the state machine model, provides facility to support incremental updates and change propagation. The drawback of this is the complexity of the syntax graph representing the generated code that makes it hard to derive the complete set of rules in the complex scenario. In those situations, the approach of using unidirectional transformation by applying templating techniques for code generation can be a solution. This technique is useful since model navigation manipulation is performed by a templating language in an imperative way. This allows more logics to be easily applied to the construction and provides faster deliver time when dealing with complex syntax structures.

Overall, representing the page behavior using finite state machine in this case enables the possibility for this approach to be applied to a wide range of web applications and pages. With this level of model abstraction, we can perform both transformations based on TGG or on imperative methods. We can reach the appropriate compromise between generality and expressive power which enhances code reuse and better reaction to changes. This makes the integration with agile development easier.

5.6 Automation of test case generation with WTML

According to the Institute of Electrical and Electronics Engineers (IEEE) standards, a test case is "a documentation specifying inputs, predicted results, and a set of execution conditions for a test item" [68]. As the definition states, there are three required elements in a test case; test inputs, predicted results and conditions of execution. IEEE's definition also clarifies the difference between test data and test case.

In order to generate test cases, test data must first be generated. Test data can be generated using different sources that describe the system, system's behavior or how the system will be used, such as source code, system specifications, system requirements, business scenarios and use cases. Our approach utilizes specification-based method for test case generation.

In this approach, we focus on the verification of the web system against the design specification that was available on the test models. This comprises of abstract information on the available operations and its parameters. Information from the specifications allows generation of test cases for boundary-value analysis, equivalence class testing or random testing [59].

In WTML platform, in order to generate the tests, we first need to generate the model implementation of the page to be tested against. A sample on how the page in Java was generated is as bellow:

```
...
@Page
@ComponentScan(basePackages={"net.webmodeling.testing"})
public class FirstPage {
    private final static String baseUrl="http://www.testpage.org/";
    private final static String iterateParamName="value";

    @Autowired
    private AutoBrowser browser;

    @Value("#{'AAA,BBB,CCC'.split(',')}")
    private List<String> iterateParams;
    private static final By rating =
                            By.xpath("//*[@id='viewcomments_click']");

    public String getRating() {
        return browser.getTextValue(rating);
    }
    public void clickOnRating() {
        browser.clickOn(rating);
    }
```

...

From the web page model syntax as seen on previous section, *iterateParam* is used when we want to iterate over a set of input parameters when testing a page. This becomes handy especially on the development of regression tests.

Another important aspect is *@Page* annotation, we introduced this annotation to inject special configuration to a page. This allows us to use Spring framework for processing pre- and post- Java bean creation. Testing data is injected directly into the page from the test models by using Spring *@Value* annotation. All setters and getters are also automatically generated from elements in the models.

For the generation of test cases, we define the following algorithm to generate tests from the models:

1. *For all commands in behavior model do:*

   If command type is one of [TYPE, CLICK, GET, MOUSEMOVE, etc] that was pre-defined in our action type, generate a method to perform this action by calling utility methods from our implemented AutoBrowser model.

2. *For all events in behavior model do:*

   Generate a boolean method ***is<EventName>*** *to test the equality between the outcome of command in the event (by calling the method that was generated in the previous step) and the assertion string in that event.*

3. *Finally, for all states in behavior model do:*

   Generate a ***test<StateName>*** *method.*

   Within that method, execute all actions defined in the state.

   For all transitions in the state model, test the assertion for output of actions in the input event and the assertion string that defined for that event.

This algorithm generates the tests that were modeled using the transitions between different states of the state machine. At each level, all the assertions are performed in order to test the correctness of the behavior against test data.

This approach also provides a solution for automating regression testing. To support regression tests, we can reuse the existing test cases from the previous system tests. Regression testing is performed when additions or modifications are made to an existing system. Since this could be run and generated automatically, regression testing could be performed anytime using WTML platform when there is a requirement.

5.7 Integration with other platform

One of the essential features of the modeling tool is the ability to integrate with other platforms. Selenium is a suite of tools to automate web browsers across many environments. We design WTML such that it can utilize Selenium to provide automatic simulation with browsers. WTML raises the level of abstraction by modeling the elements and actions on the web page. This model will then be used as input to generate code for modeling page accordingly. We use Java as the target language. Using Spring framework dependency injection we then can integrate layered architecture in the code generated. Configurations are injected into JUnit tests via Spring annotation.

To support WTML platform, we created our defined annotations in Java, this *Page* annotation consists of Configuration that can be later injected and directives to load the application context. We also defined our browser implementation in order to integrate with web driver from Selenium and provide automatic processing. In general this browser is defined in the following way:

```
...
@Component
public class AutoBrowser {
    private static final int TIME_OUT_SEC = 10;
    private static final Logger LOGGER =  LoggerFactory.getLogger(AutoBrowser.class);

    @Autowired
    private WebDriver webDriver;

    public void clickOn(By location) {
        webDriver.findElement(location).click();
    }
    public WebElement findElement(By location) {
        return webDriver.findElement(location);
    }
    public void goToPage(String url) {
        webDriver.get(url);
    }
    public void goToUrlWithParam(String baseUrl, Map<String,String> params) {
        final StringBuilder pageUrl = new StringBuilder();
        pageUrl.append(baseUrl + "?");
        for (Map.Entry<String, String> entry : params.entrySet()) {
            pageUrl.append(entry.getKey());
            pageUrl.append("=");
            pageUrl.append(entry.getValue());
            pageUrl.append("&");
        }
        goToPage(pageUrl.toString());
    }
    public void goToUrlWithSingleParam(String baseUrl, String paramName,
                                       String paramValue) {
        final StringBuilder pageUrl = new StringBuilder();
```

```
        pageUrl.append(baseUrl + "?");
        pageUrl.append(paramName);
        pageUrl.append("=");
        pageUrl.append(paramValue);
        goToPage(pageUrl.toString());
    }

    @PreDestroy
    private void destroy() {
        webDriver.quit();
    }
    public int getNumberOfElements(By location) {
        return webDriver.findElements(location).size();
    }
    public String getTextValue(By location) {
        return webDriver.findElement(location).getText();
    }
    public String getAttributeValue(By location, String attributeName) {
        return webDriver.findElement(location).getAttribute(attributeName);
    }
    public String getCssValue(By location, String propertyName) {
        return webDriver.findElement(location).getCssValue(propertyName);
    }
...
```

After the configuration of Selenium web driver is defined and loaded, we inject web driver into our AutoBrowser, this way we keep the Selenium code separated from our browser logic. This allows us to only focus on the requirements and logics of code generation and automation test runners. Finally, we define all necessary methods for our automated browser such as *getNumberOfElements* from a given XPath address inside any page.

With the integration of Selenium, we are able to perform automatic browser actions. This makes it possible to write automated tests for a web application directly in WTML, which allows for better integration in existing unit test frameworks.

5.8 Related work

In the UML world, there have been effort on proposing techniques to automatically generate and execute test cases starting from a UML model of the Web Application by Ricca et al. [67]. This approach requires a manual work in several phases. There is manual work on the creation of models for testing and in the test refinement phase. Our approach has an advantage of fully automation in test case generation using the abstract web model and its action.

Cavarra et al. [12] presented the approach on test case generation utilizing UML. The authors' approach is based on extending UML using UML profiling capabilities. In these approaches, two profiles are created for different purposes. The first one is used to model the system under test by extending class diagrams, object diagrams, and state diagrams to support testing properties. The other profile is used to capture the test directives which are composed of the object diagrams and state diagrams. A transformation is then used to verify and produce scripts that can later be used to generate test cases.

Yuan et al. [79] present an automatic approach to generate test cases of a given business process of a web service. Business Process Execution Language and UML activity diagrams are used to define the process under test. The UML Testing Profile standard and the concepts from test control notations are used to construct the test case model. In this approach, the authors defined a framework for building concise test models that can be used to generate automatically test cases by applying the Model-Driven Architecture approach and possibly conformed transformation techniques. The generated test model can be tailored to target many test types such as unit testing, component testing, integration testing, or system testing.

A model-based testing approach is presented by Bouquet et al. in [10]. Their approach is based on a combination of class, object, and state diagrams which can be found in UML and OCL expressions to automatically generate test cases from these models. Test cases are generated using a test generator that takes these diagrams and constraints as input. The authors discuss the need to alter the semantics of OCL to allow OCL expressions to have a side effect on the system state. In an overview of model driven testing techniques from the work of Mussa et al. [58], the authors pointed out the shortcomings of this approach that it violates OCL semantics, which may hinder the acceptance of the approach by the UML community. One possible solution is to use an action language to express expressions that change the state of the system.

There has been also a direct attempt to use UML activity diagrams to generate test cases for Java programs in the work of Mingsong et al. [56]. The approach is based on the generation of test cases then compares the running traces with the activity diagram to reduce the test case set. The disadvantage of this approach is the limitation to the UML activity diagram that makes it impossible to obtain concurrency or loops for the tests.

Deutsch et al. [19] introduced an approach that models data-driven web applications. This approach used Abstract State Machine to model the transitions between pages, determined by the input provided to the application. The structure and contents of web pages, as well as the actions to be taken, are determined dynamically by querying the underlying database as well as the state and inputs. The properties to be verified concern the sequences of events (inputs, states, and actions) resulting from the interaction, and are expressed in linear or branching-time temporal logic. This approach has an advantage of wide-range error detection. However, this leads to complex models that can made the integration with development methodologies impossible.

5.9 Summary

With the fast-changing aspect of recent web-based systems, techniques to assure the quality of these systems play a very important role in the development process. In this chapter, we outlined the theoretical ideas and analysis from lessons learned during the real implementation of the web testing framework. The approach introduced in our research provides a methodology for using a domain specific language in model-driven testing of web applications. This is the solution to the challenge in reducing overall test development time by supporting the reuse of common testing components.

Adopting WTML in combination with the MDA initiative allows early testing of model-driven systems and eases the sharing of models between the system developers and the system testers. This provides a more complete methodology and framework for modeling and test generation in the model driven development of web application testing paradigm.

WTML was designed at the appropriate abstraction level to provide better model readability. This is aimed at reducing test maintenance costs, since changes happen at the model level and are captured by the test models. When there are changes, we only have to regenerate the tests from the test models and all test cases are updated to the new specifications. This framework also enhances team communication because the model, test cases provide a clear, unambiguous, and unified view of both the system under test and the test. This technique decouples the testing logic from the actual test implementation. This makes the test architecture more robust and scalable. The shortcomings of this approach include a learning curve needed to adopt a new modeling language and the limitation of test behaviors based only on the possible elements modeled in a page abstraction.

Domain specific language such as WTML can be applied to automation testing of web-based applications and pages. In practice, this approach has initially gained adoption in testing of large web systems supporting financial metrics data in the financial industry where the author had the chance to work with.

Chapter 6

Conclusions

The overall goal of the research described in this dissertation is to provide solutions to the challenges in model driven development of software systems in recent emerging domain. The key contributions include:

1. We analyzed current approaches to model transformation, provided an overview of the approaches and their use cases in practice. We outlined the common weaknesses and advantages of these approaches. This information is useful for the decision-making when dealing with model transformation requirements. We made an attempt to build a guideline to agile development of PIMs in MDD with a case study. The work of the author on agile development of PIMs has been cited as references in US Patent No. 8516435 and No. 8495559.

2. In response to the challenges in recent emerging fields, we developed a framework for development of web services using domain specific language in the MDD paradigm. We introduced SWSM - a DSL for modeling web services that support automation with code generation. This technique addresses the problem in web service development of software-as-service systems that often require the support for tenant-specific architecture. This is a novel approach that is a solution to the challenge in applying model driven development within the web domain.

3. We presented an approach for model driven testing of web applications that focuses on automation and regression testing. We implemented and introduced WTML as a language for web pages modeling and automatic test cases generations. Our techniques are based on building abstractions of web pages and modeling state-machine-based test behavior. This approach is the answer to the recent challenges in the quality assurance of the fast-changing web systems.

The practical benefit of the research is to reduce human effort and potential errors by introducing techniques and approaches in order to automate the model driven development process in a specific domain. This is done by providing better software reuse, higher development speed and better cost-effectiveness via the utilization of the DSL for modeling and code generation. The application in the finance industry, where the author has a chance to work with, have demonstrated the objectives of the approach to reduce the time-to-market and maintain high standards of the application by identifying in advance possible faults with automated test case generation and execution.

To conclude, as more cloud based and software-as-service systems gain adoption recently, these systems require different approach to web service development and deployment with a more tenant-specific architecture. Current methods often focus on the usage of a generic modeling language such as UML, which leads to complex class diagrams and obstacle in achieving automation. The presented techniques address the challenge in the development of such systems by providing appropriate abstraction level to separate the logical models from its technical aspects. The contribution of the thesis is the answer to the lack of concrete methods and toolset in applying model driven development to specific areas such as web application testing and services. As members of our team are working with companies in the top global Fortune 500 dealing with large-scale web services for financial services, the outlined DSL approach has started to gain adoption and initially has been applied successfully.

The future work could be the continuation on objective improvements and further practical appliance of such techniques in other fields such as mobile model driven development and every day model driven development processes.

Chapter 7

Relevant publications of the author

Publications in impacted Journals	Author's contribution (%)
V. Nguyen, X. Qafmolla, K. Richta. Domain Specific Language Approach on Model-driven Development of Web Services, Journal of applied science, *Acta Polytechnica Hungarica*. Vol. 11, No. 8, 2014. ISSN 1785-8860.	80

Publications in reviewed Journals	Author's contribution (%)
V. Nguyen. Model Driven Testing of Web Applications using Domain Specific Language, *International Journal of Advanced Computer Science and Applications*, Vol. 6, No. 1, 2015. ISSN 2156-5570.	100
V. Nguyen, X. Qafmolla. Model Transformation in Web Engineering and Automated Model Driven Development. In: *International Journal of Modeling and Optimization.* Vol. 1, no. 1, 2011, p. 7-13. ISSN 2010-3697. The paper has been cited in: ▶ Rahmouni, M'hamed, and Samir Mbarki. MDA-Based ATL Transformation to Generate MVC 2 Web Models, International Journal of Computer Science and Information Technology, Vol. 3, No. 4, 2011, ISSN 0975-3826. ▶ Arvidsson, Daniel, and Fredrik Persson. Evaluating a Design Pattern for Black-Box Testing in Executable UML, Technical Report, Department of Computer Science and Engineering, University of Gothenburg, 2012.	60

Publications indexed by ISI/Scopus	Author's contribution (%)
V. Nguyen, X. Qafmolla. On Instance-model Querying and Meta-model Transformation. In: *Proceedings of the International Conference on Software Engineering.* Hong Kong: The International Association of Engineers IAENG, 2010, p. 710-715. ISSN 2078-0958. ISBN 978-988-17012-8-2.	77
X. Qafmolla,V. Nguyen. Automation of Web Services Development Using Model Driven Techniques. In: *Proceedings of The 2nd International Conference on Computer and Automation Engineering (ICCAE 2010).* Singapore: Institute of Electronics Engineers, Inc., 2010, p. 190-194. ISBN 978-1-4244-5585-0. The paper has been cited in: ▶ Simon, Balazs, Balazs Goldschmidt, and Karoly Kondorosi. "A Metamodel for the Web Services Standards." *Journal of grid computing,* Vol. 11, no. 4 (2013): 735-752. ▶ Chih-Min, L. O., and Sun-Jen Huang. "Applying Model-Driven Approach to Building Rapid Distributed Data Services." *IEICE TRANSACTIONS on Information and Systems,* Vol. 95, no. 12 (2012): 2796-2809. ▶ Mukhtar, Mohammed Abdalla Osman, A. Azween, and Alan G. Downe. "A Proposed Compiler to Integrate Model Driven Architecture with Web Services–Road Map." *International Journal of Computer Applications*, Vol. 15, no. 7 (2011): 1-7. ▶ Zohrevand, Z.; Bibalan, Y.M.; Ramsin, R. "Towards a Framework for the Application of Model-Driven Development in Situational Method Engineering", *Software Engineering Conference (APSEC)*, 2011 18th Asia Pacific, pp. 122 - 129. ISSN 1530-1362.	50

► Mukhtar, M.A.O.; Hassan, M.F.B.; Bin Jaafar, J. “Optimizing method to provide model transformation of model-driven architecture as web-based services”, *Computer & Information Science (ICCIS)*, 2012 International Conference on, On page(s): 874 - 879 Volume: 2, 12-14 June 2012 ► Tavares, Nírondes AC, and Samyr Vale. “A Model Driven Approach for the Development of Semantic RESTful Web Services.” In *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, p. 290. ACM, 2013. ► Zohrevand, Zahra, Yusef Mehrdad Bibalan, and Raman Ramsin. “Towards a framework for the application of model-driven development in situational method engineering.” *In Software Engineering Conference (APSEC)*, 2011 18th Asia Pacific, pp. 122-129. IEEE, 2011.	

Other publications	Author's contribution (%)
V. Nguyen, X. Qafmolla. Towards automated model driven development with model transformation and domain specific languages, In: *Proceedings of the International Conference on Computer and Software Engineering*, Kuala Lumpur, 2012, pp. 128-134. ISBN 978-93-82242-22-2.	50
V. Nguyen, X. Qafmolla. On Model Transformation Methods and Testing of Model Transformation to Support Automated Model Driven Development. In: *Proceedings of the 3rd International Conference on Computer and Automation Engineering.* Singapore: Institute of Electronics Engineers, Inc., 2011, vol. 1, p. 171-175. ISBN 978-1-4244-9462-0.	70
V. Nguyen, X. Qafmolla. Agile Development of Platform Independent Model in Model Driven Architecture. In: *Proceedings of the Third International Conference on Information and Computing.* IEEE, 2010, p. 344-347. ISBN 978-1-4244-7081-5. The paper has been cited in: ▶ Matinnejad, R. "Agile Model Driven Development: An Intelligent Compromise", *Software Engineering Research, Management and Applications (SERA)*, 2011 9th International Conference on, pp. 197 - 202. ISBN 978-1-4577-1028-5. ▶ Botturi, G.; Ebeid, E.; Fummi, F.; Quaglia, D. "Model-driven design for the development of multi-platform smartphone applications", *Specification & Design Languages (FDL)*, Forum on, pp. 1 - 8, 2013. ISSN 1636-9874.	50

Cited by US Patents: ▶ Akkiraju, Rama Kalyani T. ; Bhandar, Manisha Dattatraya ; Dhoolia, Pankaj ; Fu, Shiwa ; Ghosh, Nilay ; Mitra, Tilak ; Mohan, Rakesh ; Nigam, Anil ; Saha, Dipankar ; Zhao, Wei , "System and method for generating implementation artifacts for contextually-aware business applications", Patent No. 8516435. ▶ Akkiraju, Rama Kalyani T. ; Mitra, Tilak ; Thulasiram, Usha , "Extracting platform independent models from composite applications", Patent No. 8495559.	
X. Qafmolla, V. Nguyen. Model-driven Practices in Web Engineering. In: *Objekty 2009.* Hradec Králové: Univerzita Hradec Králové, 2009, vol. 1, p. 185-195. ISBN 978-80-7435-009-2.	50
V. Nguyen, X. Qafmolla. Metamodel Transformation with Kermeta. In: *Objekty 2008.* Žilina: Žilinská univerzita v Žiline, Fakulta riadenia a informatiky, 2008, p. 109-116. ISBN 978-80-8070-927-3.	50

Less relevant publications

Publications in reviewed journals	Author's contribution (%)
X. Qafmolla, V. Nguyen, K. Richta. Metamodel-based Generation of Web Content Management Systems. *International Journal on Information Technologies and Security*. 2014, 2014(4), 17-30. ISSN 1313-8251	10
Other publications	
X. Qafmolla, V. Nguyen. Semiautomatic Generation of Content-based Web Applications: An Evaluation of Supporting Tools. Assoc. Prof. Dr. Yurdagül Ünal, Hacettepe University, Department of Information Management.	20
V. Nguyen, X. Qafmolla. A Practical Approach on Implementing Scrum in Mid-size Companies. In: *Objekty 2009*. Hradec Králové: Univerzita Hradec Králové, 2009, vol. 1, p. 293-304. ISBN 978-80-7435-009-2.	50
X. Qafmolla, V. Nguyen. A Two-Way Meta-Modeling Approach in Web Engineering. In: *Procedia Information Technology and Computer Science*. Istanbul, Bahçeşehir Üniversitesi, 2012, ISSN 2147-5105.	50

Bibliography

[1] Aditya Agrawal, Gabor Karsai, and Feng Shi. Graph transformations on domain-specific models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, 2003.

[2] D. H. Akehurst and B. Bordbar. On querying UML data models with OCL. In *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference*, pages 91–103. Springer, 2001.

[3] Gil Allouche. Top 5 latest trends in cloud computing, 2013. URL `http://www.itproportal.com/2013/09/20/top-5-latest-trends-in-cloud-computing/`. [Accessed Jan. 23, 2014].

[4] Daniel Amyot, Hanna Farah, and Jean-Francois Roy. Evaluation of development tools for domain-specific modeling languages. In Reinhard Gotzhein and Rick Reed, editors, *System Analysis and Modeling: Language Profiles*, volume 4320 of *Lecture Notes in Computer Science*, pages 183–197. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-68371-1.

[5] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *In Proc. of the First International Conference on Graph Transformation (ICGT 2002*, pages 402–429. Springer, 2002.

[6] Peter Bell. Automated transformation of statements within evolving domain specific

languages. In *Proceedings of The 7th OOPSLA Workshop on Domain-Specific Modeling*, OOPSLA '07, pages 172–177. ACM, 2007.

[7] Gorka Benguria, Xabier Larrucea, Brian Elveseter, Tor Neple, Anthony Beardsmore, and Michael Friess. A platform independent model for service oriented architectures. In Guy Doumeingts, Jorg Muller, Gerard Morel, and Bruno Vallespir, editors, *Enterprise Interoperability*, pages 23–32. Springer London, 2007. ISBN 978-1-84628-713-8.

[8] J. Bezivin, S. Hammoudi, D. Lopes, and F. Jouault. Applying mda approach for web service platform. In *Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International*, pages 58–70, 2004. doi: 10.1109/EDOC.2004.1342505.

[9] Behzad Bordbar and Athanasios Staikopoulos. Automated generation of metamodels for web service languages. *Computer Science at Kent*, page 203, 2004.

[10] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, and Fabien Peureux. A test generation solution to automate software testing. In *Proceedings of the 3rd International Workshop on Automation of Software Test*, AST '08, pages 45–48, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-030-2.

[11] Benjamin Braatz and Christoph Brandt. A framework for families of domain-specific modelling languages. *Software & Systems Modeling*, 13(1):109–132, 2014. ISSN 1619-1366.

[12] Alessandra Cavarra, Charles Crichton, Jim Davies, Alan Hartman, and Laurent Mounier. Using uml for automatic test generation. In *In international symposium on testing and analysis ISSTA*. Springer-Verlag, 2002.

[13] Alistair Cockburn. *Agile Software Development.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0-201-69969-9.

[14] World Wide Web Consortium. Web services architecture, 2009. URL `http://www.w3.org/TR/ws-arch`. [Accessed Feb. 22, 2014].

[15] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, July 2006. ISSN 0018-8670.

[16] J. De Bruijn and H. (Ed.) Lausen. The web service modeling language WSML, 2012. URL `http://www.w3.org/Submission/WSML/`. [Accessed Feb. 20, 2014].

[17] May Dehayni, Kablan Barbar, Ali Awada, and Mohamad Smaili. Some model transformation approaches: a qualitative critical review. *Journal of Applied Sciences Research*, 5(11):1957–1965, 2009. URL `http://www.insipub.com/jasr/2009/1957-1965.pdf`.

[18] Arie Van Deursen and Paul Klint. Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998. ISSN 1096-908X. doi: 10.1002/(SICI)1096-908X(199803/04)10:2<75::AID-SMR168>3.0.CO;2-5.

[19] Alin Deutsch, Liying Sui, and Victor Vianu. Specification and verification of data-driven web applications. *Journal of Computer and System Sciences*, 73(3):442 – 474, 2007. ISSN 0022-0000. Special Issue: Database Theory 2004.

[20] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997. ISBN 01321-5871X.

[21] Schmidt Douglas C. Model-driven engineering. *IEEE Computer*, 39(2):25–32, 2006.

[22] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann, and Gabriele Taentzer. Information preserving bidirectional model transformations. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, FASE'07, pages 72–86, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71288-6.

[23] Hartmut Ehrig, Claudia Ermel, and Frank Hermann. On the relationship of model transformations based on triple and plain graph grammars. In *Proceedings of the Third*

International Workshop on Graph and Model Transformations, GRaMoT '08, pages 9–16, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-033-3.

[24] Hartmut Ehrig, Claudia Ermel, Frank Hermann, and Ulrike Prange. On-the-fly construction, correctness and completeness of model transformations based on triple graph grammars. In Andy Schurr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 241–255. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-04424-3.

[25] Dieter Fensel and Christoph Bussler. The web service modeling framework {WSMF}. *Electronic Commerce Research and Applications*, 1(2):113 – 137, 2002. ISSN 1567-4223. doi: http://dx.doi.org/10.1016/S1567-4223(02)00015-7.

[26] David Frankel and John Parodi. Using model-driven architecture to develop web services. 2012.

[27] U. Golas, H. Ehrig, and F. Hermann. Formal specification of model transformations by triple graph grammars with application conditions. *ECEASST*, 39:1–26, 2011. URL `http://journal.ub.tu-berlin.de/eceasst/article/view/646`.

[28] Object Management Group. Object constraint language specification v1.1, 1997. URL `http://www.omg.org/cgi-bin/doc?ad/97-08-08`. [Accessed Jan. 5, 2014].

[29] Object Management Group. OpenQVT revised submission to the MOF 2.0 Q/V/T RFP, 2003. URL `http://www.omg.org/cgi-bin/doc?ad/2003-08-05`. [Accessed Dec. 12, 2013].

[30] Object Management Group. Common warehouse metamodel, 2009. URL `http://www.omg.org/spec/CWM/1.1/`. [Accessed Jan. 15, 2014].

[31] Object Management Group. OMG model driven architecture, 2009. URL `www.omg.org/mda`. [Accessed Jun. 12, 2014].

[32] Object Management Group. The architecture of choice for a changing world, 2012. URL `http://www.omg.org/mda`. [Accessed Feb. 15, 2014].

[33] Object Management Group. Meta object facility specification, 2013. URL `http://www.omg.org/spec/MOF/2.0/`. [Accessed Feb. 5, 2014].

[34] Johan Den Haan. The enterprise architect - building an agile enterprise, 2013. URL `http://www.theenterprisearchitect.eu`. [Accessed Feb. 20, 2014].

[35] Reiko Heckel, Jochen Malte Kuster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *Proc. ICGT 2002. Volume 2505 of LNCS*, pages 161–176. Springer, 2002.

[36] Reiko Heckel, Marc Lohmann, and Sebastian Thöne. Towards a uml profile for service-oriented architectures. In *Model Driven Architecture: Foundations and Applications*, page 115. Citeseer, 2003.

[37] Markus Herrmannsdoerfer, Daniel Ratiu, and Guido Wachsmuth. Language evolution in practice: The history of gmf. In *Proceedings of the Second International Conference on Software Language Engineering*, SLE'09, pages 3–22, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-12106-3, 978-3-642-12106-7.

[38] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Q.*, 28(1):75–105, March 2004. ISSN 0276-7783.

[39] Gregor Kandare, Stanislav Strmčnik, and Giovanni Godena. Domain specific model-based development of software for programmable logic controllers. *Comput. Ind.*, 61(5): 419–431, June 2010. ISSN 0166-3615.

[40] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. Evaluating the use of domain-specific modeling in practice. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*, 2009.

[41] D. Kaul, A. Kogekar, A. Gokhale, J. Gray, and S. Gokhale. Posaml: A visual modeling framework for middleware provisioning. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 289c–289c, 2007.

[42] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical report, Software Engineering Group, Department of Computer Science, University of Paderborn, 2007.

[43] Tomaz Kos, Tomaz Kosar, and Marjan Mernik. Development of data acquisition systems by using a domain-specific modeling language. *Computers in Industry*, 63(3):181 – 192, 2012. ISSN 0166-3615.

[44] Tomaz Kosar, Pablo E. Martinez Lopez, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390 – 405, 2008. ISSN 0950-5849.

[45] Sabine Kuske, Paul Ziemann, and Martin Gogolla. Towards an integrated graph based semantics for UML. *Proc. Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2002)*, 2003.

[46] J.M. Kuster, R. Heckel, and G. Engels. Defining and validating transformations of UML models. In *2003 IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings*, pages 145–152, 2003. doi: 10.1109/HCC.2003.1260218.

[47] H. Lausen, A. Polleres, and D.(Ed.) Roman. Web service modeling ontology (wsmo), 2005. URL `http://www.w3.org/Submission/WSMO/`. [Accessed Feb. 23, 2014].

[48] Tihamer Levendovszky, Gabor Karsai, Miklos Maroti, Akos Ledeczi, and Hassan Charaf. Model reuse with metamodel-based transformations. In *of Lecture Notes in Computer Science*, pages 166–178. Springer, 2002.

[49] YueHua Lin. *A model transformation approach to automated model evolution*. University of Aslabama at Birmingham, 2007.

[50] Earl Long, Amit Misra, and Janos Sztipanovits. Increasing productivity at saturn. *Computer*, 31(8):35–43, August 1998. ISSN 0018-9162.

[51] Janne Luoma, Steven Kelly, and Juha-Pekka Tolvanen. Defining domain-specific modeling languages: Collected experiences. In *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling*, 2004.

[52] Luis Mandel and Maria Victoria Cengarle. On the expressive power of OCL. pages 854–874, 1999.

[53] E Michael Maximilien, Hernan Wilkinson, Nirmit Desai, and Stefan Tai. A domain-specific language for web apis and services mashups. In *Service-oriented computing–ICSOC 2007*, pages 13–26. Springer, 2007.

[54] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, March 2006. ISSN 1571-0661.

[55] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[56] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. Automatic test case generation for uml activity diagrams. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, pages 2–8, New York, NY, USA, 2006. ACM. ISBN 1-59593-408-1.

[57] Web modeling Group. Swsm language, 2013. URL `http://webmodeling.net/documentation.html`. [Accessed Dec. 21, 2013].

[58] M. Mussa, S. Ouchani, W. Al Sammane, and A. Hamou-Lhadj. A survey of model-driven testing techniques. In *Quality Software, 2009. QSIC '09. 9th International Conference on*, pages 167–172, Aug 2009.

[59] Mark Harman Mustafa Bozkurt and Youssef Hassoun. Testing web services: A survey. Technical Report TR-10-01, Department of Computer Science, King's College London, January 2010.

[60] Universite De Nantes, Groupe Sodifrance, Jean Bezivin, Erwan Breton, Gregoire Dupe, and Patrick Valduriez. The ATL transformation-based model. In *Management Framework, Research Report, Atlas Group, INRIA and IRIN*, 2003.

[61] Viet Cuong Nguyen and Xhevi Qafmolla. Metamodel transformation with kermeta. In *Objekty 2008*, volume 1, pages 109–116, 2008.

[62] Ernst Oberortner, Uwe Zdun, and Schahram Dustdar. Domain-specific languages for service-oriented architectures: An explorative study. In *Towards a Service-based Internet*, pages 159–170. Springer, 2008.

[63] Ken Peffers, Tuure Tuunanen, Marcus Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *J. Manage. Inf. Syst.*, 24(3):45–77, 2007. ISSN 0742-1222.

[64] Xhevi Qafmolla and Viet Cuong Nguyen. Automation of web services development using model driven techniques. In *Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on*, volume 3, pages 190–194. IEEE, 2010.

[65] Anna Queralt and Ernest Teniente. A platform independent model for the electronic marketplace domain. *Software & Systems Modeling*, 7(2):219–235, 2008. ISSN 1619-1366.

[66] Arend Rensink and Eduardo Zambon. A type graph model for java programs. In *Formal Techniques for Distributed Systems*, volume 5522 of *Lecture Notes in Computer Science*, pages 237–242. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02137-4.

[67] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1050-7.

[68] IEEE Computer Society. IEEE standard for software test documentation. Technical report, 2008.

[69] J. Sprinkle, M. Mernik, J. Tolvanen, and D. Spinellis. Guest editors' introduction: What kinds of nails need a domain-specific hammer? *Software, IEEE*, 26(4):15–18, July 2009. ISSN 0740-7459.

[70] Jonathan Sprinkle and Gabor Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15:3–4, 2004.

[71] Michael Striewe. Using a triple graph grammar for state machine implementations. In *Graph Transformations*, volume 5214 of *Lecture Notes in Computer Science*, pages 514–516. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-87404-1.

[72] Laurence Tratt. Model transformations and tool integration. *Software & Systems Modeling*, 4(2):112–122, 2005. ISSN 1619-1366.

[73] Girba Tudor. The moose book, 2013. URL `http://www.themoosebook.org/`. [Accessed Feb. 2, 2014].

[74] Wikipedia. Model driven engineering, 2011. URL `http://en.wikipedia.org/wiki/Model-Driven-Engineering`. [Accessed Jun. 12, 2014].

[75] Wikipedia. Domain-specific language, 2012. URL `http://en.wikipedia.org/wiki/Domain-specific-language`. [Accessed Jul. 3, 2014].

[76] Wikipedia. Model driven testing, 2014. URL `http://en.wikipedia.org/wiki/Model-based-testing`. [Accessed Jul. 4, 2014].

[77] David Wile. Lessons learned from real dsl experiments. *Sci. Comput. Program.*, 51 (3):265–290, June 2004. ISSN 0167-6423. doi: 10.1016/j.scico.2003.12.006. URL `http://dx.doi.org/10.1016/j.scico.2003.12.006`.

[78] Xiaofeng Yu, Yan Zhang, Tian Zhang, Linzhang Wang, Jun Hu, Jianhua Zhao, and Xuandong Li. A model-driven development framework for enterprise web services. *Information Systems Frontiers*, 9(4):391–409, 2007. ISSN 1541-7719.

[79] Qiulu Yuan, Ji Wu, Chao Liu, and Li Zhang. A model driven approach toward business process test case generation. In *Proc. of the 10th International Symposium on Web Site Evolution (WSE)*, pages 41–44, 2012.

Anotace

Jednou ze současných výzev softwarového vývoje je přizpůsobení softwarového systému měnícím se požadavkům a nárokům uživatele a změnám prostředí. Konečným cílem je vnořit tyto požadavky do vysokoúrovňové abstrakce, což umožňuje dosáhnout přizpůsobení základní implementace softwaru ve velkém rozsahu. Modelem řízené inženýrství (Model-Driven Engineering, MDE) je jednou z klíčových technik, která podporuje tento záměr. Efektivní vytváření modelů a jejich transformace jsou zde hlavními činnostmi, které umožňují převod zdrojových modelů na cílové za účelem změny modelové struktury nebo převodu modelů na jiné softwarové produkty. Naším hlavním cílem je umožnit automatizaci a automatizovaný vývoj systému z odpovídajících modelů. I když již existuje několik vysokoúrovňových přístupů, je zde stále absence jasné metodiky a výsledků pro aplikaci techniky MDE na specifickou oblast se specifickými požadavky, jako je oblast webových aplikací. Tento výzkum se snaží přispět k řešení problému automatizace vývoje modelů tím, že poskytuje přehled existujících přístupů s několika případovými studiemi a zavádí nový přístup v rozvíjející se oblasti webových aplikací a služeb. Abychom se mohli vypořádat se současným trendem rostoucí složitosti webových služeb, jakožto programových páteří moderních distribuovaných a cloudových architektur, navrhujeme přístup s použitím doménově specifického jazyka pro modelování webových služeb jako řešení současné výzvy ve škálovatelnosti modelování a vývoje webových služeb. Analyzujeme současný stav problému a implementujeme doménově specifický jazyk, nazvaný Simple Web Service Modeling, pro podporu automatizovaného, modelem řízeného vývoje webových služeb. Tento přístup je řešením pro problémy ve vývoji systémů typu "software-as-a-service", které vyžadují podporu pro tuto specifickou architekturu.

V oblasti zajišťování kvality webových aplikací budujeme modelovací jazyk pro modelem řízené testování webových aplikací, který se zaměřuje na automatizaci a regresní testování. Naše techniky jsou založeny na budování abstrakce webových stránek a modelování chování testů založené na konečném automatu s použitím námi vyvinutého doménově specifického jazyka pro modelování webových stránek Web Testing Modeling Language. Tato metodika a techniky pomáhají softwarovým vývojářům i testerům zvýšit produktivitu a zkrátit dobu uvedení na trh při zachování vysokých standardů webových aplikací. Navrhované techniky jsou odpovědí na nedostatek konkrétních metod a sad nástrojů pro aplikaci modelem řízeného vývoje na specifické oblasti, jako jsou testování webových aplikací a webové služby. Výsledky této práce mohou být aplikovány na praktické problémy s metodickou podporou pro integrování do stávajících postupů softwarového vývoje.