GRAPHICAL EDITORS GENERATION WITH THE GRAPHICAL MODELING FRAMEWORK: A CASE STUDY

by

Miloud Salem S. Eloumri

A thesis submitted to the School of Computing

In conformity with the requirements for

the degree of Master of Science

Queen’s University

Kingston, Ontario, Canada

(April, 2011)

Copyright ©Miloud Salem S. Eloumri, 2011
Abstract

Domain Specific Modeling (DSM) aims to increase productivity of software development by raising the level of abstraction beyond code concepts and using domain concepts. By providing a generative model-driven tooling component and runtime support, the Eclipse Graphical Modeling Framework (GMF) aims to simplify the creation of diagram editors for specific domains based on a series of model creation and transformation steps. GMF leverages the Eclipse Modeling Framework (EMF) and the Eclipse Graphical Editing Framework (GEF) to allow the graphical modeling of Domain Specific Languages (DSL).

A Domain Specific Language (DSL) is developed specifically for a specific task and specific domain. In this research, the State Machine Compiler (SMC) represents the specific domain for which a DSL in a form of a diagram editor is developed using GMF. SMC is an open source Java tool allowing generation of state pattern classes from textual descriptions of state machines.

The main objective of this research is to describe the use of GMF, highlight potential pitfalls and identify strengths and weaknesses of GMF based on certain criteria. To be able to feed the SMC diagrams created with the editor into SMC, a Java Emitter Templates (JET) transformation is used to transform SMC model instances into textual format expected by SMC.
Acknowledgements

I would like to express my sincere thanks and immense gratitude to my supervisor, Dr. Juergen Dingel, for giving me the opportunity to work with him and for providing invaluable guidance, feedback and encouragement without which, this work would not have been possible.

Dr. Juergen Dingel, it has been an honor for me to work under your supervision and I am very delighted to know such caring and understanding advisor.

I wish also to deeply think The Ministry of Education and Scientific Research, Libya, for providing the means of financial support and giving me the opportunity to pursue my graduate studies.

Special thanks due to all my friends who always have been there ready to provide assistance in any respect. In particular, I would like to extend my thanks to my friends: Otman Ali Mohamed, Fathi Mohamed El-Harabi, Khaled Omar Eshtiwi and Ahmed Ali Egmila for their limitless support, both physically and emotionally.

Most importantly, I would like to show my love and appreciation to my parents and entire family for their unlimited love, support and understanding while being abroad throughout my studies.
# Table of Contents

Abstract ................................................................................................................................. ii  
Acknowledgements ............................................................................................................... iii  
Table of Contents ................................................................................................................ iv  
List of Figures ......................................................................................................................... viii  
List of Tables ......................................................................................................................... xi  

## Chapter 1 Introduction ........................................................................................................ 1  
1.1 Overview ........................................................................................................................ 1  
1.2 Motivation and Contribution ......................................................................................... 4  
1.3 Outline of the Document ............................................................................................... 5  

## Chapter 2 Background ........................................................................................................ 7  
2.1 Eclipse Modeling Framework (EMF) ........................................................................... 7  
  2.1.1 Eclipse Modeling Framework (EMF) and Object Management Group (OMG) ....... 7  
    2.1.1.1 Unified Modeling Language (UML) ................................................................. 8  
    2.1.1.2 Meta Object Facility (MOF) ........................................................................... 8  
    2.1.1.3 XML Metadata Interchange (XMI) ................................................................. 8  
    2.1.1.4 Object Constraint Language (OCL) ............................................................ 8  
  2.1.2 EMF Ecore Model .................................................................................................... 9  
  2.1.3 Creating Ecore Based Models ............................................................................... 10  
  2.1.4 EMF Development Flow ....................................................................................... 11  
  2.1.5 EMF Generator Model ........................................................................................ 11  
  2.2 Graphical Editing Framework (GEF) ......................................................................... 12  
  2.3 Graphical Modeling Framework (GMF) ................................................................... 12  
    2.3.1 GMF Main Components ..................................................................................... 13  
      2.3.1.1 GMF Runtime ............................................................................................ 13  
      2.3.1.2 GMF Tooling ............................................................................................ 14  
    2.3.2 GMF Domain Model (EMF’s Ecore Model) ....................................................... 14  
    2.3.3 EMF Generator Model ...................................................................................... 15  
    2.3.4 GMF Graphical Definition Model ................................................................... 15  
    2.3.5 GMF Tooling Definition Model ....................................................................... 15  
    2.3.6 GMF Mapping Model ....................................................................................... 16  
    2.3.7 GMF Generator Model ..................................................................................... 17
2.4 Java Emitter Templates (JET) ............................................................................................. 17
2.5 Domain Specific Modeling (DSM) Tools ........................................................................ 18
2.6 Related Work ................................................................................................................... 19

Chapter 3 State Machine Compiler (SMC): A Case Study .................................................. 22
3.1 Using SMC ........................................................................................................................ 22
3.2 SMC Source Code ............................................................................................................. 23
   3.2.1 Verbatim Section ......................................................................................................... 24
   3.2.2 Setting Section ............................................................................................................ 24
   3.2.3 Map Section ............................................................................................................... 25
3.3 SMC Default State ............................................................................................................ 25
3.4 SMC Transition Types ....................................................................................................... 26
   3.4.1 Transition Actions ...................................................................................................... 27
   3.4.2 Transition Guards ...................................................................................................... 27
   3.4.3 Transition Precedence ............................................................................................... 28
3.5 SMC State Machine Example .......................................................................................... 28

Chapter 4 Diagram Editors Design Requirements ............................................................... 32
4.1 GMF Diagram Design Decisions ..................................................................................... 32
4.2 SMC Diagram Editor Requirements ................................................................................. 32

Chapter 5 Developing SMC EMF Models .......................................................................... 35
5.1 Creating SMC Domain Model (EMF’s Ecore Model) ...................................................... 35
   5.1.1 SMC Domain Model Structure ............................................................................... 38
5.2 Creating SMC EMF Generator Model ............................................................................ 39
5.3 Generating SMC EMF Code ............................................................................................ 41
5.4 Updating SMC EMF-generated plug-ins and properties files ......................................... 41

Chapter 6 Developing SMC GMF Models ........................................................................... 44
6.1 Developing SMC Graphical Definition Model ............................................................... 45
   6.1.1 Updating the SMC Graphical Definition Model ....................................................... 47
   6.1.2 Adding Compartments to the SMC Graphical Definition Model ............................ 53
6.2 Developing SMC Tooling Definition Model .................................................................. 54
   6.2.1 Updating the SMC Tooling Definition Model .......................................................... 56
6.3 Developing SMC Mapping Model ................................................................. 58
  6.3.1 Updating the SMC Mapping Model ....................................................... 59
  6.3.2 Updating MainMap Top Node Reference .............................................. 60
  6.3.3 Adding State Node Mapping and DefaultState Node Mapping to the MainMap Node Mapping ................................................................. 62
  6.3.4 Updating Map Top Node Reference ...................................................... 62
  6.3.5 Updating Transition Link Mapping ....................................................... 63
  6.3.6 Adding Link Mappings for All SMC Transition Types ......................... 66
  6.4 Audit Rules ............................................................................................... 68
    6.4.1 Adding Audit Rules to the SMC Mapping Model ............................. 68
  6.5 Developing SMC GMF Generator Model .................................................. 72
    6.5.1 Updating SMC GMF Generator Model ......................................... 72
  6.6 Generating SMC Diagram Code ............................................................... 74

Chapter 7 SMC Diagram Editor Validation ....................................................... 76
  7.1 Validation Examples .................................................................................. 76

Chapter 8 Developing SMC JET Transformation ............................................. 80
  8.1 SMC JET Transformation Overview ........................................................ 80
  8.2 Creating JET Transformation Project ....................................................... 81
  8.3 Configuring SMC JET Transformation ..................................................... 84
  8.4 JET Tags examples .................................................................................... 85

Chapter 9 Discussion ....................................................................................... 93
  9.1 SMC modeling and transformation tool .................................................. 93
  9.2 GMF Weaknesses .................................................................................... 94
    9.2.1 Documentation .................................................................................. 94
    9.2.2 Usability .......................................................................................... 94
    9.2.3 Updating ......................................................................................... 98
    9.2.4 Validation ....................................................................................... 98
    9.2.5 Automation and Generation ............................................................ 99
    9.2.6 Abstraction and Separation ............................................................... 99
    9.2.7 Integration ....................................................................................... 99
    9.2.8 Maintainability ............................................................................... 99
List of Figures

Figure 2-1: A simplified subset of Ecore model ................................................................. 9
Figure 2-2: Ecore unifies Java, UML and XML ................................................................. 11
Figure 2-3: EMF development flow ..................................................................................... 11
Figure 2-4: GMF dependencies ........................................................................................... 13
Figure 2-5: GMF development flow ..................................................................................... 14
Figure 2-6: JET transformation development flow .............................................................. 17

Figure 3-1: SMC work flow ................................................................................................. 23
Figure 3-2: SMC interaction between Context class and Application class ....................... 23
Figure 3-3: SMC source code structure .............................................................................. 24
Figure 3-4: SMC state machine diagram example .............................................................. 31

Figure 4-1: SMC diagram editor development process ....................................................... 33
Figure 4-2: GMF SMC diagram editor example ................................................................. 34

Figure 5-1: SMC Ecore model root package properties ...................................................... 36
Figure 5-2: smc.ecore model class diagram ...................................................................... 36
Figure 5-3: SMC domain model first version ...................................................................... 37
Figure 5-4 setting Property Category property of actionName attribute ......................... 40
Figure 5-5 Setting SMC model object name ....................................................................... 42
Figure 5-6 Setting the id property of plugin.xml file in the Editor Code project ............. 42
Figure 5-7: Setting the category property of plugin.xml file in the Editor Code project .... 43
Figure 5-8: SMC model creation wizard ............................................................................ 43

Figure 6-1: GMF development process with notices ........................................................... 44
Figure 6-2: Specifying basic graphical definition for smc.ecore model ............................. 46
Figure 6-3: smc.gmfgraph model with default settings ..................................................... 47
Figure 6-4: MainMap figure descriptor .............................................................................. 48
Figure 6-5: Properties of the flow layout of the MainMap rectangle .................................. 49
Figure 6-6: StartState figure descriptor ............................................................................ 50
Figure 6-7: SMC compartments example ................................................................. 51
Figure 6-8: Action figure descriptor ........................................................................ 52
Figure 6-9: Adding Polyline decoration................................................................. 53
Figure 6-10: Adding new compartment ................................................................. 54
Figure 6-11: SMC palette at runtime ..................................................................... 55
Figure 6-12: Specifying basic tooling definition for smc.gmftool model ............... 56
Figure 6-13: Basic definition of smc.gmftool model ............................................. 56
Figure 6-14: Updating smc.gmftool model ........................................................... 57
Figure 6-15: Basic definition of the smc.gmfmap model ...................................... 59
Figure 6-16: MainMap top node reference properties ........................................... 60
Figure 6-17: MainMap node mapping properties ................................................ 61
Figure 6-18: Properties of the Feature Label Mapping of the MainMap Node Mapping .... 61
Figure 6-19: Setting a link mapping for simple transition type ......................... 63
Figure 6-20: Adding link constraints to the simple link mapping ....................... 65
Figure 6-21: Feature label mapping of simple link mapping ................................. 66
Figure 6-22: Feature label mapping of push link mapping .................................. 67
Figure 6-23: Properties of the “unset SMC diagram name” audit rule ................. 69
Figure 6-24: PrePushedState batch validation warning ....................................... 70
Figure 6-25: PrePushedState live validation error .............................................. 71
Figure 6-26: Enabling validation properties in the smc.gmfgem model ................ 73
Figure 6-27: SMC diagram plug-in code in the explorer view ............................... 74
Figure 6-28: SMC diagram editor in use example ............................................... 75

Figure 7-1 SMC batch validation examples ......................................................... 76
Figure 7-2 SMC live validation prevents assigning Default as a name for the Main Map ...... 77
Figure 7-3 SMC live validation prevents setting popArg for non pop transition .......... 78
Figure 7-4 SMC live validation prevents changing a transition type .................. 78
Figure 7-5 SMC live validation prevents changing toState .................................. 79

Figure 8-1: JET transformation development flow ................................................ 80
Figure 8-2: SMC JET transformation project ......................................................... 82
Figure 8-3: SMC JET transformation templates .................................................... 82
Figure 8-4: Setting a model loader and a model extension ..................................... 85
Figure 8-5: SMC JET transformation in use .................................................................89
Figure 8-6: SMC JET transformation output information example ..........................90
Figure 8-7: part one of SMC generated source code for subway turnstile state machine example 90
Figure 8-8: part two of SMC generated source code for subway turnstile state machine example 91
Figure 8-9: part three of SMC generated source code for subway turnstile state machine example
......................................................................................................................................91
Figure 8-10: Java skeleton class for transitions actions of subway turnstile state machine example
......................................................................................................................................92

Figure 9-1: SMC modeling and transforming tool developing and using process .................. 93
Figure 9-2: Typical GMF model creation and updating process ......................................95
Figure 9-3: Tooling definition model editor .................................................................96
Figure 9-4: Persisting transitions action name and argument in XMI file .....................97
Figure 9-5: Graphviz diagram produced from SMC generated Turnstile.dot file ...........103

Figure C-1: SMC model instance example...............................................................116
Figure C-2: Editing comment attribute......................................................................116
Figure C-3: A transition properties view.................................................................117
Figure C-4: A transition action names dialog..........................................................117
Figure C-5: SMC dynamic model instance example ..............................................118

Figure D-1: SMC creation wizards..........................................................................120
Figure D-2: SMC diagram editor example..............................................................121
Figure D-3: SMC model and transformation tool projects..........................................123
Figure D-4: Run configurations of SMC JET transformation plug-in............................124
Figure D-5: SMC JET transformation generated files...............................................125
Figure D-6: SMC JET transformation generated SMC source code ..........................126
Figure D-7: SMC JET transformation generated Java class ......................................127
List of Tables

Table 3-1: SMC supported programming languages .............................................................................. 22
Table 3-2: SMC source code example .................................................................................................. 29

Table 8-1: main.smc.jet template tags example..................................................................................... 86
Table 8-2: info.warn.error.main.smc.jet template tags example .............................................................. 87
Table 8-3: mainMap.smc.jet template tags example.................................................................................. 88

Table 9-1: Summary of Strengths and Weaknesses .............................................................................. 101
Chapter 1

Introduction

This chapter introduces the thesis topic. Section 1.1 gives an overview of the technologies used in the thesis. Section 1.2 discusses the motivation and contribution of the research work. Lastly, Section 1.3 outlines the content of the document.

1.1 Overview

Throughout the history of software development, increasing productivity and managing complexity by improving abstraction and automation has been the focus among developers. The leap from assembly to compiler languages has tremendously facilitated software development and increased its productivity. In this sense, software modeling and Model-Driven Software Development (MDSD) in particular aim to continue this trend and provide approaches that allow increased levels of abstraction by shifting the focus from being code-centric to model-centric. In addition, Domain Specific Modeling (DSM) development allows models to be based on domain concepts rather than implementation concepts [1].

The Object Management Group (OMG) has introduced a set of standards for use under one approach known as Model-Driven Architecture (MDA), which uses models throughout the software development process [2]. In particular, the OMG standards that MDA includes are: Unified Modeling Language (UML), Meta Object Facility (MOF), XML Meta-data Interchange (XMI), and Common Warehouse Meta-model (CWM) [3]. The main concept in MDA is the separation of the system specification from its implementation, which is accomplished by introducing Platform Independent Models (PIMs) and Platform Specific Models (PSMs).

First, the PIMs represent a higher level of abstraction capturing the system specification and they are expressed in a platform independent modeling language, such as UML. Moreover, the PIMs
conform to the MOF specification and they are, by default, persisted as XMI documents. Then, by applying a sequence of mappings and transformations, the PIMs are automatically transformed to PSMs. The PSMs allow adding implementation details that are not included in the PIMs. Finally, the PSMs are taken as input to generate code implementing the PIMs and targeting a particular platform. Due to the separation of concerns, automatic model transformation and code generation, MDA promises to accelerate the software development process by reducing the complexity and improving the quality and productivity [2, 4].

Using MDA standards requires modeling technologies and tools that unify and implement the stated OMG standards. For instance, the Eclipse Modeling Project (EMP) provides support to implement many of MDA standards [3].

Essentially, EMP [5] consists of several cooperating projects providing a set of modeling frameworks, tools and standard implementations. For instance, abstract syntax development is supported by the Eclipse Modeling Framework (EMF), graphical concrete syntax development is supported by the Graphical Modeling Framework (GMF), and Model-to-Model (M2M) and Model-to-Text (M2T) transformations technologies are also supported by the Eclipse Modeling Project (EMP).

First, the Eclipse Modeling Framework (EMF) is the core modeling project within Eclipse Modeling Project (EMP) projects [6]. It provides code generation to build tools and tree-based editors based on a meta-modeling language called Ecore. Moreover, EMF supports refining, querying, validating and transforming models via its integrated subprojects and tools.

Second, the Graphical Modeling Framework (GMF) provides a generative model-driven tooling component and runtime support to develop diagram editors for specific domains based on a sequence of model creation and transformation steps [7]. The GMF goal of is to simplify the development of the Eclipse Graphical Editing Framework (GEF) [8] diagram editors by leveraging EMF models. More precisely, GMF utilizes the EMF Ecore model to capture the
abstract syntax of a Domain Specific Language (DSL) and leverages GEF capabilities to
generatively develop models depicting the concrete syntax of that domain. Then, GMF allows
developing a mapping model to connect elements and features of the abstract syntax with the
corresponding elements of the concrete syntax. By generating a generator model based on the
mapping model, GMF allows the generating of code for a diagram editor. This editor can then be
used to create graphical models in the DSL.

Third, model transformations, including M2M and M2T, play a major role in the Model-Driven
Software Development (MDSD) paradigm because they allow data exchange between modeling
technologies and tools which enhances the scalability and interoperability of the automated
software development process. The Eclipse Modeling Project (EMP) supports model
transformations by providing implementations for M2M and M2T transformations languages [9, 10]. For instance, the M2T technologies include a Java Emitter Template (JET) component which
is used for transforming models into code or text.

At this point, a Domain Specific Language (DSL), unlike a general purpose language such as
UML, is designed to be used for a particular task [3]. Basically, a DSL describes types of domain
model elements and their relationships. By using Domain Specific Modeling (DSM) tools, such
as GMF, code is generated to create a customized modeling tool that allows modeling the domain
requirements graphically.

In this research, the State Machine Compiler (SMC) [11] is taken as a problem domain and a case
study for GMF. SMC is an open source Java application allowing generation of state pattern
classes from textual descriptions of state machines. Moreover, it supports generating source code
in fourteen programming languages.
1.2 Motivation and Contribution

Modeling is a well-known practice in all engineering disciplines; software engineering is not an exception, yet the importance of modeling in software engineering is essentially more critical, especially with the increasing complexity, and tremendous dependency of software products.

In software engineering, models can be used not only for documentation, communication and demonstration, but for specification, analysis, design, implementation and maintenance.

According to Software Productivity Research (SPR) [1], the average productivity in Java is only 20% better than in BASIC. However, the move from Assembly to BASIC shows 400% productivity increases [1]. The 400% increase was a result of automation and abstraction.

In this sense, Domain Specific Modeling (DSM) development aims to increase productivity of software development by raising the level of abstraction beyond code concepts and specifying the solution directly using domain concepts [1]. Then, code is generated from these higher level abstractions. As a result of the specificity and the code generation, programming details are hidden, which is analogous to the leap from assembly to compiled languages.

DSM has been used and tested in several industrial applications resulting in improved productivity by 5 to 10 times [1]. For example, Nokia Series 60/Python mobile application has been examined as a case study on DSM. Various DSM examples and case studies have been conducted including ones involving financial systems, web applications and embedded systems.

As a domain specific development environment, GMF was introduced in 2006 [12]. Therefore, it is relatively new, yet it is intensive framework built on top of powerful Eclipse tools and frameworks such as EMF and GEF. GMF aims to simplify the creation of GEF-like diagram editors in a model-driven based approach with the emphasis of the separation of concerns. On the other hand, several artifacts need to be developed to define and map the abstract and concrete
syntax of a specific domain in order to generate code implementing a diagram editor of the specific domain.

GMF is a powerful tool that allows the creation of a wide variety of graphical editors with many different features. As a consequence, the use of GMF is by no means straightforward and supporting documentation is still quite preliminary. Moreover, the development of graphical editors for non-trivial languages is quite challenging.

The thesis makes the following contributions:

1. A case study of the use of GMF for the creation of a graphical editor for a non-trivial language.

2. A list of strengths and weaknesses of GMF observed during the case study. In particular, the evaluation is based on criteria related to the overall user friendliness of GMF (usability, documentation), its design and effectiveness (support for abstraction and automation, productivity, separation of concerns), and its support for certain key activities such as model validation, integration, and maintenance.

3. A graphical editor that can be used for the input of SMC state machines and a transformation that can be used to transform the XMI representation of these state machines into the textual format expected by SMC, which is a popular open source tool for code generation from state machines. We intend to make the editor and the transformation publically available to the SMC community. To the best of our knowledge, no other graphical editor for SMC state machines currently exists.

1.3 Outline of the Document

- Chapter 2 gives background regarding the development of diagram editors using GMF and M2T transformation using JET. Moreover, it discusses related technologies.
• Chapter 3 introduces the necessary SMC concepts and features.

• Chapter 4 describes the requirements influencing the design and development of the SMC diagram editor.

• Chapter 5 gives a detailed explanation of the development of SMC EMF-models to generate an EMF tree-based editor for SMC model instances.

• Chapter 6 builds on Chapter 5 and illustrates the development of SMC GMF-models to complete the creation of the SMC diagram editor.

• Chapter 7 discusses the SMC diagram editor validation and gives some examples.

• Chapter 8 presents the SMC JET transformation.

• Chapter 9 discusses the thesis results.

• Chapter 10 contains the thesis conclusion.

• Appendix A lists the acronyms, Appendix B describes the implementation platform including the versions of the software used, Appendix C demonstrates the use of the SMC EMF tree-based editor, Appendix D explains the use of the SMC graphical modeling and transformation tool and Appendix E gives SMC commands and grammar.
Chapter 2
Background

This chapter provides background on various related topics. Section 2.1 discusses the Eclipse Modeling Framework (EMF). Section 2.2 introduces the Eclipse Graphical Editing Framework (GEF). The Eclipse Graphical Modeling Framework (GMF) is discussed in Section 2.3 while Section 2.4 presents the Java Emitter Templates (JET) transformation. Section 2.5 introduces Domain Specific Modeling (DSM) tools. Finally, Section 2.6 discusses related work.

2.1 Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework (EMF) is the core modeling project in Eclipse Modeling Project (EMP) projects [6]. It provides code generation to build Eclipse plug-ins and tools based on a meta-modeling language called Ecore. Essentially, EMF contains three main components: EMF.Core, EMF.Edit and EMF.Codegen [6]. First, EMF.Core consists of two elements: the Ecore model, which is used to define EMF models, and runtime support which provides model persistence, change notification and a reflective API. Second, EMF.Edit supports the display and edit of model instances in a basic tree-editor by providing content and label provider classes, property view and command classes that provide common services and operations. Third, EMF.Codegen provides Java code generation to build the basic tree-editor. In essence, the generated code is divided into three levels: Model Code, Edit Code and Editor Code [6]. Refining and validating models is also supported in EMF via its integrated subprojects and tools. Furthermore, EMF allows dynamically creating model instances via its reflective API.

2.1.1 Eclipse Modeling Framework (EMF) and Object Management Group (OMG)

EMF supports some of the OMG standards, including UML, Meta Object Facility (MOF), XML Metadata Interchange (XMI) and Object Constraint Language (OCL) [13]. These standards are part of the Model-Driven Architecture (MDA) approach introduced by the OMG [2]. This section briefly introduces the stated OMG standards and highlights their relationship with EMF.
2.1.1.1 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is the OMG standard for specifying and designing software systems [14]. UML2, a component in the Eclipse Model Development Tools (MDT) Project, is an EMF-based implementation of the OMG UML2 standards. It provides diagramming and meta-modeling capabilities [15]. In essence, the EMF Ecore model is an implementation of a subset of UML class diagrams [13].

2.1.1.2 Meta Object Facility (MOF)

EMF was started as an implementation of the Meta Object Facility (MOF) specification [16]. MOF is OMG’s standard meta-model defining a meta-modeling specification [17]. It specifies how models are created, integrated and transformed into different formats.

The MOF model defines two main packages: Essential MOF (EMOF) and Complete MOF (CMOF). The EMOF package contains simple classes with attributes and operations. In turn, the Ecore model of EMF is based on and aligned with the EMOF specification [13].

2.1.1.3 XML Metadata Interchange (XMI)

The XML Metadata Interchange (XMI) is OMG’S XML-based standard format for model storage [18]. It is a serialization form of MOF models. EMF uses XMI as a standard serialization for its Ecore models [13].

2.1.1.4 Object Constraint Language (OCL)

The Object Constraint Language (OCL) is a formal language standardized by OMG and is part of UML specification [19]. Typically, OCL is used for specifying constraints over models. OCL is also used to specify pre-conditions and post-conditions on methods and derivation rules of attributes. Applying OCL expressions on models allows creating more precise models that capture more aspects of the specification. OCL, a component in Eclipse Model Development Tools (MDT) Project, provides integration and
implementation of OMG’s OCL specification for EMF-based models [15]. Using OCL with EMF-based models allows specifying class constraints, initializing features, defining model audits and metrics rules, as well as, the implementation of derived features.

2.1.2 EMF Ecore Model

EMF provides a meta-model called Ecore model to represent models in memory. As it is stated in the previous Section 2.1.1, the Ecore model is used for class modeling to describe model structure. Figure 2-1 shows a simplified subset of the Ecore model [13].

![Figure 2-1: A simplified subset of Ecore model](image)

The simplified Ecore model includes four classes. First, the EClass class is used to model classes themselves. Multiple inheritances are supported through the eSuperTypes reference. Second, the EAttribute class is used to model attributes. Third, the EDataType class is used to represent the types of attributes. Data types are not modeled, but they represent primitive or object types defined in Java. Finally, the EReference class is used to model associations between classes. EReferences have a type representing the EClass at the other end of the association. Moreover, bidirectional associations are supported through the eOpposite reference. The EReference class also has lower and upper bounds attributes to specify a reference multiplicity and a Boolean attribute to specify if the association is a containment or not [13].
2.1.3 Creating Ecore Based Models

EMF supports several options to create models based on the Ecore model. Basically, models can be created using the Ecore editor, Java, UML, XML Schema or XMI [13]. The following numbered list provides a description of each method.

1. Ecore Editor. EMF provides an Ecore tree-based editor and an Ecore class diagram editor with a properties view by which Ecore models are created and edited. Alternatively, Ecore editors are also provided by third parties such as Topcased's Ecore Editor, Omondo's EclipseUML and Soyatec's eUML [13].

2. Java. EMF supports building Ecore-based models from annotated Java interfaces. Essentially, Java interfaces, including their methods, need to be marked with a @model annotation in order to be identified as model elements. Furthermore, the @model annotation can be followed by properties to provide additional information and direct code generation [13].

3. UML. This option requires using a UML modeling tool. EMF supports importing a Rational Rose model or a UML2 model to generate a corresponding Ecore model [13].

4. XML Schema Definition (XSD). XML Schema is a World Wide Web Consortium (W3C) standard for defining XML documents [20]. EMF supports importing and transforming an XSD file to an Ecore model by mapping XSD elements to the corresponding Ecore elements [13].

5. XMI. EMF uses XMI as a standard and default serialization format of its Ecore-based models; thus, using a text editor, XMI files can be created to represent Ecore models. Providing XMI as underlying standard to save Ecore models enables interoperability and unifies previous methods: Java, UML and XML Schema, as shown in Figure 2-2 [13].
2.1.4 EMF Development Flow

Figure 2-3 demonstrates the process of creating an EMF application from models.

As shown, an EMF Ecore-based model is created, and then transformed to a model called EMF generator model from which EMF generates code that is run as an Eclipse plug-in.

2.1.5 EMF Generator Model

EMF provides a creation wizard that allows transforming an Ecore-based model to a model called a generator model. The purpose of the EMF generator model is to add information that is not included in the Ecore model such as base package name, compliance level, model directory and model plug-in ID via its properties view in order to generate the EMF code. For instance, base package name, compliance level, model directory and model plug-in ID are set from via the properties view of the EMF generator model.
2.2 Graphical Editing Framework (GEF)

The Graphical Editing Framework (GEF) allows the creation of graphical editors from existing models. Basically, GEF consists of two Eclipse plug-in components: Draw2d (org.eclipse.draw2d) and GEF (org.eclipse.gef) [21]. First, the Draw2d plug-in provides painting and layouts to display graphics. Second, the GEF plug-in is built on top of Draw2d and it employs a Model-View-Controller (MVC) architecture which enables editing the model from the view. The model in GEF is any data that is persisted and has a notification mechanism. In most cases, the model is expressed using a set of Java classes [22]. The view refers to any graphics displayed on graphical or tree-based viewers, including figures and tree items. The controller, known as EditPart, provides the link between the view and the corresponding model element.

Despite the basic and advanced editor functionalities provided by GEF, creating graphical editors using GEF is a complex task [3, 22]. That is, users need to provide persistence and notification mechanisms in the model data. Moreover, users need to implement various EditParts which contain a set of EditPolicy classes for each visualized model element. The EMF, as discussed previously, provides persistence and change notification for its models; therefore, EMF models can be used in GEF to create graphical editors. Nevertheless, using EMF models with GEF is not straightforward because GEF and EMF use a different command infrastructure [3, 22].

2.3 Graphical Modeling Framework (GMF)

With the aim of simplifying the creation of GEF-like diagram editors based on EMF’s models, the Eclipse Graphical Modeling Framework (GMF) [7] project was introduced to provide a generative bridge between EMF and GEF. Figure 2-4 shows the dependencies between GMF generated diagram editors, GMF runtime, EMF, GEF, and the Eclipse platform [23].
In essence, GMF provides a generative model-driven tooling component and runtime support to create diagram editors based on a series of model creation and transformation steps. It uses an EMF Ecore model to define the abstract syntax of a specific domain language and defines graphical and tooling models to describe the concrete syntax of the language. Then, it allows developing a mapping model to connect elements of the Ecore model to the corresponding elements of the concrete models. By generating a generator model based on the previous mappings, GMF allows generating Java code and other configuration files representing a DSL that is run as Eclipse plug-in diagram editor.

2.3.1 GMF Main Components

GMF consists of two main components: tooling and runtime.

2.3.1.1 GMF Runtime Component

GMF provides two runtime options: a full runtime and light runtime [3]. The full runtime is the default providing a set of libraries, extension points and enhancements to the underlying EMF and GEF runtimes while the light runtime provides minimal requirements to run a diagram editor. Moreover, the GMF runtime supports common
editors operations such as palette, properties view, toolbars, geometrical shapes, saving a
diagram as an image with support for several image formats and printing services.

2.3.1.2 GMF Tooling Component
The GMF tooling component provides a generative model-driven approach to create and
map models in order to generate diagram editors code. Figure 2-5 shows an overview of
the workflow used during GMF-based development.

![GMF development flow](image)

Figure 2-5: GMF development flow

As shown in Figure 2-5, creating a diagram editor using GMF requires developing six
models which are: domain model (EMF’s Ecore model), EMF generator model, graphical
definition model, tooling definition model, mapping model and GMF generator mode.

2.3.2 GMF Domain Model (EMF’s Ecore Model)
The GMF Domain Model is essentially the EMF’s Ecore model [3]. As it was covered in Section
2.1.3, the EMF Ecore model is created by using one of the following options: Ecore editing,
annotated Java interfaces, UML, XML Schema (XSD) or XMI document.
2.3.3 EMF Generator Model

By loading the domain model (EMF Ecore model), the EMF generator model is created. The EMF generator model serves as a pre-step of generating EMF code. It provides access to all information needed to generate EMF code, and also it allows adding additional information that is not included in the domain model such as base package name, compliance level, model directory and model plug-in ID via its properties view [13].

2.3.4 GMF Graphical Definition Model

To visually represent domain model elements, the GMF Graphical Definition Model is used to define figures such as shapes and labels. Basically, the graphical definition model consists of a Canvas element which contains two main components [3]. The first component is a Figure Gallery which contains Figure Descriptors elements that define actual figures. A graphical definition model can have one or more Figure Gallery elements. The second component represents Canvas elements where Nodes, Connections, Diagram Labels and Compartments are defined. Canvas elements act as references to the figures defined in the Figure Descriptors and they are used in the mapping model to assign figures to corresponding domain model elements. Accordingly, Nodes reference figures, Connections reference links, Diagram Labels reference labels and Compartments reference nested figures. Having these layers, figures can be reused to represent more than one element in the domain model [3].

2.3.5 GMF Tooling Definition Model

The GMF Tooling Definition Model is used to define a Palette consisting of Creation Tools that are used to select and create figures on the diagram surface. Mainly, the tooling definition model has a root element called Tool Registry [3]. The Tool Registry element consists of the Palette element and some other elements, such as a Toolbar and a Menu; however, GMF currently implements only the Palette element [3]. On the Palette level, Tool Groups, Separators and Creation Tool can be added.
2.3.6 GMF Mapping Model

The GMF mapping model allows loading and mapping the domain model elements to the corresponding figures and creation tools that are defined in the graphical definition model and the tooling definition model respectively. Typically, Nodes, Connections, and Diagram Labels from the graphical definition model are mapped to the domain model elements and assigned Creation Tools from the tooling definition model.

The mapping model has a root element named Mapping which may contain the following elements as children: Canvas Mapping, Top Node Reference, Link Mapping, Audit Container, Metric Container, and Generic Style Selector; however, the last element is not yet implemented [3]. First, the Canvas Mapping element represents the highest mapping level which binds the root element of the domain model to the Canvas element from the graphical definition model and the Palette element from the tooling definition model. Second, the Top Node References elements manage mappings of Nodes (figures) from the graphical definition model and their Creation Tools from the tooling definition model to the corresponding elements (classes) of the domain model. Third, the Link Mappings elements are responsible for mapping Connections (links) from the graphical definition model and their Creation Tools from the tooling definition model to the appropriate elements (references) of the domain model. Fourth, the Audit rules are defined in Audit Containers and they have several properties and options that enable defining constraints on the domain model elements. Finally, Metrics rules specify measures for diagram elements. The Object Constraint Language (OCL) can be used in the mapping model to specify constraints and to initialize features in Node Mappings and link Mapping, as well as to define constraints in a form of audit and metric rules.
2.3.7 GMF Generator Model

The GMF generator model allows loading the mapping model and the EMF generator model in order to generate the code for the diagram editor plug-in. Thus, it is analogous to the EMF generator model in that it allows setting and adding code generation parameters [3].

2.4 Java Emitter Templates (JET)

Code generation and model transformation are important aspects in software modeling. The Eclipse Modeling Project (EMP) supports model transformations by providing implementation for Model-to-Model (M2M) and Model-to-Text (M2T) transformations technologies [5].

When generating code or text from a model, M2T technologies are used. The Java Emitter Templates (JET) is a component in the Eclipse M2T Project which is used by default to generate code or text from EMF models [10]. JET has an enhanced version called JET2 which is currently used in the M2T project [24]. Figure 2-6 shows a simplified workflow of an Eclipse JET transformation project.

![Figure 2-6: JET transformation development flow](image)

As an input, JET expects models persisted as XML standard documents [25]. By providing a collection of JET templates operating on the input model, JET generates the output based on these templates. In essence, JET templates are a mixture of plain text, which JET outputs unmodified, and commands called JET tags. JET tags are XML-like directives used to access and extract data from the input model. Typically, JET tags are grouped into four libraries [25], as follows:
1. *ControlTags* library which is used to access the input model.

2. *FormatTags* library which is used for text formatting.

3. *JavaTags* library which is used specifically when generating Java code.

4. *WorkspaceTags* library which is used for creating resources in the workspace.

Moreover, JET implements the XPath language to address parts of the input model [25]. XML Path Language (XPath) is a World Wide Web Consortium (W3C) recommendation language used for navigating XML-like documents [26]. It uses the path expressions structure, similar to common expressions that are used in directories, to navigate and select nodes of an XML document. Furthermore, XPath provides predicates with its expressions to perform more filtering over the selected nodes and a library of standard functions including functions for manipulating string, numeric and Boolean values [26, 27].

### 2.5 Domain Specific Modeling (DSM) Tools

A number of meta-modeling tools are available providing support for the creation of DSLs. These tools are typically based on two approaches: the mapping approach and the transformation approach [28, 29]. Both approaches use meta-modeling concepts to define the abstract syntax (domain model) and the concrete syntax (graphics model) of the language with a mechanism to define the relationship between them. Examples of the mapping-based approach tools include GMF [7], MetaEdit+ [30] and Microsoft DSL Tools (MS DSL) [31] while examples of the transformation-based approach tools include METAclipse [30], Tiger [32] and ViatraDSM [33].

First, mapping-based approaches use mapping models to assign an element from the graphics model to a corresponding domain model element. The mappings are then validated using some constraint languages, as OCL in GMF and custom languages in MS DSL and MetaEdit+ [29]. Thus, the DSLs functionality is based on these mappings.

Second, in the transformation-based approach the relationships between abstract and concrete models are defined dynamically by transformations using model transformation languages [29].
For instance, the Tiger framework uses a graph transformation language called AGG and the ViatraDSM framework uses a transformation language called Viatra2. The transformation languages specify what modifications must be applied to one of the models and how other models need to be updated accordingly. Therefore, the functionality of the DSLs is based and controlled by these transformations. A combination of two approaches, mapping and transformations, would improve the design and implementation of DSLs since the DSLs may contain complicated and simple parts [28]. A more specific and detailed comparison between DSM tools can be found in [34, 35].

2.6 Related Work

One of GMF principles is the separation of abstract syntax from concrete syntax to the most extent to allow reuse and interoperability, and then map specifications for the final generation. However, D. Kolovos and et.al [36] proposed a different approach. The authors stated that GMF is a powerful, flexible and generative framework, yet it is complex to use and error prone mainly because it requires developing and maintaining several models. Furthermore, GMF tooling component provides wizards for the creation of models’ initial definition. Nevertheless, the wizards produce good results only for simple meta-models. In addition, when errors occur while developing the models, GMF produces ambiguous error messages. As a result, current GMF development process shows a hard experience particularly for new developers leading to a negative perception of the usability and effectiveness of graphical modeling in general. To address these issues, D. Kolovos and et.al [36] proposed a “single-sourcing approach”. The approach embeds annotations and attributes to the Ecore model to allow adding graphics and generation information in one place. Then, automated M2M transformation is used to generate other models required by EMF and GMF. According to the authors, this approach reduces the complexity of GMF and EMF, and also improves productivity, quality, usability and maintainability.
Moreover, the Tiger project team [32, 37] stated that GMF does not support creating and editing complex commands and operations; therefore, the team proposed an extension of GMF in a form of graph transformation to define and generate complex editing commands. Typically, the graph transformation extension adds an additional model called *Transformation Rule Model* to GMF models. In addition to adding or deleting a complete decision structure in one step, complex editing commands offered by the extension can be used for model optimizations such as refactoring. The Tiger project has been added to the Eclipse modeling projects [38] under the project name Henshin [39].

Furthermore, Epsilon [40], a component in Eclipse Generative Modeling Technologies (GMT) Project [41], provides a set of meta-model languages for creating, querying and modifying EMF models. Among the languages developed in Epsilon project is the Epsilon Wizard Language (EWL), which provides interactive in-place model transformations in form of wizards selected by the user. D. Kolovos and et.al [42] propose a bridge between GMF editors and EWL allowing users to execute wizards in-place. The proposed approach adds a new submenu named *Wizards* to the context menu of GMF editors allowing the specification of custom wizards. The advantage of this extension is that it enables automating common model refactoring and repetitive tasks. The authors also plan to implement interactive fault analysis in GMF editors.

Additionally, the Modeling Amalgamation Project [43] was recently added to the Eclipse Modeling Project projects with the aim of providing improved integration, packaging and usability of GMF. Moreover, the Generic Eclipse Modeling System (GEMS) [44] Project aims to remove some of the GMF low level work by providing an integrated graphical modeling environment to facilitate the usability and interoperability of Eclipse modeling technologies. The GEMS project is based on EMF, Draw2D and GEF with the goal of providing constraint solver integration to allow rapid development of domain specific modeling tools. Moreover, GEMS focuses on enabling complex analysis, queues, simulations and constraint batch processing by
including a facility called Model Intelligence Guides (MIGs) to provide intelligent modeling mechanisms. With MIGs, a user participates in an interactive process of specifying elements of a model, adding and debugging constraints, as well as, automating model construction and optimization. Constraints, in addition to Java and OCL, can be specified using Prolog. Thus, GEMS appears to replace GMF [34].

In his MSC thesis, Tarek Abdunabi [45] developed an XMachine diagram editor using GMF from an existing XMachine model. As the author stated, the developed XMachine diagram editor allows creating XMachine diagram editors containing states and transitions, and then it generates Java test cases from the model instances of the diagram using a JET transformation. Although the author’s thesis aim was only to develop XMachine graphical modeling tool using GMF, the author found that GMF provides rich features and support to produce fully functional XMachine graphical modeling tool. Moreover, the author stated that GMF makes it easy to extend and add new features to the developed XMachine graphical modeling tool. However, the author found using GMF is not straightforward and results in a long learning curve, especially, due to the lack of documentation.

Tobias Jahnel [46] in his thesis MSC developed a GMF Compare plug-in which allows the graphical comparison of models. The GMF Compare plug-in is based on the EMF Compare engine and integrates with GMF editors. Despite the lack of support and complexity mainly in technical details and source code, the author found GMF is well documented; yet, it took a long time to get familiar with the structure of GMF.
Chapter 3
State Machine Compiler (SMC): A Case Study

This chapter provides the necessary background on the State Machine Compiler (SMC) requirements and features. SMC [11] is taken as a case study to represent a problem domain for which a DSL in a form of a diagram editor is developed using GMF. Moreover, a JET transformation plug-in is developed to transform SMC model instances, created by the SMC diagram editor, into SMC source code and a Java skeleton class.

SMC is a popular open source Java application which has over 12,000 downloads since January 2008 and it allows code generation from textual descriptions of state machines. Currently, SMC supports generating code in fourteen programming languages. Table 3-1 shows the supported programming languages [11].

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Lua</td>
<td>Ruby</td>
</tr>
<tr>
<td>C++</td>
<td>Objective-C</td>
<td>Scala</td>
</tr>
<tr>
<td>C#</td>
<td>Perl</td>
<td>Tcl</td>
</tr>
<tr>
<td>Groovy</td>
<td>PHP</td>
<td>VB.net</td>
</tr>
<tr>
<td>Java</td>
<td>Python</td>
<td></td>
</tr>
</tbody>
</table>

Table 3-1: SMC supported programming languages

3.1 Using SMC

As a command-line and Java-based tool, SMC takes the following command line arguments:

```
Java –jar smc.jar –{target language} -{options} smc-file-name.sm
```

Appendix E illustrates the command line use and arguments as provided in [11]. In particular, the last argument (smc-file-name.sm) represents an SMC textual file which contains SMC source code that a user writes to describe a state machine. SMC compiles this file to generate code. In essence, the generated code is saved in one file containing a main class called Context class which contains inner classes implementing the states of the state machine. Figure 3-1 shows SMC work flow [11].
In addition to the SMC source code, SMC users need to write a class called *Application* class to interact with the generated *Context* class. Figure 3-2 depicts the interaction between the generated *Context* class and the user *Application* class [11].

Typically, the *Application* class should instantiate the generated *Context* class and implement transitions actions as methods. The transitions action methods should have public or package accessibility, as well as they should be declared as void methods; otherwise, SMC ignores any return value. Moreover, the transitions action method names must match the names of the transitions actions defined in the SMC source code.

### 3.2 SMC Source Code

SMC source code can be divided into three main sections: *verbatim*, *setting* and *map* section, as shown in Figure 3-3.
3.2.1 Verbatim Section

The verbatim section is optional and it is enclosed between %{ and %} delimiters [11]. It is used to include copyrights notes which SMC copies literally into the generated code. Both // and /*...*/ comment formats are supported.

3.2.2 Setting Section

The setting section sets the SMC file according to the targeted programming language for which SMC will generate code. For instance, the setting section shown in Figure 3-3 considers Java as a target language. The %class keyword specifies the name of the Application class that a user has written to interact with the generated Context class. The SMC compiler will assign to the generated Context class the same name as the name of the Application class specified by %class keyword, but suffixed by the word Context. The %start keyword specifies the name of the map
that will be considered the main map and a state name in this main map which is used as the start state of the state machine [11].

3.2.3 Map Section
The map section is meant to group logically related states into one block [11]. The SMC file may include several maps, but the main map specified in the %start keyword is required. Each map starts with the %map keyword followed by its name. Then, states are encapsulated between %% and %%% delimiters. Each state has a name and may have Entry or Exit sections listing actual actions between /and/ symbols. A state action has a name and may have a comma-separated argument list enclosed in parentheses and followed by a semicolon. Then, a state lists its outgoing transitions between /and/ keywords. A transition definition includes its name and may have a comma-separated argument list in parentheses and a guard in brackets; then, it should include a state name representing the next state of the transition. A transition also may include actions enclosed between /and/ keywords. Each action has a name and may have a comma-separated argument list enclosed in parentheses and followed by a semicolon. Typically, the introduced transition definition represents a simple transition type, but SMC defines other transitions types which may have different definition, as will be presented in the following sections. Moreover, SMC defines a special state called Default state.

3.3 SMC Default State
A map may include one Default state [11]. The Default state must be named Default. Moreover, transitioning to the Default state is not allowed because the purpose of the Default state is to define transitions that should be taken when any other state in the same map receives a transition to handle, but it is not defined among its outgoing transitions. A state may receive a transition that is undefined in that state from a special transition type called pop transition.
3.4 SMC Transition Types

SMC distinguishes between transitions according to their use; therefore, it defines a number of transition types. Transition types are: simple, loopback, Default, push and pop [11].

- The simple transition is a standard transition that transitions from a current state to a new state in the same map.

- The loopback transition is used when there is a need to remain in the same state. The keyword nil is used as a name of the next state indicating a loopback transition. Alternatively, the current state name can be used instead of the keyword nil.

- The Default transition must be named Default and it can be defined in any state to serve as a backup for all transitions in that state when the state receives a transition that is undefined in it. When a Default transition is defined in the Default state, it serves as a fallback for all undefined transitions in the state machine. Since a Default transition can be triggered if there is any undefined transition, it may take no argument. Moreover, since defining Default states and Default transitions is optional, it is the case that SMC may encounter undefined transitions resulting in a transition undefined exception [11].

- The push transition is used to transition between states in different maps. It supports two syntaxes. First, the initial syntax is as follows:

  name(args)[guard] push (map name :: state name) { action name(args);}

  The push transition must have a name, may be followed by a comma-separated argument list and a guard. Then, the keyword push must be added with a map name and a state name separated by a double colon and enclosed in parenthesis. Second, the new syntax is as follows:

  name(args)[guard] state name/push(map name :: state name) { action name(args);}
The new syntax causes to transition to a state in the current map prior to pushing to a state in a different map [11].

- The *pop* transition is used to return to a state that issued the corresponding *push* transition. Thus, it does not specify a next state, but instead it takes an optional argument that includes a transition name which should be one of the outgoing transitions defined in the returning state. The *pop* transition syntax is as follows:

  \[
  \text{name (args)[guard]} \quad \text{pop(a transition name, additional args) \{ action name( );}}
  \]

  The additional arguments will be passed to the named transition which in turn should define the corresponding arguments. If a *pop* argument representing a transition name is not provided or that transition is not defined in the return state, the *Default* state or *Default* transition will be taken according to certain precedence [11].

### 3.4.1 Transition Actions

The actions of a transition are enclosed in a pair of curly brackets “{ }”. Basically, an action definition consists of a name and an argument enclosed in parenthesis and ending with a semicolon. The argument list must be either empty or consist of a list of comma-separated literals. Method calls can be also used in the argument list. When calling a method defined in the generated *Context* class, the method name should be prefixed with *ctxt* keyword [11].

### 3.4.2 Transition Guards

A transition’s guard is enclosed in a pair of square brackets “[ ]" after a transition name and its argument if they exist. SMC copies the guard condition literally into the generated code; therefore, the condition must be valid target language source code. When the guard condition calls a method defined in the generated *Context* class, the method must be prefixed with the *ctxt* keyword [11].
3.4.3 Transition Precedence

A state may have multiple transitions with the same name and argument list as long as they all have unique guards [11]. In this case, SMC checks the transitions in order of presence except for the unguarded version which will always be taken if all the guarded versions fail. In particular, transition definitions have the following precedence:

1. Guarded transition
2. Unguarded transition
3. The Default state's guarded transition
4. The Default state's unguarded transition
5. The current state's guarded Default transition
6. The current state's unguarded Default transition
7. The Default state's guarded Default transition
8. The Default state's unguarded Default transition

3.5 SMC State Machine Example

Table 3-2 shows SMC source code example describing a state machine for a subway turnstile with normal and diagnostic mode.
The SMC subway turnstile state machine example shown in the Table 3-2 consists of two maps. First, the NormalMode map is the main map, as specified in the SMC keyword %start, and it defines the start state (Locked) and the Unlocked state with their transitions. Second, the DiagnosticMode map has three states with their transitions. In particular, the DiagnosticMode map includes the Default state with a Default transition. As described in the Section 3.3 and Section 3.4, a map may contain one Default state and Default transitions defined in any state to serve as a backup mechanism. The subway turnstile state machine shown in the Table 3-2 can be described as follows:

- Turnstile is initially in the Locked state.
- When a Coin transition is triggered in the Locked state (a user inserts a coin), the Unlock action is performed and the turnstile transitions to the Unlocked state.

- While in the Unlocked state and if the user passes through the turnstile, the Pass transition is taken and the Lock action is called and the turnstile transitions back to the Locked state.

- When one of the two Diagnostic transitions occur in the Locked state or Unlocked state, the push transition pushes to the Check state in the DiagnosticMode map and the SetLockedStatus or SetUnlockedStatus action is performed.

- When a Coin or Pass transition is triggered in the Check state, the turnstile remains in the Check state.

- While in the Check state and if a Fix transition is triggered and the fix attribute is false, the turnstile remains in the Check state, otherwise, the turnstile transitions to the Fixed state.

- While in the Fixed state one of the two Return transitions is taken depending on the result of their guards, then, the turnstile transitions back to the NormalMode by including a transition name in each Return transition argument. The first Return transition argument includes Coin which is a transition name defined in the Locked state and the second Return transition argument includes Pass which is a transition name defined in the Unlocked state.

A state may have multiple transitions with the same name and argument list as long as they all have unique guards. Transitions precedence was introduced in the previous section.

Figure 3-4 shows the state diagram of the subway turnstile state machine shown in Table 3-2 without transitions actions.
It can be seen that the two \textit{pop} transitions specify no next state, but instead a transition name that should be defined in the state that issued a \textit{push} transition. If a \textit{pop} transition defines no transition name in it argument, the \textit{Default} state or \textit{Default} transitions precedence is taken if they are defined, otherwise, SMC issue a “transition undefined exception”.

Figure 3-4: SMC state machine diagram example
Chapter 4
Diagram Editor Design Requirements

This chapter briefly discusses the design requirements of diagram editor. Section 4.1 overviews GMF diagram design considerations, and then Section 4.2 presents SMC diagram editor requirements.

4.1 GMF Diagram Design Decisions

Creating a diagram editor for a specific domain using GMF requires mapping the abstract syntax of the domain to its graphical concrete syntax by developing and connecting a chain of models. Whereas the graphical concrete syntax is depicted by GMF models, the abstract syntax is defined by EMF models. On the one hand, while models that represent the abstract syntax serve as a cornerstone of the diagram editor, the design of these models should consider the subsequent development. On the other hand, many other important factors should be considered. First, solid analysis and understanding of the requirements and specification of the problem domain facilitates the design and creation of all models. Second, displaying information visually requires the management of graphics and associated text, as well as layouts. In summary, the design of the diagram editor is influenced by:

- The requirements of the problem domain and weather it has coherent specification or not.
- The capabilities of GMF, and
- Whether or not the input created with the editor are to be processed further via, e.g., M2M or M2T transformations.

4.2 SMC Diagram Editor Requirements

In Chapter 3, SMC requirements and features were discussed in detail. Therefore, the SMC diagram editor to be developed must adhere to the SMC specification. The development of the SMC diagram editor using GMF requires developing several models and Eclipse plug-in projects.
More precisely, the SMC diagram editor consists of EMF and GMF models with their plug-ins projects, as well as, a JET transformation plug-in project. Figure 4-1 shows an overview of the process of developing and using SMC diagram editor plug-ins.

![Figure 4-1: SMC diagram editor development process](image)

Looking back to the SMC source code and state diagram example for the subway state machine shown in the Table 3-2 and Figure 3-4 in the previous chapter, Section 3.5, it can be seen that the GMF SMC diagram editor must contain a single main map with a start state and may have a number of states, and also may contain a Default state. Moreover, SMC diagram editor may include one or more maps which may have a number of state and a Default state with their transitions and attributes. Maps and states are represented by nodes (figures) while transitions are represented by connections (lines). That is, every node represents a specific SMC entity, such as a map or state. The type of the node determines its shape, properties and behavior at runtime which is specified using OCL. Moreover, connections represent SMC transition types. The type of the transition determines its shape, properties and behavior at runtime which is specified using OCL. All other features of SMC such as state actions, and transition arguments, guards and actions are associated with their nodes or connections as required. The SMC diagram editor validates user input and gives error, warning or information messages in live or batch mode as appropriate. Then, the user can transform the SMC model instance created using the SMC diagram editor to SMC source code and a Java skeleton class by running the SMC JET transformation plug-in.
Figure 4-2 shows a screen shot of the SMC GMF diagram editor to be developed being used to draw the subway state machine example given in Chapter 3, Section 3.5, Table 3-2.

Figure 4-2: GMF SMC diagram editor example

As shown, the developed SMC diagram editor has two files, as shown on the Explorer view on the right side of the figure. The first file is (Turnstile.sm) which represents the SMC model instance of the subway turnstile state machine and the second file is (Turnstile.smc_diagram) which is open and represents the diagram editor. On the right side of the figure there is the SMC Palette which is used to create SMC elements on the diagram surface. Moreover, transition actions and their arguments are entered using a dialog box from the Properties view, as shown at the bottom of the figure. The following chapters explain how the SMC GMF diagram editor shown in the figure is developed along with its JET transformation project.
Chapter 5
Developing SMC EMF Models

The development of GMF-based diagram editors consists mainly of two successive phases. The first phase consists of developing EMF models and plug-ins projects, and the second phase requires the development of GMF models and the diagram editor plug-in project. This chapter describes the development of SMC EMF models that allow creating and editing SMC model instances in an EMF tree-based editor whereas Chapter 6 builds on this chapter and illustrates the creation of GMF models to complete the development of the SMC diagram editor.

5.1 Creating SMC Domain Model (EMF’s Ecore Model)

A new GMF project is required to be created using the New GMF Project Wizard. The wizard also includes an option to open the GMF dashboard view. Instead of invoking EMF and GMF new model creation wizards from the File > New menu, the GMF dashboard view provides a guide line to create all models.

Looking back at the example provided previously in Table 3-2 and Figure 3-4 in Chapter 3, Section 3.5, it is clear that the SMC domain model should contain maps which contain states and transitions. However, which other concepts are needed, what the relationship between them is and which constraints they are subject to is less clear. These questions are answered in this section by creating a GMF domain model representing essentially, the abstract syntax of SMC state machines.

As it is mentioned in Chapter 2, Section 2.1.2, the GMF domain model is essentially an EMF Ecore model which can be created either directly using the Ecore editor, or by loading annotated Java interfaces, a UML model, or an XML Schema.

In this project, the GMF domain model for SMC (smc.ecore) is created directly using the Ecore model editor. Each Ecore model has a single root package which has properties that must be set. Figure 5-1 shows the properties of the root package of the smc.ecore model.
Sub-packages and classes with their attributes are added to the Ecore model using the Ecore tree editor, shown in Figure 5-1 or by generating the Ecore class diagram from the Ecore tree editor. Figure 5-2 shows the Ecore class diagram of the smc.ecore model.
It took a substantial amount of time to construct the `smc.ecore` model, shown in the Figure 5-2. As shown, the `smc.ecore` model structure is simple; however, it satisfies the SMC language requirements and facilitates the JET transformation, as will be explained in the next chapter while developing all SMC diagram editor models. Two of the many different versions of the `smc.ecore` model are discussed below.

The first attempt to construct the `smc.ecore` model was done based on the SMC generated code as shown in Figure 5-3.

![Figure 5-3: SMC domain model first version](image)

However, this version makes developing the SMC graphical editor more complicated because instead of having abstraction such as abstract map and abstract state directly providing inheritance for different maps and states, the SMC generated code is centered mainly around a class called `DefaultMap` which subclasses from the `AbstractState`, as shown on the Figure 5-3.

The second version was improved by adding abstractions as described above, yet it was found difficult to use with the JET transformation. Therefore, the `smc.ecore` model shown in Figure 5-2
serves both the SMC diagram editor and SMC JET transformation by capturing the SMC abstract syntax and keeping the JET transformation simple to the most extent.

### 5.1.1 SMC Domain Model Structure

As shown in Figure 5-2, the `smc.ecore` model elements are maintained in the `SmcModel` class. Moreover, the model contains a single `MainMap` element and a number of `Map` elements. First, the `MainMap` element is required and must contain a single `StartState` element and may have a `DefaultState` element and a number of `State` elements. Second, the `Map` element is optional and may contain a `DefaultState` element and a set of `State` elements. Furthermore, a `StartState`, `DefaultState` and any `State` elements contain their outgoing `Transition` elements, as well as `entry` and `exit` `StateAction` elements. Two enumerations types called `TransitionType` and `ActionType` are also defined in the model. The following list highlights some important features of the `smc.ecore` class diagram.

- The `ID` property of the attribute `name` in `NamedElement` and `SmcModel` class is set to `true` which causes the actual value of the attribute `name` to be unique and to be used in the XMI serialization instead of its fragment path [13]. Setting the `ID` property to `true` will help retrieving the value of the attribute `name` during the SMC JET transformation development. Unlike actions and transitions, names of states, but not of `DefaultState`, and maps must be unique; therefore, `Action` and `Transition` classes do not inherit from `NamedElement`. Similarly, `DefaultState` must be named `Default`; thus, it does not subclass the `NamedElement`. The `DefaultState` name will be constrained and initialized to `Default` when developing the SMC mapping model.

- The `StateAction` class has the `actionType` attribute of type `ActionType` enumeration and `id` attribute of type `EInt`. When developing the SMC mapping model, the `id` attribute will be used to validate the `Entry` and the `Exit` action types.
• In the Transition class, the actionName attribute is defined as a multiple value attribute by setting its Lower Bound property to 0 and Upper Bound property to -1, and similarly to the actionArg attribute. This is because a transition may have more than one action. Moreover, the popArg attribute represents a specific argument associated with pop transitions. This attribute will be constrained and validated when developing the mapping model, as well as for the prePushedState attribute which represents a state name associated only with push transitions. Furthermore, the reason for specifying the outgoing end reference as containment is to store and encapsulate outgoing transitions in the XMI serialization as inner elements for each state element which eases transforming transitions elements during the development of the SMC JET transformation.

5.2 Creating SMC EMF Generator Model

The next step is to transform the smc.ecore model to a platform specific model (PSM). The PSM of an Ecore model is represented by EMF generator model. A new EMF generator model is created by invoking EMF model creation wizard which allows importing the Ecore model (smc.ecore) to create the EMF generator model (smc.genmodel).

Modifying and setting the properties of the EMF generator model (smc.genmodel) model elements allows to control and direct the code generation. For instance, some properties allow specifying directories for Model Code, Edit Code, Editor Code and Test Code, and also they allow specifying names for plug-in IDs. Furthermore, the Compliance Level property should be set to 5.0 or higher. Copyrights can be provided in the Copyright Text property while the Base Package property of the root package found under the top level element of the EMF generator model (smc.genmodel) model must be set.

Moreover, to allow multi-line text to be entered, the Multi-line property is changed from false to true for the copyright attribute of the SmcModel class, and also for the comment attribute in all classes. Since the id attributes in the Transition class and the StateAction class are used only for
validation purposes when developing SMC mapping model, their Type Property property is set to None indicating that the id feature is not displayed in the properties view.

Furthermore, because the Transition class includes several attributes which are displayed as properties in the properties view of transitions, it is convenient to group these properties in categories, and thus, they can be recognized and edited easily. For instance, “Transition Actions” is entered as a value for the Property Category property for both actionName and actionArg attributes to group them under the same category entitled Transition Actions. Figure 5-4 shows the setting of the Property Category property of the actionName attribute in the Transition class.

![Figure 5-4 setting Property Category property of actionName attribute](image)

Since, fromState and toState references contained in the Transition class and incoming reference contained in the AbstractState class are not containment references, their Notify property by default is set to false. However, the Notify property is changed to true to notify the GMF runtime and update these references in-place without closing and reopening the diagram. Moreover, to prevent modifying incoming references from the property view, its Property Type property is changed from Editable to Readonly. The reason for this change is that in addition to the fact that incoming references are set automatically, modifying incoming references from the property view
will cause errors because of the OCL constraints that will be added when developing the SMC mapping modeling.

### 5.3 Generating SMC EMF Code

In addition to testing the functionality of SMC model instances in a basic tree editor, generating EMF code is required since the SMC Diagram Code that will be generated using GMF has some dependence on EMF code. In essence, the EMF code to be generated is classified in four categories: Model Code, Edit Code, Editor Code and Test Code. However, generating EMF Editor Code and Test Code is not required since the SMC Diagram Code that will be generated using GMF is used to create SMC diagrams and model instances. The generated EMF code contains Java code and other plug-in configuration files. First, the Java code, in addition to some utility classes, provides implementation for all classes in the domain model, which allows creating and editing model instances in a tree-based editor, and it also persist the model instances as XMI file by default. Second, the plug-in files identify the code as Eclipse plug-ins by providing information and settings to guide the building and packaging of the plug-ins. They also declare the projects dependencies on other Eclipse plug-ins. Creating and using SMC EMF model instances is described in the Appendix C.

### 5.4 Updating SMC EMF-generated plug-ins and properties files

Plug-in and properties files identify each project as an Eclipse plug-in and declare its various properties and dependencies. The plugin.properties file in each project includes the pluginName and providerName property. The default value of the providerName property is www.example.org which is changed to www.cs.queensu.ca. Moreover, the plugin.properties file in the Edit Code project includes the _UI_SmcModel_type property with a default value set to Model. This value (Model) is the desired SMC model object representing SmcModel class and it should be selected in the SMC model creation wizard. The value Model is changed to SMC Model which identifies the model object clearly. Figure 5-5 shows the impact of this change.
By default, a new SMC model creation wizard is found under the Example EMF Model Creation Wizards category. To create a separate category for the SMC model creation wizard some properties need to be modified. First, the plugin.properties file found in the Editor Code project includes the property: _UI_Wizard_category = Example EMF Model Creation Wizards which is changed to be _UI_Wizard_category = SMC Creation Wizards. This will only change the name of the category. To have a separate category, other properties of the plugin.xml file found in the Editor Code project need to be changed as follows: the id property in the Extensions tap of the plugin.xml file has org.eclipse.emf.ecore.Wizard.category.ID value which is changed to ca.queensu.cs.afmg.smc.Wizard.category.ID, as shown in Figure 5-6.
Then, the value of the *id* property is assigned to the *category* property, as shown in Figure 5-7.

Figure 5-7: Setting the category property of plugin.xml file in the Editor Code project

The results of previous changes are shown in Figure 5-8.

Figure 5-8: SMC model creation wizard
Chapter 6
Developing SMC GMF Models

This chapter builds on the previous chapter to complete the development of SMC models and diagram editor plug-in project. Two EMF models, the Ecore model (smc.ecore) and the generator model (smc.genmodel) have been created and the EMF code has been generated. The generated EMF code was divided into four plug-ins projects representing, Model Code, Edit Code, Editor Code and Test Code. In order to proceed in creating a diagram editor for SMC using GMF, four GMF models are to be developed. The models are: graphical definition model, tooling definition model, mapping model and GMF generator model. Then, Diagram Code is generated from the last model. Figure 6-1 shows the GMF development process annotated with notes providing the chapter and section number where the text explanation can be found.

Figure 6-1: GMF development process with notices

The work presented in this chapter is rather technical and GMF-specific. It, however, represents a substantial amount of work, because currently available resources do not explain the use of GMF in a way that enables users to easily create graphical editors. More precisely, using GMF involves several steps during which several artifacts are created with many attributes that need to be set.
The values of these attributes are crucial to the correct functioning of generation process and the generated editor. Our case study thus required extensive experimentation to determine the correct values of many attributes via a laborious trial-and-error process. Because the determination of the correct attribute settings required this lengthy process, they are included in this chapter making the text rather technical. However, to further increase the value of our work, the purpose of the attributes and the effect of their values are explained as much as possible. To increase readability many examples and screenshots are provided.

6.1 Developing SMC Graphical Definition Model

The graphical definition model defines figures and connections that are displayed on a diagram surface. Basically, the smc.ecore model is loaded, using the Simple Graphical Definition Model Wizard, and a Diagram Element is selected from its classes. The Diagram Element represents the Canvas of the graphical definition model which is mapped in the mapping model to the root element of the Ecore model. In case of the smc.ecore model, the SmcModel class is the root element and is selected as the Diagram Element. The next page of the wizard is for specifying the basic graphical definition of the domain model elements. It lists all domain model elements with default selection for each element as a node, link, label or nothing. Figure 6-2 shows the basic graphical definition specified for the smc.ecore model elements.
As shown in the Figure 6-2, some elements are selected as nodes, one element is specified as a link and labels for each selected element are also specified. Specifically, MainMap, Map, StartState, State, DefaultState, StateAction and Action classes are selected as nodes and for each the attribute name is specified as a label except for the StateAction class which takes its actionType enumeration as a label. Finally, the Transition class is specified as a link and its attribute name is selected as a label. Since transitions have other information that will need to display in the same label of the attribute name and because the Simple Graphical Definition Model Wizard does not support specifying more than one attribute for the same label, transition labels information will be set according to each transition type when developing the mapping model. Figure 6-3 shows the created basic smc.gmfgraph model definition.
As shown, depending on the selections made during the creation of the `smc.gmfgraph` model, GMF creates the model containing default figures, nodes, connections and diagram labels with default settings and properties.

### 6.1.1 Updating the SMC Graphical Definition Model

In order to have all required graphical information included and set correctly, some default settings of `smc.gmfgraph` model elements are modified and some new elements are added. The following numbered list illustrates each updated element.

1. **MainMap Figure Descriptor.**

   Figure 6-4 shows the updated *MainMap Figure Descriptor* with its corresponding *Node* and *Diagram Label* elements.
Basically, the MainMap Figure Descriptor contains two children, a Rectangle and a Child Access. The Child Access is used to access the Rectangle Label. To distinguish the MainMap figure from Maps figures, a blue Foreground child is added to the MainMap Rectangle, and also the Rectangle Label is given a Basic Font child with BOLD Style and 10 Height properties. To keep the Label from being too close to the Rectangle’s left and right edges, an Insets child is added to the Rectangle with Left and Right properties set to 5. Moreover, a Compartment element will be added to the MainMap Rectangle to contain StartState, State and DefaultState figures. Adding Compartments will be discussed in the next section and a screen shot showing a compartment example is given in Figure 6-7. A number of layout types such as Flow Layout, Grid Layout, Border Layout, XY Layout and Stack Layout are provided to add as children to figures. The Flow Layout is used here because it is considered a good choice since it arranges figures in rows or columns and has a number of properties that are used to control the layout. Whereas, for instance, the Grid Layout has only two properties which are Equal Width and Num Cloums and XY
Layout has no properties. Therefore, a Flow Layout child is added to the MainMap Rectangle. Figure 6-5 shows the properties of the Flow Layout child.

Figure 6-5: Properties of the flow layout of the MainMap rectangle

2. Map Figure Descriptor

The Map Figure Descriptor resembles the MainMap Figure Descriptor, except the added Foreground child is set to black and the Rectangle Label does not include a specific Basic Font child. Similarly to the MainMap Rectangle, the Map Rectangle will include a Compartment which is used to hold State and DefaultState figures.

3. StartState Figure Descriptor

The default Rectangle of the StartState Figure Descriptor is deleted and a new Rounded Rectangle child is added with its Corner Height and Corner Width properties set to 25. Moreover, a number of children are added to the Rounded Rectangle, as presented in the following Figure 6-6.
As shown in Figure 6-6, when deleting the default Rectangle and adding the Rounded Rectangle, the Child Access element, which is used to get the value of the Rectangle label, references no label. Therefore, the newly created Label of the Rounded Rectangle is selected as a value for the Figure property of the Child Access; otherwise, an error occurs when developing the mapping model since a Child Access element must not be blank, but reference a label.

4. State Figure Descriptor

Because the StartState Figure Descriptor is analogous to the State Figure Descriptor, it is copied and pasted, then set to represent State Figure Descriptor.

5. DefaultState Figure Descriptor

Similarly to StartState and State figure descriptors, the DefaultState Figure Descriptor is updated and set to represent the Default state.

The StartState, State and DefaultState Rounded Rectangles will include two compartments to hold the Entry figure and the Exit figure. For instance, Figure 6-7 shows a screen shot of an SMC diagram.
It can be seen that a MainMap element is created on the surface of the SMC diagram editor, and also a start state, state and Default state are created in the states compartment inside the MainMap figure. Moreover, the state named Run and the Default state contain compartments to hold Entry and Exit figure and the Entry and Exit figures also contain a compartment to hold the actual actions figure which is a rectangle without borders. Compartments are collapsible, as is the case with the start state.

6. StateAction Figure Descriptor

When first creating the smc.gmfgraph model and specifying the basic graphical definition, the StateAction class was selected as a node with its actionType attribute as a label. Therefore, the graphical definition creation wizard produced one Figure Descriptor for the StateAction element with corresponding Node and Diagram Label elements. Since reuse of figures is allowed, the Figure Descriptor of StateAction is updated and used to represent both action types, Entry and Exit.

Accordingly, the StateAction Figure Descriptor needs to be referenced by two Nodes and two Diagram Labels. While a new Node and a Diagram Label are added to the Canvas.
element to represent Exit actions, the existing Node and Diagram Label are used to represent Entry actions. When discussing adding compartments in the next section, the StateAction figure will include a Compartment to hold action figures, a screen shot example was shown previously in Figure 6-7.

7. Action Figure Descriptor

The Rectangle of the Action Figure Descriptor is used to hold action names and arguments. Figure 6-8 shows the updated Action Figure Descriptor with its corresponding Node and Diagram Label elements.

Figure 6-8: Action figure descriptor

Because actual actions are already contained in the Rectangle of the StateAction Figure, as shown in Figure 6-7, the Rectangle of the Action Figure Descriptor contains the Outline property which is set to false, therefore, the Rectangle will be displayed without borders. Moreover to keep a reasonable vertical space between actions inside Entry or Exit node, a Default Size Facet child with the Dimension child is added to the Node element of the Action Figure Descriptor, then the Dx property is set to 40 and Dy property is set to 15.

8. Transition Figure Descriptor
The *Transition Figure Descriptor* by default contains a *Polyline Connection* with a *Label* child and a *Child Access*. Moreover, it has corresponding diagram elements represented by a *Connection* and a *Diagram Label* which reference the *Transition* figure for use in the mapping model. The *Polyline Connection* is a solid line with no source and target decoration. To have a transition line with open arrow as shown previously in Figure 6-7, a *Polyline Decoration* child is added to the *Polyline Connection*. Then, the added *Polyline Decoration* is set to form an open arrow by adding *Template Points* children, as shown in Figure 6-9.

![Figure 6-9: Adding Polyline decoration](image)

Then, the *Polyline Decoration* is selected for the *Target Decoration* property of the *Polyline Connection*. The last modification considering *Transition Figure Descriptor* is done to its corresponding *Diagram Label*; specifically, the *Element Icon* property of the *Diagram Label* is set to *false* to keep the diagram surface simple and not cluttered especially the transitions label contains more than one attribute to display.

### 6.1.2 Adding Compartments to the SMC Graphical Definition Model

A *Compartment* is used in the mapping model mappings to allow forming nested figures, as was shown in Figure 6-7. First, a *Compartment* element should be added to the graphical definition model. Then, a *Compartment* has some properties including the *Figure* property which is used to reference a parent figure, as shown in Figure 6-10.
Figure 6-10: Adding new compartment

As shown, a new Compartment is added and named GenericStateCompartment with its Collapsible property set to true to allow hiding a map states when not needed. The Figure property of the Compartment allows selecting a parent figure. However, instead of specifying a specific parent figure, the Figure property is left unset. In this way, the Compartment is reused in the mapping model for several node mappings. Since MainMap and Map figures need a Compartment to add different states, the added GenericStateCompartment is used in the mapping model in both, MainMap and Map Node Mapping. Similarly, a StartState, a State and a DefaultState figure need to include two compartments to allow adding Entry and Exit figures. Furthermore, each Entry or Exit figure may include several actions; thus, a Compartment is added to the Entry and Exit figures to allow adding actual actions. As a result, four compartments are added to the Canvas of the smc.gmfgraph model.

6.2 Developing SMC Tooling Definition Model

The tooling definition model is used to design a diagram Palette which includes creation tools and other elements. For instance, Figure 6-11 shows the developed SMC palette at runtime.
Figure 6-11: SMC palette at runtime

It can be seen that the palette contains tool groups which contain creation tools. For instance, the Maps group contains the Main Map creation tool and a Map creation tool. Tool groups are separated by a separator and they are all collapsible.

Similarly as with other GMF models, the tooling definition model is created by invoking its wizard, Simple Tooling Definition Model. The wizard allows loading the domain model (smc.ecore) and selecting the Diagram Element which in case of the smc.ecore model is SmcModel class. Then, the last page of the wizard is for specifying the basic tooling definition, as shown in Figure 6-12.
Figure 6-12: Specifying basic tooling definition for smc.gmftool model

Figure 6-12 shows that MainMap, Map, StartState, State, DefaultState, StateAction and Action class are selected as nodes and Transition class is specified as a link. By finishing the wizard, the smc.gmftool model is created in the model folder in the GMF project, as shown in Figure 6-13.

Figure 6-13: Basic definition of smc.gmftool model

The wizard creates the Palette with a single Tool Group element containing creation tools for each element selected as a node or link. Each creation tool includes default icons children.

6.2.1 Updating the SMC Tooling Definition Model

Tool groups, creation tools and separators are added to the Palette. Each tool group is meant to contain similar creation tools and therefore, they can be recognized easily and collapsed together,
as shown previously in Figure 6-11. A Separator is added between tool groups and creation tools to have a horizontal line in the generated palette. Figure 6-14 shows the updated smc.gmftool model.

![Updated smc.gmftool model](image)

Figure 6-14: Updating smc.gmftool model

As shown, creation tools are grouped in four tool groups: Maps Tool Group, States Tool Group, State Actions Tool Group and Transition Tool Group. Moreover, instead of defining one creation tool for the state actions type which the tooling definition model wizard produced, a new creation tool is added to represent Exit elements. Similarly, a Creation Tool is added for each transition type. Each Tool Group includes Active, Collapsible, Description, Stack and Title properties. The Active property specifies a Creation Tool within a tool group to be the active tool. The Collapsible property is set to true for each Tool Group to have collapsible tool groups and the default value (false) for Stack property is kept allowing tool groups to respond properly. Description and Title property are entered according to each tool group. Finally, the Palette element has Default, Description and Title properties which are set, as shown in Figure 6-14.
6.3 Developing SMC Mapping Model

The mapping model allows binding the graphical and tooling definition models information to the domain model elements. For instance, the MainMap figure (rectangle) defined in the SMC graphical definition model (smc.gmfgraph) is mapped to the MainMap class defined in the SMC domain model (smc.ecore) and assigned the Main Map creation tool defined in the SMC tooling model (smc.gmftool). Furthermore, OCL constraints can be added in the mapping model to constrain and validate domain model elements. The mapping model is created by invoking the Guide Mapping Model Creation Wizard. The wizard allows loading the domain model (smc.ecore) and selecting a class for Canvas mapping which is the SmcModel class in this case. Proceeding with the wizard, the next page is for loading the smc.gmftool model and selecting its Palette element for the Canvas mapping. Then, the next page is for loading the smc.gmfgraph model and selecting its Canvas element for the Canvas mapping. The last page is for specifying the domain model elements as nodes or links. On this last page, the wizard suggests a list of nodes and links. However, the wizard defaults are modified by removing all nodes except MainMap and Map in the nodes list and also the Transition link in the links list. The reason for removing other nodes is that the nodes list on the wizard represents Top Node References that will be created directly on the diagram surface. Accordingly, the SMC diagram contains only two Top Node References representing MainMap and Map element. This meets the SMC requirements of having maps that encapsulate states and transitions. When updating the smc.gmffmap model, states nodes with their inner nodes and elements will be added as children to these two Top Node References. Figure 6-15 shows the created smc.gmffmap model.
6.3.1 Updating the SMC Mapping Model

Basically, Child References will be added to MainMap and Map Node Mapping to map StartState, DefaultState and State nodes. Moreover, to each state Node Mapping, Child References will be added to represent Entry and Exit Node Mapping. Furthermore to Entry and Exit Node Mapping a Child Reference will be added for Action Node Mapping. Likewise, Link Mappings will be added to represent the five SMC transition types. Therefore, the smc.gmfmap model will contain five Link Mappings. Considering refining and validation the model, OCL constraints will be added to the Node Mappings and Link Mappings to constrain and initialize some features and other OCL constraints will be added in Audit Rules elements to perform various validations, as it is described in Section 6.4. The following sections discuss updating each element in the smc.gmfmap model.
6.3.2 Updating MainMap Top Node Reference

The *MainMap Top Node Reference* references the *MainMap* element that is directly created on the diagram surface. Therefore, it contains a *MainMap Node Mapping*. Figure 6-16 shows the properties of the *MainMap Top Node Reference*.

![Diagram of MainMap top node reference properties](image)

Figure 6-16: MainMap top node reference properties

As shown, the *Containment Feature* property specifies that new instances of *MainMap* class are to be maintained in the *mainMap* containment reference contained in the *SmcModel* class. Moreover, the *Children Feature* property is left unset indicating that the *MainMap* instances will be directly retrieved from the same *mainMap* containment reference specified for saving the instances. Finally, the *Child* property specifies the *MainMap Node Mapping* as a child of the *MainMap Top Node Reference*. Regarding the *MainMap Node Mapping*, Figure 6-17 shows its properties.
Figure 6-17: MainMap node mapping properties

The **Element**, **Diagram Node**, and **Tool** properties settings show that the **MainMap Node Mapping** binds the **MainMap** class defined in the **smc.ecore** model to the **MainMap Diagram Node** defined in the **smc.gmfgraph** model and the **MainMap Creation Tool** defined in the **smc.gmftool** model. Moreover, the **MainMap Node Mapping** has a child named **Feature Label Mapping** which is used to map the **MainMap Diagram label** defined in the **smc.gmfgraph** model to the attribute **name** of the **MainMap** class. Thus, the attribute can be displayed and edited. The properties of the **Feature Label Mapping** are shown in Figure 6-18.

Figure 6-18: Properties of the Feature Label Mapping of the MainMap Node Mapping
As shown in Figure 6-18, the Edit Method and View Method properties are set to the default MESSAGE_FORMAT value, which indicates that the Java MessageFormat class will provide the underlying implementation for parsing, editing and displaying the attribute name [3]. Since the MainMap label has only the attribute name to display, Editor Pattern, Edit Pattern and View Pattern properties are left blank, meaning there are no specific characters or patterns to include. The MainMap Node Mapping in its current state considers only creating a rectangle with a label, as specified in the smc.gmfgraph model, to represent the MainMap class on the diagram surface. However, the MainMap Node Mapping should include states according to the SMC requirements. Therefore, StartState Node Mapping, State Node Mapping and DefaultState Node Mapping are added to the MainMap Node Mapping.

6.3.3 Adding State Node Mapping and DefaultState Node Mapping to the MainMap Node Mapping

Since the State Node Mapping and the DefaultState Node Mapping are mostly analogues to the StartState Node Mapping, the startState Child Reference is copied and pasted twice to the MainMap Node Mapping. Then, properties and constraints are changed for each pasted startState Child Reference to represent the State Node Mapping and the DefaultState Node Mapping.

Moreover, because the SMC Default State must be named Default, a Constraint child and a Feature Seq Initializer child are added to the DefaultState Node Mapping to constraint and initialize the attribute name of the DefaultState class using OCL constraints. At this point, the first Top Node Reference representing the MainMap Node Mapping is complete.

6.3.4 Updating Map Top Node Reference

The Map Top Node Reference references Map elements that are directly created on the diagram surface. Therefore, it contains a Map Node Mapping.

Similarly to the MainMap Top Node Reference discussed previously, the properties of the Map Top Node Reference with its Map Node Mapping child are set to represent the Map class on the
diagram, as required by SMC requirements. Furthermore, since the *Map* element is used to contain a number of states and an optional *Default* state, the *State Child Reference*, the *DefaultState Child Reference* and *GenericStateCompartment Compartment Mapping* defined in the *MainMap Node Mapping* are copied and pasted to the *Map Node Mapping*.

### 6.3.5 Updating Transition Link Mapping

A *Link Mapping* is used to map a *Connection* defined in the graphical definition model and a *Creation Tool* defined in the tooling definition model to a containment reference defined in the domain model. For the *smc.gmfmap* model, five transition *Link Mappings* are added to represent the five SMC transition types.

Basically, when creating the *smc.gmfmap* model, the *Transition* class was selected as a link; therefore, the wizard produced one *Link Mapping*. This generated *Link Mapping* is used to represent the SMC *simple* transition type, as shown in Figure 6-19.

![Diagram](image)

Figure 6-19: Setting a link mapping for simple transition type
As shown, the outgoing containment reference is selected for the Containment Feature property which indicates that transition instances are maintained for each state in the outgoing containment reference.

Moreover, as specified in the Element, Diagram Link and Tool properties, the Link Mapping maps the Transition class to the Transition Connection figure and to the simple Creation Tool defined in the graphical and tooling definition model respectively. Furthermore, the fromState reference is selected for the Source Feature property and the toState reference is selected for the Target Feature property.

To distinguish between transition types and specify this Link Mapping to be used only for the simple transition type, a Constraint child is added and its Body property is set to the OCL constraint (transitionType = TransitionType::SIMPLE). Then, the transitionType feature is initialized by adding a Feature Seq Initializer child and to it a Feature Value Spec child is added with its Feature property is set to transitionType attribute, then a Constraint child is added to the Feature Value Spec with its Body property set to the (TransitionType::SIMPLE) OCL constraint. A second Feature Value Spec child is added to the existing Feature Seq Initializer to initialize the id attribute, defined in the Transition class, to 0. The id attribute is initialized to a unique value for each Link Mapping representing a specific transition type. The value of the id attribute is used when defining Audit Rules to validate transition types and their features. The current settings of the simple transition Link Mapping allows the simple transition to be used as a loopback transition. Moreover, it allows the simple transition to transition to the Default state which violates SMC specification. To prevent the simple transition from being used as a loopback transition, a Link Constraints child is added. Then, to the Link Constraints child a Source End Constraint child is added and for its Body property the (self <> oppositeEnd) OCL constraint is entered. The (self <> oppositeEnd) constraint specifies that the simple transition source state
must not be its target state. Figure 6-20 shows the added Link Constraints with its Source End Constraint child.

![Figure 6-20: Adding link constraints to the simple link mapping]

To prevent the simple transition from transitioning to the Default state, a Target End Constraint child is added to the Link Constraints and for its Body property the (not(self.oclIsTypeOf(DefaultState))) OCL constraint is entered. Finally, the simple transition Link Mapping needs a label to display and edit its information. Hence, a Feature Label Mapping child is added to the Link Mapping and its properties are set as shown in Figure 6-21.
As shown, the Features property specifies that a simple transition name, argument and guard attributes are displayed and edited following the pattern \( \{0\}([1])\{2\} \) specified in the View Pattern and Edit Pattern properties. This pattern conforms to the SMC simple transition syntax, such that the transition name followed by arguments, if any, in parenthesis and a guard in square brackets. The transition actions and their arguments are displayed and edited using a dialog box invoked from the properties view.

6.3.6 Adding Link Mappings for All SMC Transition Types

Four other Link Mappings are added to the smc.gmfmap model to map other SM transition types which are: loopback, Default, push and pop. Therefore, the defined simple transition Link Mapping is copied, and pasted four times to the Mapping element of the smc.gmfmap model. Then, for each pasted Link Mapping, children and properties are modified to represent a specific transition type.

In essence, the Tool property of each pasted Link Mapping is assigned a specific Creation Tool defined in the smc.gmftool model representing a specific SMC transition type. Moreover, the
Constraint child, Feature Seq Initializer child, Link Constraints child and Feature Label Mapping child are all set according to each transition type. For example, the loopback Creation Tool is specified for the Tool property. Furthermore, the Link Mapping is constrained and initialized to be loopback transition. The Link Constraints child of the loopback transition, unlike the Link Constraint of simple transition, contains only Target End Constraint child with the OCL constraint \( \text{self} = \text{oppositeEnd and not( self.oclIsTypeOf(DefaultState))} \). Since push transitions, pop transitions and Default transitions have different syntax definition, the Feature Label Mapping child in their Link Mapping is different and it is set accordingly. For instance, Figure 6-22 shows the properties of the Feature Label Mapping child of the push Link Mapping.

![Resource Set](image)

**Figure 6-22: Feature label mapping of push link mapping**

Moreover, unlike other transition types, the Default transition must be named Default; therefore, the Body property of the OCL constraint in its Link Mapping is set to \( \text{(name = 'Default' and transitionType = TransitionType::DEFAULT)} \). Furthermore, an additional Feature Value Spec child is added to initialize the constrained attribute name to Default.
6.4 Audit Rules

To validate and identify problems in a diagram, GMF allows defining Audit Rules in the mapping model. For instance, audit rules can be added to ensure that the names of all elements are set, and also that the names of map elements are not identical. To add audit rules, first, an Audit Container child is added to the Mapping element of the mapping model to hold the actual Audit Rules. An Audit Rule has six properties: Description, Id, Message, Name, Severity and Use In Live Mode. The Description, Id and Name properties identify the Audit Rule while the Message property specifies the text that is displayed to the user when the model violates the Audit Rule. The Severity property has three selections: INFO, WARNING and ERROR.

The Use In Live Mode property has two options: true and false which determine using the Audit Rule in live or batch validation mode. When using batch validation mode, the user is required to run the validation manually using the Validate option from the Diagram menu. Third, the Audit rule has children that can be added by right-clicking on it and selecting from the popup menu. Basically, the Audit Rule children are used to specify a Target Element and a Constraint. For example, the Target Element can be a Domain Element representing a class in the domain model, a Domain Attribute or a Diagram Element representing a node in the diagram. The Constraint child represents the actual rule and has two properties: Body and Language.

To activate any Audit Rule in a diagram, some properties of the GMF generator model must be set and enabled. Creating the GMF generator model and setting its properties is introduced in Section 6.5. Moreover, the user can disable and enable any Audit Rule by navigating to Window > Preference > Model Validation where a list of defined Audit Rules is found.

6.4.1 Adding Audit Rules to the SMC Mapping Model

It is convenient to separate each related Audit Rule in groups by adding Audit Containers children to the main Audit Container. For the smc.gmfmap model, the first Audit Container is added to
hold Audit Rules for validating the SMC diagram name. Then, an Audit Rule to validate that the
diagram name is set is added, as shown in Figure 6-23.

![Figure 6-23: Properties of the “unset SMC diagram name” audit rule](image)

Since the added “unset SMC diagram name” Audit Rule should target the SmcModel class, a
Domain Element Target child is added with its Element property set to SmcModel class. To finish
the “unset SMC diagram name” Audit Rule, a Constraint child is added to it with the OCL
constraint (self.name.size( ) > 0) in its Body property.

Considering that Default, pop and push are reserved words, Audit Rules are added to validate that
the SMC diagram name is not set to one of the these reserved words. In this case, the live
validation is used to alert the user immediately if there is any violation. For example, the
(self.name <> 'Default') OCL constraint states that the attribute name must not be set to Default.

Similarly, Audit Rules are grouped in Audit Containers and added to validate the names of other
elements in the SMC diagram. More precisely, the Default transition name is validated to be
always set to Default, and other transition types names are validated not to be set to Default.

Moreover, because a pop transition includes a popArg attribute which should be set and include a
transition name defined in the SMC diagram, two Audit Rules are added. The first Audit Rule is
for validating that the popArg attribute is set; therefore, it includes the OCL constraint
(self.transitionType = TransitionType::POP implies self.popArg.size() > 0). The second Audit Rule is for preventing setting the popArg attribute for all other transition types, thus, it defines the OCL constraint (self.transitionType <> TransitionType::POP implies self.popArg->isEmpty()).

Similarly, the push transition includes the prePushedState attribute when using its new syntax definition, but it is not required when using the initial syntax. Thus, two Audit rules are added.

The first Audit Rule operates in the batch validation mode and checks that the prePushedState attribute is set for the push transition and if it is not set, the Audit Rule issues a warning indicating the use of the initial syntax of the push transition, as shown in Figure 6-24.

![Figure 6-24: PrePushedState batch validation warning](image)

The second Audit Rule is for preventing setting the prePushedState attribute for all transition types except push transition. This Audit Rule operates in the live validation mode, as shown in Figure 6-25.
Moreover, since transition types are displayed in the transition properties view, it is appropriate to
prevent changing the transition type after issuing them. For this reason, each transition type was
assigned a unique id when defining Link Mappings in the smc.gmfmap model. For instance, the
Audit Rule for preventing changing the simple transition type is captured by the
(self.transitionType = TransitionType::SIMPLE implies self.id = 0) OCL constraint.

Furthermore, Audit Rules are added to prevent modifying the property toState from the properties
view of transitions. The reasons for preventing changing toState from the properties view are:
first, toState is set automatically when issuing transitions. Second, a transition to the Default state
is not allowed by OCL constraints applied to the transition types and changing the toState
property from the properties view to the Default states causes an error to occur. Therefore, the
(not(self.toState.oclIsTypeOf(DefaultState))) Audit Rule is added to prevent setting the toState
property to any existing Default state in the SMC diagram. Moreover, to prevent modifying
toState property of a loopback transition, a new Audit Rule is added with OCL constraint
(self.transitionType = TransitionType::LOOPBACK implies self.toState = self.fromState).
Conversely, to prevent setting toState equal to fromState for other transition types as to be
loopback transition, a new Audit Rule is added with OCL constraint (self.transitionType <>
TransitionType::LOOPBACK implies self.toState <> self.fromState).

Furthermore, two Audit rules are added to ensure that a state action type is not modified after it
was added. Specifically, the (self.actionType = ActionType::EXIT implies self.id = 1) and
OCL constraints prevent changing Entry and Exit type.

It should be noted that different transition types and states action types in SMC diagram are mainly created using creation tools defined in the SMC palette. Hence, transitionType and toState, as well as, actionType features could be set to Read only when developing the smc.genmodel model (EMF generator model); in that case, it would not be necessary to define Audit Rules preventing modification of these features from the properties view. Nevertheless, the reason for not disabling these features in the properties view is that when using the SMC model instance editor generated using EMF, the only way to set element properties is from the properties view. Moreover, GMF generator model does not include options to set the properties to Read only as it is supported by EMF generator model properties view.

Finally, some defined OCL constraints could be grouped into one constraint using Boolean operators; however, for the sake of simplicity and clarity, especially, considering the ability to report precise error messages to the user, most defined OCL constraints are kept simple.

### 6.5 Developing SMC GMF Generator Model

In order to allow generating the SMC diagram editor code, the smc.gmfmap model and the smc.genmodel model (EMF generator model) information need to be bound and transformed to a model called GMF generator model. The GMF generator model is created by invoking the GMF generator model creation wizard which allows loading the mapping model and EMF generator model and creates the GMF generator model.

### 6.5.1 Updating SMC GMF Generator Model

The created smc.gmfgem model includes elements with properties. The properties allow providing more information for code generation. For instance, the Copyright property allows providing copyright text. Importantly, the SMC mapping model (smc.gmfmap) defines Audit Rules and to
activate them, validation properties of the *Gen Diagram element* in the *smc.gmfgen* model need to be enabled. Specifically, the Validation Enabled property and the Validation Decorators property are set to `true`. These two properties have corresponding priority properties which are Validation Provider Priority and Validation Decorator Provider Priority, respectively and they are set to Medium. Finally, the Live Validation UI Feedback property is set to `true`. Figure 6-26 shows the setting of the discussed validation properties.

Figure 6-26: Enabling validation properties in the smc.gmfgen model

Moreover, by default the Creation Wizard Category ID property of *Gen Diagram element* is set to *(org.eclipse.ui.Examples)* which indicates that the SMC diagram creation wizard will be placed under the Examples directory.

To have the SMC diagram creation wizard placed in the same directory as the EMF SMC model creation wizard which is set as discussed in Chapter 5, Section 5.4, the property Creation Wizard Category ID is changed to "ca.queensu.cs.afmg.smc.Wizard.category.ID". Moreover, the *Gen Plugin* element has properties for naming and identifying the diagram plug-in. For instance, the
Name property is set to SMC Diagram plugin, the Provider property is set to www.cs.queensu.ca and the Printing Enabled property is set to true.

6.6 Generating SMC Diagram Code

The last step in the GMF process to create a graphical editor involves the generation of the diagram code. The code is generated from the smc.gmfgen model. Then, a new plug-in project is generated and created in the workspace. Figure 6-27 shows the SMC Diagram plug-in project expanded in the Package Explorer view.

![Figure 6-27: SMC diagram plug-in code in the explorer view](image)

The SMC diagram plug-in code contains Java code that allows creating, editing and viewing the SMC diagram elements. It also provides services for handling the SMC diagram preferences, properties view and other common operations. The generated code also contains plug-ins files which identify the SMC diagram code as an Eclipse plug-in and declare its dependencies.

Figure 6-28 shows the SMC diagram editor being used to draw the subway state machine example given in Chapter 3, Section 3.5.
Moreover, Appendix D explains in details the execution and use of the SMC diagram editor.
Chapter 7
SMC Diagram Editor Validation

As mentioned in the previous chapter, Section 6.4, the SMC diagram editor defines Audit Rules in the mapping model (smc.gmfmmap) to validate and identify problems in a diagram. As appropriate, some of the defined Audit Rules are set to operate in the live validation mode and other set to operate in the batch validation mode. The live audit rules are invoked automatically, whereas the batch ones are invoked manually by running the Validate option found on the Diagram menu. Then, any errors will be shown in the Problems View categorized according to their type. Elements on the diagram canvas that have these errors will be decorated with icons indicating the error type. Moreover, live validation has other options including showing live errors in a popup dialog or a Console. These options are accessible by navigating to Window > Preferences > Model Validation where the Constraints category is also found listing all SMC diagram Audit Rules and providing information for each constraint, as well as, allowing disabling and enabling any constraint.

7.1 Validation Examples

1. Figure 7-1 shows an example for testing some Audit Rules operating in the batch validation mode.

![Figure 7-1 SMC batch validation examples](image-url)
As shown, eight errors and one warning are detected. The first error reported in the Problems view indicates that two states have the same name. Uniqueness of naming applies also to all maps. The next six errors report unset names for various elements including: a map name, SMC diagram name, start state name, action name, state name and transition name. The last error reports an unset popArg attribute of a pop transition. Finally, one warning is detected indicating an unset prePushedState attribute of a push transition. Moreover, error descriptions shown in the Problems View can be also seen on the canvas by placing the mouse over any error icon. Since the MainMap element and the StartState element are required, they must be created on the diagram; otherwise, the validation issues an error when manually validating the diagram.

2. The SMC diagram live validation mode prevents the user from providing unacceptable data or assigning wrong information according to specific element rules. For instance, when attempting to enter Default as a name for the MainMap element, a dialog box, as shown in Figure 7-2 pops up indicating the violation of the integrity of the model and preventing this naming operation.

![Live Validation](image)

Figure 7-2 SMC live validation prevents assigning Default as a name for the Main Map

Similarly, the Default state and Default transition are validated to be always named Default. Furthermore, the live validation prevents setting the prePushedState and popArg attributes for all transition types except for the push transition and the pop transition. Figure 7-3 shows the SMC live validation preventing setting the popArg for non pop transition type.
For the transition types and state action types, the live validation rejects changing the types from the Property View or editing them on the diagram canvas after they have been created. In other words, once a transition or state action has been created, its type cannot be changed. Therefore, if required, they should be deleted. A screen shot in Figure 7-4 shows the SMC live validation preventing changing a transition type from the Property View.

Figure 7-3 SMC live validation prevents setting popArg for non pop transition

Figure 7-4 SMC live validation prevents changing a transition type
Likewise, the toState feature included in the properties view of all transitions is validated in the live mode. Three conditions are considered. First, it cannot be set to the Default state. Second, it cannot be set to form a loopback transition for all transition types except the loopback transition. Third, it cannot be changed when a transition type is loopback. As shown in Figure 7-5, the SMC live validation rejects changing the toState to Default State.

Figure 7-5 SMC live validation prevents changing toState
Chapter 8
Developing SMC JET Transformation

This chapter discusses the development of a JET transformation plug-in project to transform an SMC model instance created using the SMC diagram editor to SMC source code and a Java skeleton class. Section 8.1 gives an overview of the SMC JET transformation plug-in project and Section 8.2 discusses the creation of the SMC JET transformation plug-in project with its templates and configurations, and it also provides some JET tags examples.

8.1 SMC JET Transformation Overview

When generating code from a model, Model-to-Text (M2T) technologies are used. In this project, the Java Emitter Templates (JET), a component in Eclipse M2T project [10], is used to transform an existing SMC model instance created using the SMC diagram editor to SMC source code and a Java skeleton class. Figure 8-1 shows the JET transformation development process.

![Figure 8-1: JET transformation development flow](image)

As shown, a JET transformation project is created to add JET templates that create resources and extract data from an SMC model instance. Moreover, the JET transformation project includes plug-ins and properties files which are set to allow loading the SMC model instance. Then, the JET transformation plug-in project is run as an Eclipse plug-in.

By loading an existing SMC model instance as the input for the SMC JET transformation plug-in, the model is inspected and a folder named *smc* is created in the same project that contains the SMC model instance. Furthermore, the SMC JET transformation project includes templates to
perform error checking over the input model and outputs transformation process to the Eclipse Console. The outputted information shows information about each element being transformed and any errors or warnings. For instance, it outputs information of each map being processed such as a map name and location or index in the XMI file. Similarly, information on each state being transformed is outputted. If there are any errors or warnings, the JET transformation also reports these issues and gives information on how to fix them.

As a result of the transformation, two files are generated and saved in the created smc folder. The first file has the same name as the name of the root element of the SMC diagram with .sm extension. This file contains SMC source code. The second file also has the same name as the name of the root element of the SMC diagram and has extension .java. This file contains a Java skeleton class used to implement transitions actions. The generated SMC source code considers Java as a target language.

Then, the generated SMC source code can be fed as input into the SMC compiler to generate Java classes implementing the state machine created with the SMC diagram editor and the generated Java skeleton class can be used to implement transitions actions.

### 8.2 Creating JET Transformation Project

JET2 [24], an enhancement of the original JET that EMF by default uses to generate code, is used for the development of SMC transformations. Therefore, a new JET Transformation project is created. Figure 8-2 shows a new JET transformation project created in the workspace.
The SMC JET transformations project defines nine templates which can be classified into four categories based on their task. The first category is a main template which is responsible for creating resources and invoking other templates. The second category is templates which validate the SMC model instance and output information to the Eclipse Console. The third category is templates which are responsible for transforming the SMC model instance to the SMC code. The fourth category is a template that generates the Java skeleton class with the transitions actions methods signature. Figure 8-3 shows the SMC JET transformations project with the nine templates are added.

Figure 8-2: SMC JET transformation project

Figure 8-3: SMC JET transformation templates
The following numbered list gives a description of each template. The templates are listed in their order of execution.

- The first category

1. *main.smc.jet* is the starting template used to create resources and it invokes the following templates.

- The second category

2. *info.warn.error.main.smc.jet* template validates the *MainMap* element of the SMC model instance; yet, it also performs other validation on the structure of the model. Moreover, it outputs to the Eclipse Console information about the transformation process and any existing warnings or errors.

3. *info.warn.error.map.smc.jet* template; similarly to the previous template, this template validates any existing *Map* elements in the SMC model instance and outputs transformation process to the Eclipse Console.

- The third category

4. *include.smc.jet* template is the last template invoked by *main.smc.jet* template and it service as a medium template to invoke and process all of the following templates.

5. *verbatim.smc.jet* template is responsible for transforming the *verbatim* section of the SMC model instance, if exists.

6. *set.smc.jet* template is used to set the SMC source code to target the Java programming language. It should be noted that some other information, such as importing specific Java classes or providing a package name may be added by the user after the transformation is complete. Therefore, this template includes a user region that generates comments to alert the user to include other SMC code if required. Furthermore, when targeting different
programming languages than Java, the difference in SMC source code is mainly on this section, hence, the transformed SMC source code can be used to target any supported language by modifying this section.

7. `mainMap.smc.jet` template handles the task of transforming the `MainMap` element of the SMC model instance to the corresponding SMC source code.

8. `map.smc.jet` template is for transforming any existing `Map` elements in the SMC model instance to the corresponding SMC source code.

- The fourth category

9. `appclass.smc.java.jet` template generates a Java skeleton class. More precisely, it transforms any transitions actions with their arguments to Java signature methods and instantiates the `Context` class that is to interact with. The generated methods should be implemented by the user.

Because the SMC JET transformation includes templates that perform error checking and validate the SMC model regardless of the constraints defined on the SMC diagram editor, the transformation can be used to validate and generate SMC source code for any given SMC model that is written as XML document based on the structure, elements and attributes specified for the SMC model. Moreover, outputting transformation information to the Eclipse Console not only assists in understanding the transformation, but it also helps to trace any existing errors, especially with large models. Furthermore, dividing templates to be nine templates is appropriate because each template performs a specific transformation which makes the SMC JET transformation project more readable and easier for future maintenance.

### 8.3 Configuring SMC JET Transformation

By default, JET transformation accepts an XML standard document as input; however, JET transformation can be configured to allow loading EMF models and other resources using several
model loaders [47]. Since the SMC model instance is based on EMF’s Ecore model and is persisted by default as XMI file with .smc extension, the SMC JET transformation is configured to enable loading SMC model instances as XML document. Therefore, the (org.eclipse.jet.xml) id is specified for the modelLoader attribute and the smc is provided for the modelExtension attribute. The modelLoader (org.eclipse.jet.xml) is one of the JET defined model loader classes that allow loading models into the memory as standard XML files and the smc is the specified extension for SMC models when developing SMC models. In the JET transformations project, the modelLoader and modelExtension attributes are located in the Extensions tab of the plugin.xml file under transform element of the org.eclipse.jet.transform extension, as shown in Figure 8-4.

![Figure 8-4: Setting a model loader and a model extension](image)

### 8.4 JET Tags examples

This section provides some JET tags examples taken from the SMC JET transformation templates.

First, Table 8-1 shows some JET tags used in the main.smc.jet template.
Table 8-1: main.smc.jet template tags example

The first tag “setVariable” is a control tag and creates a variable named “root” using the attribute “var” and sets its value to the result of the “/*” XPath expression provided with the attribute “select”. The “/*” XPath expression finds the root element of the input model, which in this case is the SMC model. When using the “root” variable with other tags, it is referenced by the $ sign ($root). The two “include” tags invoke and process the specified templates; then, they include the result in the output of the current template.

Moreover, the first “if” tag (<c:if test="$root[@name and @name!='']">) tests if the name attribute of the root element exist and set using XPath expression with a predicate. The workspace tag “folder” creates a folder named “smc” in the same project that contains the SMC model instances using the dynamic XPath expression “{$org.eclipse.jet.resource.project.name}”. The “{ }” braces refer to a dynamic XPath expression and the variable “$org.eclipse.jet.resource.project.name” is a pre-defined variable refers to a project name from which JET transformation are run. The “file” tag writes the output of the include.smc.jet template by creating a file with “.sm” extension using the “{camelCase($root/@name)}” Xpath expression. That is, the XPath expression “$root/@name” gets the value of the attribute name of
the root element and the XPath function “camelCase” uppercases its first letter. Similarly, the second “file” tag writes the output of the “appClass.smcl.ave.jel” template to a file with the “.java” extension. These two files are created in the created “smc” folder.

Second, Table 8-2 shows JET tags used in the “info.warrn.error.main.smcl.jet” template.

Table 8-2: info.warn.error.main.smcl.jet template tags example

As shown in Table 8-2, JET tags with XPath expressions are used to iterate over maps elements of the SMC model instance in order to access and ensure the uniqueness of the value of the attribute name. Accordingly, it outputs transformation process information and reports any identical map names.

Third, As a last example, Table 8-3 shows some JET tags used in “mainMap.smcl.jet” template.
Table 8-3: mainMap.smc.jet template tags example

JET tags shown in the Table 8-3 transform a push transition issued by the start state. Appendix C demonstrates the execution and the use of the SMC JET transformation.

Figure 8-5 shows SMC JET transformation plug-in being used to transform the SMC model instance, created by SMC diagram editor, of the subway turnstile state machine example given in Chapter 3, Section 3.5, Table 3-2.
As shown on the Explorer view on the right side of the figure, the SMC JET transformation generates a folder named `smc` and two files, `Turnstile.sm` and `Turnstile.jave`. Moreover, SMC JET transformation produces transformation process information, as shown at the bottom of the figure.

On the diagram surface, it can be seen that after performing SMC batch validation, two warnings are reported indicating the use of the initial syntax of the push transition. These warnings are also reported on the Problem view.

The JET transformation also reports these two warnings on the transformation process information, as shown in Figure 8-6.
Furthermore, the generated `Turnstile.sm` file contains the SMC source code for the subway turnstile example. Due to the length of the file, Figures 8-7, 8-8 and 8-9 show the file divided among these figures.

Figure 8-7: part one of SMC generated source code for subway turnstile state machine example
Figure 8-8: part two of SMC generated source code for subway turnstile state machine example

Figure 8-9: part three of SMC generated source code for subway turnstile state machine example

Moreover, the generated Turnstile.java file contains a Java skeleton class defining method signatures of the transition actions, as shown in Figure 8-10.
Figure 8-10: Java skeleton class for transitions actions of subway turnstile state machine example
Chapter 9
Discussion

The main objective of this work is to highlight some weaknesses and strengths of GMF based on the use and experience obtained while developing a domain specific modeling language in a form of a diagram editor for SMC. As stated in the motivation and contribution of the thesis in Chapter 1, the GMF evaluation is based on certain criteria. These criteria are discussed in this chapter in details. It is important to note that our discussion of the strengths and weaknesses is rather informal and based solely on the observations of the author during the described case study. The significance and general validity of the evaluation is therefore limited. A more thorough evaluation of GMF is left for future work.

This chapter discusses the findings in the context of the thesis’ objectives. Section 9.1 discusses the SMC modeling and transformation tool. Section 9.2 presents the weaknesses of GMF while Section 9.3 gives the strengths of GMF. Section 9.4 summarizes the evaluation of GMF and Section 9.5 criticizes the problem domain (SMC). Finally, Section 9.6 states the future work.

9.1 SMC modeling and transformation tool

Using GMF, a rich and fully functional SMC diagram editor was created and using JET, SMC source code and a Java skeleton class was generated based on the SMC model instance. The combination of GMF and JET has led to the SMC modeling and transformation tool that is practically valuable. Figure 9-1 abstracts the process of developing and using the SMC modeling and transformation tool.

Figure 9-1: SMC modeling and transforming tool developing and using process
Although the SMC modeling and transformation tool allows the input of SMC state machines graphically, some improvements are needed as it will be discussed in the Future Work, Section 9.6. Moreover, some aspects of SMC are found hard to represent graphically. These aspects and other SMC limitations are presented in Section 9.5.

9.2 GMF Weaknesses

The following subsections address GMF weaknesses based on the research experience with respect to the GMF 2.1.0 release.

9.2.1 Documentation

While the “Eclipse Modeling Project: A Domain-Specific Language Toolkit” reference [3] provides valuable and detailed information on GMF development, at the beginning of this research, GMF documentations provided limited help specifically on details and meaning of several properties and features associated with GMF models, as well as, meaning of error messages produced by GMF which imposes time consuming and continuous generate and test cycles. The lack of GMF documentations was pointed out by researchers, as presented in the Related Work in Chapter 2.

9.2.2 Usability

Using GMF means developing, generating, updating and maintaining several models. This process results in poor usability and large amounts of low level work. In particular, it involves using several long model creation wizards that require significant time to complete initial definitions, and then manually fixing, adding elements and features to obtain the final definitions. Updating models is also done via these wizards. All of which results in a painful usability experience. For instance, when major changes are made on the domain model, all other models need to be updated repeatedly. Figure 9-2 shows the typical cycle of required for model creation and update activities.
Moreover, the property view of all models lacks a collective ‘undo’ function. Consequently, restoring the default values after a customization needs to be done individually.

Furthermore, the EMF generator model provides good support to its elements properties via the properties. For instance, properties can be set as editable, read only or not displayed. However, the GMF generator model does not provide such support for element properties from the properties view. This requires, for instance, the addition of some OCL constraints in the SMC mapping model to prevent modifying transition types, action types and toState property from the properties view.

Moreover, the GMF runtime appears to have a deficiency in handling layouts. For instance, figures drawn on the diagram surface appear to respond inappropriately to the user input; especially, to figures that are contained in compartments. In our case study, we experimented with different layout types, but the layout always seemed to be imperfect.

Although elements such as menus and toolbar, as shown in Figure 9-3, are included among available children in the tooling editor model, they have not been implemented which is confusing especially given the lack of documentation.
Additionally, the GMF graphical definition model provides ready to use common shapes such as rectangles and lines. Yet, specific, complicated and decorated shapes need to be manually created either by adding template points which is not very practical or by manually writing classes to implement the shapes, and then reference them in the graphical definition. The graphical definition model should be more extensible to allow adding complex figures. Moreover, three dimensional figures are not supported.

Considering the mapping model, first, its creation wizard does not map all domain model elements correctly as nodes or links, therefore, the basic mapping model definition dialog on the wizard needs to be corrected and set manually. Moreover, the generated basic mapping model definition requires major settings and modifications since the wizard does not map some features such as Feature Label Mapping, which is the feature most likely to be used, for node mappings and link mappings.

Second, link mappings in the mapping model are not as well supported as the node mappings. More precisely, while node mappings can have Child Reference children, link mappings cannot. Therefore, mappings are limited and restricted because nodes can be mapped only to classes and
links to associations. As a result, a link mapping cannot directly have other elements as its children which imposes a significant limitation. For example, in the SMC model instance, it would have been helpful to be able to have transition actions persisted as elements with attributes under the *outgoing* element in the XMI serialization. However, since this was not possible, transition actions were defined as multiple-valued attributes in the *Transition* class and persisted as text elements which complicated the SMC JET transformation task. It also leads to poor usability of the SMC diagram editor because transition names and arguments are edited in separate dialogs and have no relationship except that they are listed as text elements inside an *outgoing* element. Figure 9-4 shows an example of transition actions representation in an SMC XMI model instance.

```xml
<state name="Done" id="0">
   <outgoing name="poped" transitionType="pop" toState="Run" id="1">
      <actionName>close</actionName>
      <actionName>end</actionName>
      <actionArg id="name" name="someone"/>
      <actionArg id="reset"/>
   </outgoing>
   <entry>
      <action type="Input"/>
      <actionArg id="id" name="send"/>
   </entry>
   <exit transitionType="Exit" id="1">
      <actionArg id="end" name="Done"/>
   </exit>
</state>
```

Figure 9-4: Persisting transitions action name and argument in XMI file

From Figure 9-4, it can be seen that there are two text elements (*actionName*) representing actions and two text elements (*actionArg*), which has some attributes (*id, name, reset*) representing action arguments. For instance, removing the first action name (*Close*) will cause the second action name (*end*) to take the first action argument (*id, name*) in the JET transformation. Therefore, since actions and their arguments are not explicitly linked, deleting a transition action also requires removing its arguments. Otherwise, the SMC JET transformation will assign arguments to transition actions by order and produce the wrong result.


9.2.3 Updating

First of all, the Ecore class diagram editor provided with GMF does not synchronize as expected with the Ecore model editor; especially, when changes are first made in the Ecore model editor, the Ecore class diagram editor appears to crash. This has been experienced during developing and updating the SMC domain model which requires regenerating the Ecore class diagram from the Ecore model instance and redo all elements arrangement manually and recover lost data.

Furthermore, for some changes, the mapping model does not update automatically when opening and changing other models elements and properties within its editor, as it is stated and recommended in the GMF documentations. For example, renaming attribute names in the domain model classes and then updating the mapping model, requires manually setting the Feature Label Mapping elements for each node or link mapping.

Moreover, some changes to the properties of the elements of the GMF generator model are not preserved when regenerating the model.

For instance, the Validation Provider Priority property always defaults to its default value “low” when regenerating and needs to be reset. Additionally, validation properties in the GMF generator model are set to false by default, whereas they should be automatically set to true whenever a mapping model includes audit rules.

9.2.4 Validation

The mapping model allows restricting and initializing values of features of the domain model elements in the form of constraints and it supports validating domain model elements and features in the form of Audit Rules expressed as constraints. However, checking and debugging the OCL constraints lacks good support and developers have to use external tools or run the diagram editor to detect faulty OCL constraints. For instance, if the developer inputs accidentally a not well formed OCL constraint, it may be very difficult to determine the source of the problem, especially; if the model contains many OCL constraints.
9.2.5 Automation and Generation

The update of models and regeneration of code typically requires use of several GMF wizards to load models and reflect changes. Java projects in Eclipse, for instance, support “automatic builds” and it would be a very useful feature for GMF models too whenever it is possible.

9.2.6 Abstraction and Separation

Abstraction is a well known practice used to ease and manage the complexity of any system. With the MDA methodology of separating system specifications from its implementations, the GMF separation of concerns has led to several weaknesses in its usability as discussed in the previous sections. Therefore, GMF should adapt and find the right balance level between abstraction and separation.

9.2.7 Integration

Based on the previous GMF weaknesses, integration of GMF models is, thereby, affected. For instance, updating models is not well integrated and a large amount of manual work needs to be performed. Moreover, improved integration is needed in the upper level that is between GMF and other Eclipse modeling frameworks and tools that GMF uses. Eclipse modeling unification and integration was addressed widely by Eclipse community and researches, as it was discussed in the Related Work, Section 2.6.

9.2.8 Maintainability

Having several models in the GMF development process leads to information to be spread out and duplicate which requires more effort when adding and maintaining elements across these models. Therefore, the likelihood of having errors and bugs in the development is high. Thus, it appears that GMF violates the software development principle of “Do not Repeat Yourself” or DRY programming [48].

99
Moreover, the Ecore class diagram editor lacks the functionality of reordering attributes. Besides, default names in all models such as names of, nodes, links, figures, labels and diagram labels are alike and need to be changed to be recognized clearly. The naming aspect is important to prevent errors especially when updating and maintaining.

Last but not least, when developing complicated GMF diagram editors, it is likely that manual customization of generated code is necessary. However, this complicated maintenance because understanding of generated code is required. In fact, customizing generated code is the main obstacle facing the success of software modeling paradigm in general.

9.2.9 Productivity
Based on all previous weaknesses, GMF has reasonably good productivity only when the user has extensive experience and familiarization with the development process, as well as, some technical details. Even though, large amounts of low level and repetitive work are still necessary and cast doubt over the GMF development process.

9.3 GMF Strengths
The goal of GMF is to simplify the creation of GEF editors. Despite the discussed GMF weaknesses, GMF succeeds in hiding the complexity of GEF to the extent that developers need not to know any GEF implementation details. Moreover, GMF generates readable, well structured, customizable, and easy-to-trace code. Additionally, the GMF runtime provides good support for common operations and services. Furthermore, the mapping model provides a convenient structure for adding and defining OCL constraints. Several Eclipse projects and tools, as some discussed in the Related Work section 2.6, Chapter 2, have been built on top of GMF indicating that GMF provides essential infrastructures to Eclipse graphical modeling.
9.4 Summary of Strengths and Weaknesses

As presented previously in this chapter, several factors were taken as evaluations criteria on GMF. The evaluation is based on the observation and experience gained while using GMF. Table 9.1 summarizes the evaluation findings.

<table>
<thead>
<tr>
<th></th>
<th>Very good</th>
<th>Good</th>
<th>Poor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Usability</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Model Updating</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Validation</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Model Automation</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Abstraction vs. Separation</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Integration</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Maintainability</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Productivity</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Code generation</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Generated code quality</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Runtime support</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Table 9-1: Summary of Strengths and Weaknesses

As shown in Table 9.1, most of the poor attributes lie on the area of the overall GMF design which mainly results in a painful usability user experience.

9.5 The Case Study (SMC) Limitations

On the one hand, the State Machine Compiler (SMC) provides automatic code generation in several programming languages for implementing state machines described in a simple textual file. On the other hand, SMC requires interaction with the generated code by implementing transitions actions in an application class for a specific programming language. As a result, the
application class needs to be completely replaced when targeting a different programming
language in addition to some changes in the SMC source code.

Moreover, SMC allows using push transitions and pop transitions to transition between states in
the same map, and in turn, it allows simple transitions to transition between states in different
maps. Therefore, these transition types do not really seem to be fundamentally different.

Furthermore, the pop transition does not specify a next state, but it assumes returning to the state
that issued the push transition by accepting an argument representing a transition name which
thereby is transitioned to and processed. Allowing transitions to issue other transitions cause
errors to occur. Instead of allowing pop transitions to issue other transition, and if there is any
actions that need to be checked and performed after executing the pop transition, it would be
better to explicitly specify the next state and include these actions in its Entry actions.

Because the SMC pop transition definition does not include a next state, it is hard and not
appropriate to express it graphically. That is, in this case, the pop transition line in the diagram
editor will have no target figure (next state). Thus, the pop transition graphically is modeled to
have a target state according to the transition name that it includes in its argument, but when
generating SMC source code, the pop transition transformed without a target state as required by
SMC.

Even though the SMC documentation does not state that transitioning to a Default state is not
allowed, the SMC compiler issues an error reporting “transition to the Default state is not
allowed”. Consequently, constraints in OCL were added to the SMC mapping model to prevent
issuing transitions to the Default state.

Furthermore, having one keyword used for different elements is considered to be a bad design
choice; however, SMC uses Default as a keyword for the Default state and Default transition. As
a result, readability of the SMC source code decreases. Moreover, SMC relies on the Default
transition and Default state to provide a mechanism of handling faults in state machines. Nevertheless, defining Default state and Default transition are optional, and therefore, SMC may issue “transition undefined exceptions” which reduces the reliability of the SMC generated code.

Last but not least, SMC compiler does not generate correct “dot” files that are used to generate Graphviz diagrams [49]. For instance, when attempting to generate a dot file for the SMC subway turnstile state machine example provided in Chapter 3, Section 3.5, Table 3-2 using the following SMC command line: “Java –jar smc.jar –graph –glevel 2 C:\Turnstile.sm” and feeding the generated Turnstile.dot file to the Graphviz, the following diagram was obtained.

![Graphviz Diagram Produced From SMC Generated Turnstile.dot File](image)

Figure 9-5: Graphviz diagram produced from SMC generated Turnstile.dot file

It can be seen that push transitions are not handled correctly.

### 9.6 Future Work

As with any research project, further improvements and researches are possible. For instance, several other evaluation criteria, such as performance, reliability, scalability and testability can be investigated on GMF.
Although several constraints in OCL were added, improvements are needed to improve the integrity of the SMC diagram editor. Moreover, the appearance of the SMC diagram editor would be more professional if the default icons of the SMC diagram editor, the SMC creation wizard, and the palette creation tools were replaced by other more expressive icons. Likewise, lines representing transition types could be improved by giving each type a different target or source Polyline Decoration.

Another possible improvement is that the SMC diagram editor plug-in projects can be migrated to the Eclipse Modeling Amalgamation Project which provides better packaging and integration, as well as, easing future updates and maintenance.

Moreover, as discussed in the Related Work, Chapter 2, Section 2.6, the GEMS project provides superior enhancement to the development of graphical modeling tools; therefore, the entire development of SMC diagram editor could be shifted to the GEMS project.

Finally, building an Eclipse plug-in for the SMC compiler and developing a textual editor for SMC using Eclipse Textual Modeling Framework (TMF) project [50] will provide alignment with the SMC diagram editor and enable SMC users to compile the generated SMC source code inside Eclipse which facilitates the use of SMC itself.
Chapter 10
Conclusion

An SMC Eclipse modeling and transformation tool has been developed, which allows graphical
input of SMC state machines and automatic generation of SMC source code. The graphical
representation of SMC is based on GMF and the SMC source code generation is based on a JET
transformation. Although the tool still needs some enhancement, as stated in the Future Work
Section 9.5 in the previous chapter, it satisfies the SMC requirements well. On the other hand, it
allows us to explore the GMF development and identify some of its weaknesses and strengths,
which is the main objective of this research.

10.1 Summary

In addition to demonstrating the creation of the SMC modeling and transformation Eclipse plug-
in tool using GMF and JET, several technologies and topics have been covered, particularly, the
Eclipse Modeling Framework (EMF), the Graphical Modeling Framework (GMF), the Java
Emitter Templates (JET) transformation and the State Machine Compiler (SMC) with all related
tools and languages such as OCL and XPath.

The results of the research were discussed in the previous chapter. Moreover, the technical use of
the SMC modeling and transformation tool is given in the Appendix D.

Whereas the requirements of SMC were graphically met quite well, experience show that not all
aspects of a domain are appropriate to be represented graphically; instead, a combination of
textual and graphical editors with further M2T or M2M transformations will enhance the use of
domain specific modeling tools. Moreover, The JET was found to be easy to learn and integrate
with GMF models.

While specificity of development is an important DSM development principle, the main issue
with current Eclipse modeling practice is that it involves using several frameworks and tools with
little unification and integration. In particular, GMF’s integration with other Eclipse modeling
technologies that it relies on is insufficient. Recent added Eclipse modeling projects, such as the Amalgamation project and the GEMS project are addressing these issues.

10.2 Thesis Conclusion

In conclusion, GMF is a powerful and intensive graphical modeling framework. One of the advantages of GMF is that it hides the complexity of the implementation of the GEF-based graphical editors which is the motivation behind the GMF project. Moreover, GMF generates high quality code and provides good runtime support and services.

However, GMF development process is complex especially for new developers and has major weaknesses mainly in the areas of usability, updating and maintainability which results in large amounts of low level and repetitive work.

Some of these weaknesses have been addressed by researchers and approaches have been introduced to improve GMF development process. Nevertheless, most of the proposed approaches attempt to add more layers and models atop of GMF itself.

Therefore, the GMF development process and models structure should be reengineered given that the development of all models should be centered and integrated in one editor that could be a generic wrapper and mapping model from which the development is started, edited and updated and at the same time preserving the separation of abstract and concrete syntax to the most extent.

10.3 Outlook

The MDA principle of developing PIMs, and then transforming to PSMs appears to be less feasible especially when having several models as with GMF. Therefore, the separation of concerns should have the right balance. In this context, several approaches have been introduced to overcome this issue by adding concrete annotations into the domain model structure or adding graph transformation models to the GMF development process, as well as, building other
frameworks and tools on top of GMF which makes the entire framework more complex in many respects.

To improve the GMF development process, the author’s point of view is to unify and wrap all models by providing a generic-wrapper and mapping model in which the development should be started, edited, updated, synchronized and maintained, yet at the same time the separation of abstract and concrete syntax should be kept to the most extent.

To explain, to hide some of its usability and low level work weaknesses, GMF provides a dashboard from which all model creation wizards can be only invoked, and then proceed normally as they were invoked from the File >> New menu. Thus, this is an anti-pattern method and it does not provide a proper solution.

Moreover, as mentioned in the Related Work Section 2.6, some approaches have suggested adding concrete annotations and properties to the domain model; yet, this pollutes the domain model and breaks the entire chain of models and separation. Likewise, having another graph transformation model added to the GMF models increases the complexity and may impact of usability, low level and maintainability work negatively.

Therefore, the suggested generic mapping model should allow developing all models in one place and keep map keys or entries for each model, and then automatically separate the information according to each model key, and also synchronizes and updates whenever changes made in any model that affect any other model. This approach can be viewed as a reverse engineering of the current GMF development process such that mappings via the suggested generic mapping model are made simultaneously as the development of all models advances. Then, instead of having an EMF generator model and a GMF generator model, only one generator model is produced to generate both EMF and GMF code. From a developer’s view, all models should be visible as main elements of this generic mapping model in one tree editor including the generator model after generating it.
While developing the SMC diagram editor using GMF, I was faced with the question: “what I can do with the SMC diagram editor and why I should develop it?” Visualization in general can be easily achieved using several available UML-based modeling tools. However, by acquiring experience with GMF, I realized the importance of specificity specifically in the DSM development. Yet, the question: “what I can do and what are the benefits of the SMC GMF diagram editor” still at a certain degree same as before. Especially, considering that GMF is a powerful, yet complex graphical modeling framework. On the other hand, JET transformation found to be flexible and easy to learn and integrate with GMF diagram editors to generate SMC source code.

Model transformations specifically M2T is a valuable and practical direction on DSM development. For instance, having a model of a certain domain, modeling transformation languages allow to generate code or any preferred text format, which is the center of the new software development generation considering modeling versus programming or model-centric than code-centric. Moreover, correctness and validity of models can be ensured and specified using modeling transformation languages before generating the final transformation, as the SMC JET transformation does.
References


[9] Eclipse Model To Model (M2M) Project, www.eclipse.org/m2m, last visited April 1, 2011


[26] XML Path Language (XPath) version 2.0, www.w3.org/TR/xpath20, last visited April 1, 2011


## Appendix A
### List Of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CWM</td>
<td>Common Warehouse Meta-model</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>DSM</td>
<td>Domain Specific Modeling</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>EMP</td>
<td>Eclipse Modeling Project</td>
</tr>
<tr>
<td>EWL</td>
<td>Epsilon Wizard Language</td>
</tr>
<tr>
<td>GEF</td>
<td>Graphical Editing Framework</td>
</tr>
<tr>
<td>GEMS</td>
<td>Generic Eclipse Modeling System</td>
</tr>
<tr>
<td>GMF</td>
<td>Graphical Modeling Framework</td>
</tr>
<tr>
<td>GMT</td>
<td>Generative Modeling Technologies Project</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JET</td>
<td>Java Emitter Templates</td>
</tr>
<tr>
<td>M2M</td>
<td>Model-to-Model Transformation Project</td>
</tr>
<tr>
<td>M2T</td>
<td>Model-to-Text Transformation Project</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-Driven Architectural</td>
</tr>
<tr>
<td>MDSD</td>
<td>Model-Driven Software Development</td>
</tr>
<tr>
<td>MDT</td>
<td>Model Development Tools Project</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>MIGs</td>
<td>Model Intelligence Guides</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Facility</td>
</tr>
<tr>
<td>MS DSL</td>
<td>Microsoft DSL Tools</td>
</tr>
<tr>
<td>MVC</td>
<td>Model-View-Controller</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>SMC</td>
<td>State Machine Compiler</td>
</tr>
<tr>
<td>SPR</td>
<td>Software Productivity Research</td>
</tr>
<tr>
<td>TMF</td>
<td>Textual Modeling Framework Project</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Meta-data Interchange</td>
</tr>
<tr>
<td>XPath</td>
<td>XML Path Language</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Definition</td>
</tr>
</tbody>
</table>
Appendix B

Project Prerequisites

The thesis work is based on the Ganymede release of Eclipse which contains Eclipse Modeling Tools Package. The Eclipse Modeling Tools Package is also available at http://www.eclipse.org/downloads. This modeling package contains Eclipse Modeling Project components, including EMF, GMF, MDT, XSD, OCL, UML, M2M, M2T, and EMFT. It also includes a complete Eclipse SDK, developer tools and source code. In particular, the following table shows the main project names and versions that SMC diagram plug-ins projects are built with:

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse Platform</td>
<td>3.4.1</td>
</tr>
<tr>
<td>Eclipse Modeling Framework (EMF)</td>
<td>2.4.1</td>
</tr>
<tr>
<td>EMF Validation Framework</td>
<td>1.2.0</td>
</tr>
<tr>
<td>Graphical Editing Framework (GEF)</td>
<td>3.4.1</td>
</tr>
<tr>
<td>Graphical Modeling Framework (GMF)</td>
<td>2.1.0</td>
</tr>
<tr>
<td>Java Emitter Templates (JET2)</td>
<td>M2T JET 0.9.0</td>
</tr>
</tbody>
</table>

Downloading the Eclipse Modeling Tools Package stated above should include these main projects and their integrated subprojects and tools.
Appendix C
Running and creating SMC EMF Editor

If the EMF Ecore model is modified, the EMF generator model needs to be updated to reflect the changes. The generator model is updated by right-clicking on it in the Package Explorer view and selecting *Reload*. Generating or regenerating EMF code can be done for the entire model or a subset of the model by right-clicking on the intended element and selecting from the generation options. Moreover, the generated Java code is annotated with *@generated* tags. Removing this tag for a specific class or method, or adding *NOT* after it prevents EMF from overwriting it; therefore, preserving user modification.

This appendix shows how to run and create an SMC EMF model and use it in the EMF tree based editor. The appendix also describes how to create SMC model instance dynamically and use it in the EMF tree editor without running the SMC EMF code.

C.1 Creating SMC Instance Model

The SMC EMF-plug-ins projects were generated and updated in Chapter 5, Section 5.3. These projects are run as Eclipse applications by right-clicking on one of them and selecting *Run As > Eclipse Application*. To create an SMC model instance, it is necessary, first, to create a new general project. Then, an SMC instance model is created by right-clicking on the newly created project and selecting *New > Other*, then, selecting SMC Model wizard found under SMC Creation Wizards folder. Proceeding with the wizard, the SMC model file name must be provided with *.smc* extension. By finishing the wizard, the SMC model instance is created and opened in a tree-based editor where elements are added and edited. Figure C-1 shows an example of an SMC model instance in a tree-based editor.
Because the *Multi-line* property of *copyright* and *comment* attributes is set to *true* when developing *smc.genmodel* model, editing these attributes from the properties view brings up a multi-line box for entering text, as shown in Figure C-2.

Moreover, the attributes of the *Transition* class were grouped in categories when developing the *smc.genmodel* model; thus, the properties view of transitions is displayed as shown in Figure C-3.
Furthermore, the ActionName and ActionArg attributes of the Transition class are defined as multiple value attributes; thus, when edited, a dialog for providing action names and a dialog for providing action arguments. Figure C-4 shows a dialog for adding and editing action names.

C.2 Creating Dynamic Instances of the SMC Model

Since the SMC diagram editor that is developed using GMF provides means for creating SMC model instances, generating EMF Editor Code is not required. Because, creating EMF SMC model instances and testing their functionality in a basic tree-based editor, as presented in the previous section, can be achieved dynamically without running the SMC EMF editor. This is done by opening the smc.ecore model and right-clicking on the root element, SmcModel class,
then selecting *Create Dynamic Instance*. The dynamically created SMC model instance is saved as an XMI file in the existing GMF project, as shown Figure C-5.

![Figure C-5: SMC dynamic model instance example](image)

It can be seen that an SMC EMF model editor is created in the workspace without running the SMC EMF code.
Appendix D

Running and Using the SMC Modeling and Transformation tool

The SMC GMF diagram editor consists of five Eclipse plug-in projects including four EMF plug-in projects and one GMF plug-in project. The generation of SMC EMF plug-in projects was discussed in Chapter 5, Section 5.3 and the generation of SMC GMF diagram plug-in project was described in Chapter 6, Section 6.6. All five projects are run together as Eclipse plug-in applications to allow creating SMC diagram editors.

However, SMC toolkit includes a JET transformation plug-in project. The creation of the SMC JET transformation plug-in project was introduced in Chapter 8.

Therefore, there are six SMC Eclipse plug-in projects that make the SMC graphical modeling and transformation tool.

This appendix demonstrates the execution and using of the SMC graphical and modeling transformation tool. For simplicity seek the appendix is divided into two sections as follows.

The first section (Section D.1) shows how to run and use the SMC diagram editor without JET transformation project.

The second section (Section D.2) demonstrates the entire run and use of all six SMC plug-in projects that make the SMC graphical modeling and transformation tool, as a final deployable Eclipse plug-ins.

D.1 Running SMC Diagram Editor Projects

The SMC diagram editor plug-in projects are run by right-clicking on one of them and selecting Run As > Eclipse Application from the popup menu to launch the Run Configurations dialog. Then, the second instance of Eclipse opens. To create a new SMC diagram, a new general project is created to hold the SMC diagram files. By right-clicking on the created project and selecting
New > Other a new SMC Diagram wizard is found under SMC Creation Wizards, as shown in Figure D-1.

![Figure D-1: SMC Creation Wizards](image)

The SMC Creation Wizards includes two options: SMC Diagram and SMC Model. The SMC Diagram wizard creates the SMC diagram editor generated using GMF. The SMC Model wizard creates the SMC model instance in EMF tree-based editor, as discussed in the previous appendix. Proceeding with the SMC Diagram wizard, the first page is for entering a diagram file name which must be terminated with the .smc_diagram extension. Then, the second page is for providing the model with a name which must have .smc extension. By ending the wizard, the new SMC diagram is created in the specified project.

It should be noted that by default the diagram file and model file are persisted in two separate files such that the diagram file stores the graphical information of the diagram and the model file stores non-graphical information representing the domain model elements. The default persistence behavior can be changed to store all information in one file by setting the Same File For Diagram And Model property to true. The Same File For Diagram And Model property is found under the Gen Editor element of the GMF generator model. Moreover, the diagram file extension and the model file extension can be changed using the Diagram File Extension and Domain File
Extension properties found under the Gen Editor element, too. Figure D-2 shows an example of an SMC diagram.

![Figure D-2: SMC diagram editor example](image)

As shown in Figure D-2, the Project Explorer view on the left side shows the created model file and the diagram file. The diagram file has been opened and shows two maps which have been added using the Palette, shown on the right, which contains creation tools grouped in groups as specified in the tooling model (smc.gmftool). Both, the model file and diagram file are saved as XMI files and can be open using a text editor. As specified in the mapping model (smc.gmfmap), the MainMap element and Map element are created directly on the diagram surface whereas states elements are created inside their compartments either in the MainMap or Map element except for the StartState element which can be only created once in the MainMap element. Similarly, Entry and Exit elements are created only inside their compartments in any state element; in turn, Action elements are added only inside their compartment in any Entry or Exit element. A transition can be created between two states according to its type and link mapping.
specification specified in the mapping model. The properties view is used to provide or modify
text and appearance of the elements. From Figure D-2, it can be seen that the property view is
open displaying some properties of the selected transition grouped as specified in the EMF
generator model (smc.genmodel).

D.1.1 Using SMC Diagram Editor
The SMC diagram Palette contains creation tools that can be selected to create elements on the
canvas. Alternatively, placing the mouse over the canvas or a specific element created on it
allows a popup bar to appear containing available elements to select and create. Textual
information for each created element is entered in place on the canvas or from the properties
view. Likewise, common features such as zooming, arranging, aligning and selecting all elements
on the diagram are all supported by the GMF runtime. These services and others are available
from the main toolbar or from the Diagram menu.

Because the Printing Enabled property of the Gen Plugin element in the smc.gmfgen model is set
to true, the SMC diagram is printable by right-clicking on the canvas and selecting File > Print
from the popup menu, which also includes Save As Image File option. The Print and Print
Preview options are also available from the File menu in the Eclipse main menu. Moreover, the
property view also includes options for displaying rulers and grids, as well as, for modifying the
appearance of any element including fonts and colors unless the element has been given a specific
appearance in the graphical definition model. Additionally, the generated SMC diagram code
includes a package for handling the diagram preferences which provide more options to control
how the diagram is displayed. The SMC Diagram Preferences are accessible by navigating to
Window > Preferences > SMC Diagram.

D.1.2 SMC Validation
Chapter 7 discusses and gives examples on SMC validation. In short, to validate and identify
problems in the SMC diagram, SMC GMF diagram editor defines Audit Rules and, as
appropriate, some of the Audit Rules are set to operate in the live validation mode and other set to operate in the batch validation mode. The live audit rules are invoked automatically, whereas the batch ones are invoked manually by running the Validate option found on the Diagram menu. Then, any errors will be shown in the Problems View categorized according to their type. Elements on the diagram canvas that have these errors will be decorated with icons indicating the error type. Moreover, live validation has other options including showing live errors in a popup dialog or a Console. These options are accessible by navigating to Window > Preferences > Model Validation where the Constraints category is also found listing all SMC diagram Audit Rules and providing information for each constraint, as well as, allowing disabling and enabling any constraint.

**D.2 Running SMC Diagram Editor projects with SMC JET Transformation Project as a Graphical Modeling and Transformation Tool**

This section shows how to run all SMC Eclipse plug-in projects, and also explains the final deployment of the SMC modeling and transformation tool. As with SMC diagram editor projects, JET transformations are also run as Eclipse plug-ins. As shown in Figure D-3 there are six plug-in projects for the SMC modeling and transformation tool.

![Figure D-3: SMC modeling and transformation tool projects](image)

Right-clicking on any plug-in project and selecting Run As > Eclipse Application launches the Run Configurations dialog. By default, all opened workspace projects are included in the Plug-ins of the Run Configurations dialog. Next, in the Eclipse workbench, an SMC diagram is created
and used as demonstrated in the previous sections. The SMC JET transformation plug-in; then, is invoked by right-clicking on the SMC model instance in the Project Explorer view and selecting Run As > Run Configurations. In the Run Configurations dialog, JET Transformation is selected to create a new configuration, as presented in Figure D-4.

![Run Configurations dialog](image)

Figure D-4: Run configurations of SMC JET transformation plug-in

As shown, the SMC model instance (test.smc) is automatically selected as the Transformation Input and the Transformation ID is set to "ca.queensu.cs.afmg.smc.jet", which is the ID that was set for the SMC JET transformation plug-in project. By clicking Run button, the SMC model instance will be inspected and transformed to the SMC source code and a Java skeleton class, as shown in Figure D-5.
As presented in Figure D-5, the SMC JET transformation plug-in creates a folder named (smc) in the same project containing the SMC model instance (test.smc). Moreover, two files (Example.sm and Example.java) are generated in the created smc folder. First, the Example.sm file contains SMC source code and it can be opened in a text editor, as shown in Figure D-6.
Second, the `Example.java` file contains the Java skeleton class, which defines method signatures of transitions actions and instantiates the `Context` class. The user of SMC should implement these methods to interact with the generated `Context` class. Figure D-7 shows the generated `Example.java` class opened in the Java editor.
Moreover, the SMC JET transformation plug-in outputs the transformation process and any errors or warnings into the Eclipse Console as shown previously in Figure D-5.

### D.2.1 Packaging and Deploying SMC plug-ins projects

Eclipse provides an Export wizard to export and deploy resources from the workspace. The Export wizard is accessible from File > New menu and it supports exporting resources in some different types. For exporting and deploying SMC plug-ins projects, the Deployable plug-ins and fragments type is selected. By finishing the wizard, SMC plug-ins will be saved as JAR files representing Eclipse plug-ins and SMC modeling and transformation tool. Users need to copy the plug-ins into plugins folder found in the Eclipse directory.

---

Figure D-7: SMC JET transformation generated Java class
Appendix E
SMC Commands and Grammars

The State Machine Compiler (SMC) [11] release SMC v.5.0.2 is used as a case study in this project. The SMC essential background was provided in Chapter 3. SMC is available from download at http://smc.sourceforge.net. The SMC installation requires extracting SMC zipped package and placing it in a directory. Then, the SMC package includes Smc.jar (SMC’s Compiler) and statemap libraries. The full path to the ../../../Smc/bin directory, which contains Smc.jar, must be added to the PATH environment variable, and also the full path to the statemap libraries should be added to the CLASSPATH environment. As, a Java command line application, SMC requires Java 1.5.0 (JRE-standard edition) or higher to be installed.

E.1 SMC Commands

SMC’s command-line takes the following arguments:

```
Java –jar smc.jar –{target language} -{options} {sm file name}.sm
```

- `smc.jar`: represents the SMC compiler
- `{sm file name}.sm`: represents the SMC file containing SMC source code
- `{target language}`: takes one of the following supported programming languages:
  - `-c`: generates C code.
  - `-c++`: generates C++ code.
  - `-csharp`: generates C# code.
  - `-groovy`: generates Groovy code.
  - `-java`: generates Java code.
  - `-lua`: generates Lua code.
  - `-objc`: generates Objective-C code.
  - `-perl`: generates Perl code.
  - `-python`: generates Python code.
  - `-ruby`: generates Ruby code.
  - `-tcl`: generates [incr Tcl] code.
  - `-vb`: generates VB.net code.
  - `-php`: generates PHP code.
-**scala**: generates Scala code.

-**{options}**: options are as follows:

-**table**: generates an HTML table describing the FSM.

-**graph**: generates GraphViz.dot file.

-**glevel**: (used only combining with –**graph** option) determines the details representation that will appear in the GraphViz diagram. It takes Integer value from 0 for least details to 2 for most details as follows:

  - **glevel 0**: generates state names, transition names and pop transition nodes only.
  - **glevel 1**: generates all of the –**glevel 0** plus transition guards and transition actions.
  - **glevel 2**: generates all of the –**glevel 0** and –**glevel 1** plus state entry and exit actions, transition parameters, pop transition arguments and transition action arguments.

-**sync**: (used only with the -java, -groovy, -vb and -cs) causes the addition of synchronized keyword to the transition methods declarations in the context class as for Java and Groovy implementations. For VB.Net and C# implementation, –**sync** causes the encapsulation of the transition method’s body in Syncclock Me, End Syncclock block, and lock(this) block respectively. Hence, -**sync** insures thread safety in these specified supported languages.

-**help**: prints SMC’s command line options.

-**version**: outputs the version of SMC.

-**verbose**: produces verbose messages during compilation.

-**suffix**: allows to specify the suffix for the generated file name than the default one.

-**noex**: causes not to generate exception handling code. (Used only with the -c++); Generates assert( ) rather than thrown exceptions in case there are unhandled errors; for instance, when an action yields a transition from within a transition. SMC, even when using -**noex**, still generates try/catch/rethrow blocks in order to protect the FSM against application thrown exceptions.

-**nocatch**: causes not to generate try/catch/rethrow blocks.

-**cast**: (used only with the -c++), sets casting operator. The allowed C++ cast operators are: dynamic_cast (default), static_cast and reinterpret_cast. For instance –**cast** can be used as follows: -cast static_cast.

-**d**: determines the output directory for the generated code. By default the generated code is the same directory as the (.sm) file. When specifying a particular directory, the directory must be accessible from the current working directory and writeable.

-**headerd**: (used only with the -c, -c++ and -objc options.), specifies output directory for (.h) generated files. If neither -**d** nor –**headerd** is specified, then (.h) files are placed in the same directory as the (.sm) file.
-g: causes the addition of debugging output to the generated code. Yet, this output is not generated unless turned on at run time. To do so; for example in Java:

```java
_state_machine.setDebugFlag (true); where _state_machine is the FSM instance. By default the debugging output is placed in the standard error; for instance, in Java: (System.err), but this can be changed as follows:

_state_machine.setDebugStream(java.io.PrintStream).
```

-nostreams: (used only with the -c++ and -g options.), causes the generated code not to use IOStreams for debugging output, but the application must provide subroutines or macros to handle such situations.

-serial: used when persisting SMC’s finite state machines, as will be discussed in the persistence section.

-reflect: used with reflection, further details are presented in the reflection section.

-return: causes SMC not to exit.

### E.2 SMC EMNF Grammar

FSM := source? start_state class_name header_file? include_file* package_name* import* declare* access* map+

source := '%{ raw_code %}'

start_state := '%start' word

class_name := '%class' word

header_file := '%header' raw_code_line

include_file := '%include' raw_code_line

package_name := '%package' word

import := '%import' raw_code_line

declare := '%declare' raw_code_line

access := '%access' raw_code_line

map := '%map' word '%%' states '%%'

states := word entry? exit? '{' transitions* '}'

entry := 'Entry {' actions* '}'

exit := 'Exit {' actions '}'

transitions := word transition_args? guard? next_state '{' actions '}'

transition_args := '(' parameters ')' parameters := parameter | parameter ',' parameters parameter := word ':' raw_code
guard := [' raw_code ']
next_state := word |nil' |push_transition |pop_transition
push_transition := word '/' 'push(' word ')'|'nil/push(' word ')'|push(' word ')
pop_transition := 'pop' |'pop(' word ')' |'pop(' word ',' pop_arguments* ')
pop_arguments := raw_code |raw_code ',' | pop_arguments
actions := dotnet_assignment | action | action actions
dotnet_assignment := word ' = ' raw_code ' ; '
action := word '(' arguments* ' ); '
arguments := raw_code | raw_code ',' arguments
word := [A-Za-z][A-Za-z0-9_.]* | [A-Za-z][A-Za-z0-9_.]*
<div class="comment">// Reads in code verbatim until end-of-line is reached.</div>
raw_code_line := .* \n\r
<div class="comment">// Read in code verbatim.</div>
raw_code := .* 
<div class="comment">// Both the // and /* */ comment types are supported.</div>
<div class="comment">// Note: SMC honors nested /* */ comments.</div>
comment1 := '//'. '*\n\r
comment2 := '/\*'. '*\r\n