Model Transformation in context of Driver Assistance System

Meta-model based transformation for Simulink and Scicos

Master Thesis

for

the fulfillment of the academic degree

M.Sc. in Automotive Software Engineering

Faculty of Computer Science

Department of Computer Engineering

Submitted by : Abhishek Mallikarjuna Kappattanavar

Matriculation number: 325392

Supervisor : Prof. Dr. W. Hardt, Technische Universität Chemnitz

Dr. W. Lindner, Elektrobit Automotive GmbH, Böblingen

Acknowledgement

This master thesis would not be possible without the extensive support and encouragement of many people. I am very grateful to my mentors Prof. Dr. Wolfram Hardt and Dr. Ariane Heller who supported me throughout my master's degree. It is only with the guidance of Prof. Dr. Hardt, I have been able to successfully pursue by master thesis at Elektrobit Automotive GmbH under the department of Driver Assistance System.

I would like to acknowledge and extend my heartfelt gratitude to my master thesis advisor, Dr. Wolfgang Lindner, for his constant guidance and support during my thesis work. I wish to acknowledge my heartfelt gratitude to Elektrobit Automotive GmbH for giving me an opportunity to pursue my master thesis and providing technical support to me during my thesis work.

I would like to thank all the members of the Elektrobit Automotive GmbH for their warm welcome and the great working environment they created.

Finally, I would like to thank my family and friends, without their continuous support this work would not have been possible.

Abstract

In today's world we see that Embedded Systems forms a major part in the life of a human being. Almost every device today has an electronic chip embedded in it. When it comes to automotive, these electronic devices are multiplying. This has resulted in innovative methods of developing Embedded Systems. Among them, Model Based Development has become very popular and a standard way of developing embedded systems. Now, we can see that most embedded systems, especially the automotive systems, are being developed using Model development tools like Simulink. In the design and development of Driver Assistance System, Model Based Design (MBD) plays an important role from system design and simulation to code generation. Modeling tool Matlab/Simulink is now among the most popular tools. Due to the proprietary nature of Simulink and challenges in requirement elicitation phase the industry is looking towards an open source alternative, such as Scicos. Since, most of the OEMs are still using Simulink, there is a need for interoperability between Simulink and Scicos. The present work proposes metamodels for Simulink and Scicos, and Model transformation using these Metamodels for the inter-operability.

In order to develop the model transformation the metamodels for Simulink and Scicos were developed using EMF Ecore. These metamodels conform to OMGs MOF Standards. These metamodels were used in developing the transformation definition using the language QVTo. First a simple model was developed, and transformation rules were applied and verified using it. Then a Simulink subsystem of a cross wind assistance system was subjected to forward transformation. The outputs of the model before transformation and that after transformation were compared. They were found to give the same output as desired. Thus, verifying the transformation definition. An attempt was made to achieve reverse transformation. A subsystem in Scicos was considered for reverse transformation. After subjecting it to transformation, an intermediate model conforming to Simulink metamodel was obtained. This shows that the interoperability between Scicos and Simulink can be achieved.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Embedded Systems have become ubiquitous in today's world, be it consumer electronics, medical instruments, military equipment or transportation vehicles. The automotive systems today consists of a large number of embedded systems catering to several requirements in the automotive systems, from human-machine interface to safety, from chassis system to power train. These systems are turning more and more complex as the requirements of each systems are increasing daily, based on the market requirements, government regulations and many other reasons. In order to deal with these complexities, Model Driven Engineering approach is being increasingly used.

Model Driven Engineering (MDE) is a methodology for development of software. The main focus of MDE is creating and exploiting domain models. These models are conceptual models to a particular problem and to its related topics.

1.1 Motivation

Model Driven Engineering has become a standard way of developing embedded systems [1]. In the design and development of Driver Assistance Systems in the automotive field, Model Based Design (MBD) plays an important role from system design and simulation, to code generation. The modeling tool, Matlab/Simulink [2][1] is the most popular tool for Model Based Design. Due to the proprietary nature of Simulink and challenges in requirement elicitation phase the industry looks towards an open source alternative, such as Scicos. Since, most of the Original Equipment Manufacturers (OEMs) are still using Simulink, there is a need for an interoperability between Simulink and Scicos.

1.1.1 Open Source Software and Open Standards

A standard is a level of the quality or attainment of a product or a service. It is a measure against which other products are conformed. Standards help in improving the

levels of quality, safety, reliability, efficiency, and interoperability. This benefits the users as well as producers in terms of convenience, portability, extensibility and monetary costs.

Standards gain a lot more importance in the field of Software, especially when it comes to interoperability of products and services in software and/or hardware. Internationally, several organisations set or define standards for common interfaces, and any change in the standards are generally made only with the consensus of the members or stakeholders involved.

An **open standard** [3] is a standard that is available for the public to use. It may include various rights associated with it. It may also include several properties based on the process it was designed. Among the popular definitions of Open Standard is the definition given by Bruce Perens[4], an Open Source exponent. As per Perens, Open Standard is more than just a specification. It is the principles behind a standard and the way of offering and operating the standard that makes a standard to be Open. It should include certain principles as follows:

i. Availability: The standard should be open for all to read, use or implement.

ii. Maximize end-user choice: No lock in by vendor.

iii. No royalty: Open standards are free of royalty or any fee and free for all to use and implement. However, it may include a fee for Certification by an organisation setting standards.

iv. No Discrimination: Open standards cannot be favourable to one over another except for the standard's compliance of a vendor's implementation.

v. Extension or subset: The implementation of the open standard may be extended or may be made a subset. However, the certifying organisation may decline to certify the extended version or may include restrictions.

vi. Predatory practices: License terms may be employed by the Open Standard to protect it against subversion of standard by embrace and extend tactics.

1.1.2 Open Source Software and Tools

Open Source Software and Tools also follow certain principles in its development, licensing, distribution, usage and even modification and extensions like Open Standards. In general open source software should have certain freedoms associated with it, which includes [3]:

i. Freedom to use and run the software
ii. Freedom to study and understand the software including the source code and also to adapt changes based on the needs of the user
iii. Freedom to distribute or redistribute the copies, so that anyone can use it.
iv. Freedom to modify or extend the software including the freedom to improve the software and release them to the public.

Benefits of using Open Source Software

Some of the advantages and benefits of using Open Source Software are [3]:

i. **Reliability:** In an Open Source Software, generally the presence of any bugs or errors are known very quickly and fixed.
ii. **Stability of the Software:** A Software may become obsolete when a vendor releases a new version and decides not to support the older version. These problems are greatly reduced when using Open Source Software.
iii. **Auditability:** In an open source software, the source code is open for all to read and modify. Hence, any bugs or any security risks are more visible and user knows about it. It is easy for a third party to audit an open source software unlike a proprietary software.
iv. **Flexibility and freedom:** Open source software provide more freedom and flexibility, since most of the open source software follow Open Standards. This will help in modifying the set of software and match it with other modules of the software which provide the required feature for the change in the requirements.
v. **Support and Accountability:** Open source software also disclaim any liabilities and warranties like proprietary software. With respect to support, it is claimed that proprietary software vendors provide support for their software while open source software vendors many times do not. However, many vendors create open source software and then retain the company to provide support for the software.

Also, if an open source software becomes very popular, then many companies which gains expertise would then provide support of the software. So effectively equating itself to the support provided by the proprietary software vendors.

vi. **Security:** An open source software developer focuses more on the technical aspects of the software, making sure that the software covers all the technical features including security. Also, if any security vulnerability is present then it is immediately discovered in an open source software as against a proprietary software.

1.1.3 The Requirement of Interoperability

An approach of MDE is called Model Based Design (MBD). MBD is used in the development of embedded systems. A tool such as Matlab/Simulink is required for the development of embedded systems based on Model Based Design approach. As such, Simulink has virtually monopolised [2] the embedded field, especially the automotive industry. Simulink is a proprietary software tool which has high cost of purchase and at the same time low freedom. Due to this reason some of the software vendors in the automotive domain are looking towards an alternative tool, an open source tool which would provide more freedom and be cost effective.

Among the alternatives present, Scilab/Scicos is a more popular tool. However, many among the OEMs are using Simulink in the development of the systems. In the development of an automotive system, in many instances, the initial or the basic design is made by the OEMs. This design is then provided to vendors to include various major systems and to develop different functions. The vendors may prefer to use Scicos in the development for obvious reasons mentioned above, but the OEMs provide the basic design developed in Simulink. In order to use Scicos in the development, the vendors would have to redesign the whole system again using Scicos. This brings in the requirement of transformation of Simulink model into Scicos model in an automated way and vice versa. Therefore, this has provided a motivation to make an attempt to transform the models using Model Transformation techniques.

1.2 Objective

Model transformation has been proposed for the interoperability between Simulink model and Scicos model. The present work proposes the development of metamodels for model transformation. The following are the objectives of the present work:

1. Development of metamodel for Simulink
2. Development of metamodel for Scicos
3. Development of Forward Transformation Definition

The metamodels for Simulink and Scicos are proposed to be developed based on OMG MOF Standards. The metamodels are to be developed using the EMF Ecore's Graphical Modeling Tool. As a part of the solution for the interoperability, the development of the transformation definition using QVT Operational Mappings (QVTo) is also proposed.

1.3 Organisation of Thesis

Chapter 1 explains about the open standards and the open source software. It is followed by the discussion about the current trend in the industry regarding Model Based Development and the requirement of interoperability between Simulink and Scicos. Chapter 2 presents an extensive survey carried out on model transformation. Chapter 3 presents the concepts behind the model transformation including Model Driven Engineering, Model Driven Architecture, Model Based Design, Metamodels and Model Transformation. Chapter 4 explains the different tools, languages and technologies, some of which are part of the transformation itself and others which have been used to bring about the transformation. Chapter 5 presents the development of Scicos and Simulink metamodels. It explains the development process from analysis of the model to the development of the metamodels required for the model transformation. Chapter 6 presents the results of the development of metamodels. Chapter 7 presents the development of the transformation definition using QVTo. The forward transformation and the reverse transformation are explained in detail. Chapter 8 presents the results of transformation and includes verification and validation of the transformations. Finally, in Chapter 9, the conclusion and the further work is discussed.

Chapter 2
State of the Art

A brief introduction to the motivation behind the current work and the objective to develop the metamodel and the model transformation between Simulink and Scicos was explained in Chapter 1.

In this chapter a survey on prior art has been carried out and the findings are discussed. Several transformations have been carried out between different kinds of models and technologies.

Di Natale et al [5] have worked on the model transformation between SysML and Simulink. SysML follows the Model Driven Architecture (MDA) approach while Simulink follows the MBD approach. The authors have tried to make use of the benefits of both the approaches in their work in order to leverage the strengths of the both. They have realized the transformation using the TOPCASED modelling tool Acceleo. Using Acceleo they have transformed SysML model to text. The transformation automatically generates Simulink subsystems from a SysML model in a top down flow or it can also generate a SysML model of a Simulink Subsystem in a bottom up flow. The authors have initially felt that the use OMG QVT [6] language would be suitable candidate for the transformation. However, QVT assumes that both the source model and the target model conforms to the respective metamodels expressed by Meta Object Facility (MOF). Though this holds for SysML model, it is not so for the Simulink model. There is no openly available MOF description or metamodel for Simulink. Hence, in [5], model-to-text transformation has been performed, wherein the SysML model is translated into Matlab model generation script. This output Matlab script is then processed by the Matlab engine to obtain Simulink model with the same expressiveness of the source Simulink model. In the reverse transformation the Matlab script is used to parse the Simulink model and generate an XML model description that then transformed into XMI, which is the standard input for the SysML and is supported be TOPCASED.

Some work has also been carried out on the transformation of UML model into Multidimensional model [7]. Arrassen, I. et al [7] have worked on the transformation of

the UML model into Multidimensional (MD) model. The authors have developed MOF based metamodels for the UML Model and MD model. They have developed the transformation using the model transformation language SmartQVT, which is an Eclipse plugin and is based on the OMGs QVT Operational specifications. Their source metamodel is the metamodel of UML and their target metamodel is the MD metamodel. They have represented their source model using file format XMI 2.0. This file is provided as input of the QVT transformation, which automatically generates another file in XMI format. This file represents the target model conforming to the metamodel corresponding to the MD model.

Zhang, L. et al [8] presents a framework which enables the transformation of Matlab/Simulink model into an actor oriented design language (SysteMoC) automatically, that enables Design Space Exploration (DSE). This is a vertical transformation for code generation.

Peng G. et al [9] presents a model transformation between UML and Simulink. UML, being a semi-formal modeling language lacks accurate semantics and cannot be used for validation of correctness of embedded software development. Model transformation has been used by the authors to solve this problem. They have transformed UML design model into Simulink simulation model as a solution. The source metamodel used is UML StateMachine metamodel and the target metamodel used is the Simulink/Stateflow metamodel. A set of mapping rules have been formed for the transformation. Their implementation claims to improve the efficiency of the embedded software. The work has included development of metamodels using KM3 language, which is a lightweight text metamodel development language developed by INRIA. The language used in the transformation is ATL (Atlas Transformation Language), also developed by INRIA.

Dae-Kyoo Kim et al [10] have used metamodel based transformation approach is for unifying the IEC 61850 and IEC 61970 core standards of smart-grid domain for substation automation and power operation management. There are significant data exchanges between these two standards, requiring high levels of compatibility. These standards having different perspectives and independent evolution, are not much compatible with each other. Due to this reason the practitioners come with their own data mapping in an ad-hoc way, which results in issues with inter-operability and data

consistency. To solve this issue, the authors have found a solution by defining a common semantic model of the standards and then provide a semantic transformation method to transform the model of standards into common semantic model. The authors have presented a metamodel based approach for unifying the standards and then transformation using QVT. Metamodels are developed for the each of the standards and also for the Unified Model.

Meedeniya, D et al [11] have developed model transformation tool called SD2CPN. It is scenario-based and having analysis capabilities. It models scenarios in UML2 sequence diagrams (SDs), and this is transformed into coloured Petri nets (CPN), which enables analysis of the synthesised models. The work claims to facilitate the software engineers to design and develop software and also perform verification of the software.

Ben Younes, A et al [12] and Achouri, A et al [13] include model transformation from UML Activity Diagram (UML AD) into Event-B model. The works proposes the transformation tool for transforming the UML AD model into Event-B model for verification using the B-prover.

The present work proposes a triangular transformation as presented in figure 2.1. The ultimate goal comprises of transformation between SysML and Scicos, Simulink and Scicos, and SysML and Simulink. It can be divided into three different parts:

a. Transformation between SysML and Scicos
b. Transformation between Simulink and Scicos
c. Transformation between SysML and Simulink

It can further be divided into Forward transformation and Reverse transformation. The current work on this thesis focuses mainly on the forward transformation from Simulink to Scicos along with an attempt on reverse transformation. This will complete a cycle of transformation. The transformations of the other two are out of the scope of the current thesis.

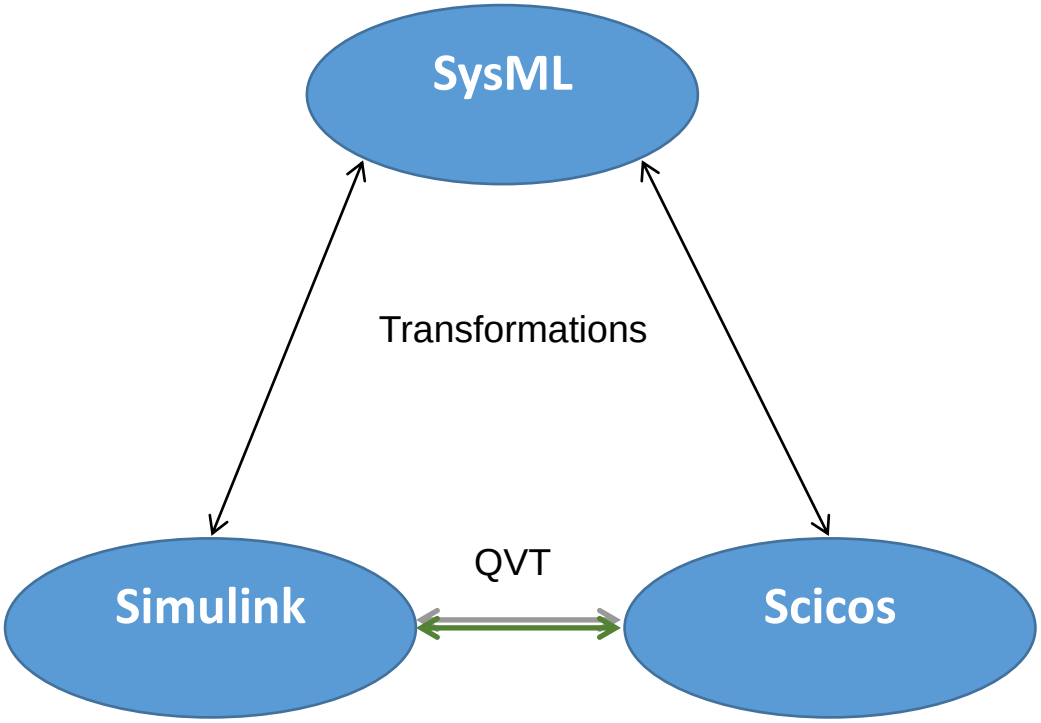

Figure 2.1: Proposed triangular transformation

Several attempts have been made in bringing about transformations for other types of models. As seen in the literature survey, transformation between UML and Simulink, SysML and Simulink, Sequence Diagram to Coloured Petri Nets model, UML to Multidimensional model and other types of models have been attempted. However, this triangular transformation has not been attempted in the previous works. Also, no transformation attempt has been made to complete the cycle of transformation between Simulink and Scicos. Hence, the current work is a novel work.

Summary

This chapter presents the literature survey carried out to understand the latest work on the model transformation between different kinds of models and technology. Model transformation has been carried for Simulink and UML, UML to MD model, Sequence diagrams to Coloured Petri-nets and for many more different kinds of models. The survey has shown that Metamodels are being used for the development of the transformation definition. Metamodels have been used in the current work also. The

work distinguishes itself with the proposal of triangular transformation. It also attempts both the forward and reverse transformation to complete a round-trip transformation.

Chapter 3
Background Concepts

A literature survey on model transformation was carried out in chapter 2. The prior art on model transformation between different types of models were discussed. The findings of the literature survey were presented in the previous chapter. In this chapter, Model Driven Engineering, a concept related to model transformation is discussed. Model Driven Architecture and Model Based Design are later explained.

3.1 Model Driven Engineering

Models are increasingly playing an important role in the development of embedded system and software development. The term used to define the methodology of using models is Model Driven Engineering (MDE). MDE is a term that focuses on creating and exploiting models in the field of Software Engineering. Its main purpose is to increase or maximise compatibility between, and among various systems by including reusability of the model components. It helps in simplifying process and also promotes communication within and among various teams by standardizing terminologies and best practices.

In order to understand Model Driven Engineering, an idea of a model has to be known clearly. A **model** is a representation of a system. It is an abstraction of reality. It must be able to reflect the original system for which it is developed, even if the system is yet to be built or even if it is just an imaginary system. The model even if it is a simple representation of a system, must be able to represent at least some of the properties. A model must be usable, which means that the model developed must be able to be used in place of the actual system for some purposes. A model may be in the form of a graphical representation or in the form of text. Usage of the models in system development constitutes Model Driven Engineering.

There can be many different models for a given system. The details may vary and they may be described or may not be. There exist a relationship between two models of the same system.

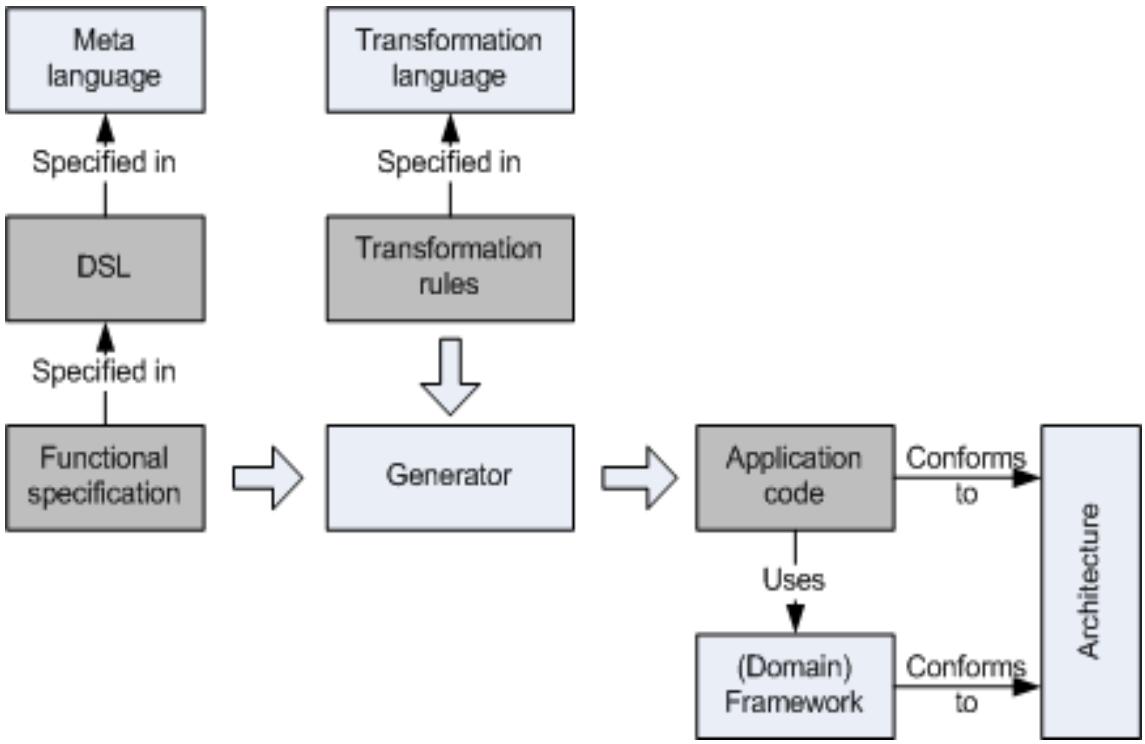

Figure 3.1: Model Driven Engineering Overview [14]

An overview of Model Driven Engineering is presented in figure 3.1. A model of a system describes the functional specifications of that system. It is specified using a Domain Specific Language (DSL) such as Simulink or Scicos. This is in turn specified using a Metalanguage. The model is used as an input to a Generator which generates a code for any specified application. The code conforms to some Framework and is produced in a high level language such as C, C++, Java etc. It also conforms to a certain architecture and complies with certain standards. The standards depend on the application. For example, ISO 26262 is a safety standard for automotive applications and DO-178B is a safety standard for aerospace applications. The Generator uses certain transformation rules to generate the application code. These rules are specified using a transformation language such as QVTo.

In MDE two major trends can be identified[5]. One is the Model Driven Architecture (MDA) and the other Model Based Design (MBD). Both these trends have common objectives and principles; they show differences in their approaches and practices including the technologies and languages they use.

3.2 Model Driven Architecture (MDA)

Model Driven Architecture (MDA) is an approach for developing software systems. It gives a set of rules or guidelines for modeling a system. MDA[15] is a standard developed in 2001 by Object Management Group (OMG) in order to define reference architecture for the development of software systems based on models. MDA is a registered trademark of OMG. As per MDA, it will be able to generate code using requirement models.

A model is first developed as a Platform Independent Model (PIM) using a Platform Independent Modeling language. This Platform Independent Model is then transformed into Platform Specific Model (PSM) making use of transformations and mappings of formal rules. Some important standards related to the concept of MDA include Unified Modeling Language (UML), SysML, Meta-Object Facility (MOF), XML Metadata Interchange (XMI) and Metamodels, which are also defined by OMG. An important aspect of MDA approach is that the system specifications are independent of the implementation platform or technology.

3.3 Model Based Design (MBD)

Model Based Design (MBD), though similar to MDA, has a different approach. While MDA is mostly oriented towards software design, MBD is more popular in control and systems engineering domain. In MBD, the syntax and semantics are restricted. The popular tools such as Simulink and SCADE [25] come under the MBD category of tools. The models can generally be simulated and also code can be generated in an automated way using these tools. They have an underlying Model of Computation based on mathematical rules, using which the process of verification can also be achieved.

The steps in the development of a system in Model Based Design approach are as follows:

1. Modeling a system: The requirements of the system are gathered in the first step. Using these requirements a model is developed using a modeling tool. This model

consists of connected network of blocks. Each block represents a physical elements of the system being modeled. The connections represents the flow of signals and data.

2. Analysis and synthesis: The previously conceived system model is analysed and the dynamic characteristics of the model are identified from the model. A control system is synthesised based on these characteristics.
3. Simulation: The model thus developed is simulated off-line first, which helps in identifying errors immediately in specification, requirements and modeling. Then a real-time simulation is carried out. The code for the model is generated and deployed on to a prototyping hardware. Using this hardware, a real-time simulation is carried out and tested.
4. Integration and deployment: The model and the components of the model developed are integrated and code generation is done automatically. The code thus generated is deployed onto the hardware.

3.3.1 V-Model

V-Model [26] which is supported by Model Based Design is presented in figure 3.2. V-model stands for Verification and Validation model. It is a sequential path of execution of processes. Each phase must be completed before the next phase begins. Testing of the model is planned in parallel with the corresponding development phase.

Requirements Engineering:

The first phase of the V-model is the Requirements Engineering. The requirement specification of a system is collected. For example, in a window control system of an automotive, the specifications would include the basic functions of opening and closing the window. Further, based on different laws and safety regulations, the window should have a child lock system. Again, based on market requirements or geographical requirements, the window should have an emergency/express closing system or during winter it must have a de-freezing facility. These requirements have to be captured in the requirement specification of the system. The captured specifications must be developed into a conceptual model of the system. This process from capturing the specifications to

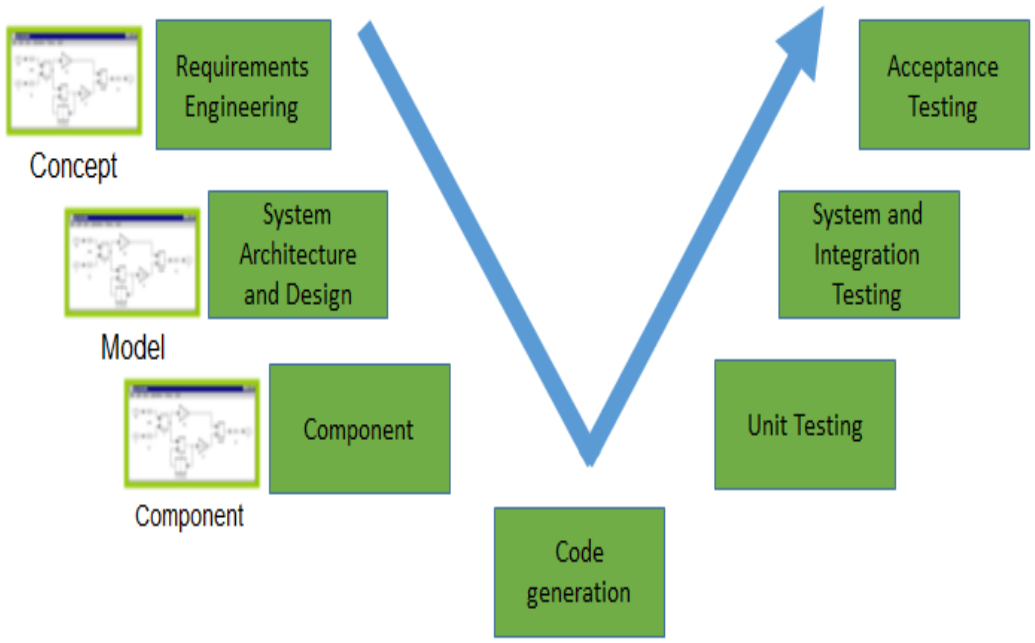

Figure 3.2: Model Based Design which supports V-Model

the development of the conceptual model of the system form the Requirements Engineering phase. The resultant output of this block is the conceptual model of the system. The basic functionality of a system is conveyed by a conceptual model. The users should be able to understand the system with ease using the conceptual model. The essential purpose of a conceptual model is as follows:

a. To improve the understanding of the system
b. To convey the details of the system specification efficiently
c. To form a reference model for the system designers and developers
d. To assist in the communication and collaboration for the development of components
e. To provide documentation for future reference and enhancement

System Architecture and Design

The conceptual model obtained as a result of the requirement engineering phase is used for the development of the system architecture and design. A system architecture is the high level description of the system. In Model Based Design, the system architecture

is developed as an interconnected network of blocks. Each of the blocks represent a component of the system. It describes the relationship between the interfaces, its architecture, dependencies and the details of the technology. This results in the system model. The system model provides top level description of the system.

Component Design:

The obtained system model which is the design of the system is then divided into its component systems. These component systems include the details of every component that forms the system. It provides the functional logic, details of inputs, outputs and the events of the components. The result of this component design is the component model.

Implementation and Code Generation:

The component model developed during the component design phase is used by the developer to develop and implement the components. Then code is generated automatically using automatic code generators for these components at this step. The generated code may be in the form of Java, C, C++, Ada or any high level languages.

Unit Testing:

The unit testing is done along with related control data and determines if the module functions as designed. In the V-model, the generated code is deployed onto a virtual or a prototype hardware. Then unit testing is performed on the system, then the validation and verification is carried out against each component model. Unit tests are designed based on the internal module designs.

Integration and System Testing:

The integration testing ensures reliability and functional performance of each block in a system after the components are integrated into a system. After the integrated system passes the integration test, the system test is performed on a complete system. It is conducted to check if the system is working as per the requirements specification. It is verified and validated against the System Design.

Acceptance Testing:

Acceptance testing is conducted to check if the system has implemented and meets all the requirement specification provided by the user. At this stage the system is verified and validated against the requirement specification.

3.3.2 Advantages of Model Based Design

The advantages of the Model Based Design include:

1. Simplified Communication due to the use of models. The models form the graphical representation of the system and can be easily understood by different stakeholders of the system.
2. The functional model of the system can be simulated early during the development of the system. This helps to identify and correct the errors in the design phase.
3. Code generation is done in an automated way. This reduces the possible errors that gets introduced otherwise and makes the generated code highly consistent with the model designed. It also reduce the time, costs and effort need to develop the system, thereby making the system development more efficient.
4. Model Based Design helps in the design and development of the highly complex system with ease unlike the design of the same system using classical software development methods.

3.4 Metamodels

A conceptual model of a modeling technique is a metal model [16]. A metamodel is a model of a model. It describes the model. It provides a formal definition for a modeling language. A metamodel captures or describes several information of an application domain like syntactic and semantic information. It provides a definition for the family of models that can be developed in the modelling environment that results from it.

It describes the entities, attributes and the relationship between them, in a modeling environment. A metamodel can also be called a modeling paradigm. A metamodel may be described using textual format or in the form of graphs via diagrams or even in the form of tree structure. For example, the certain eclipse extension tool allows developers to define a metamodel using textual syntax, while EMF Ecore tools allows developers to develop a metamodel using tree structure, or class diagrams or even in the format of an XML file.

A metamodel has to be written in a well-defined language called metalanguage. It is a specialised language to describe modeling languages. In the MDA framework different symbols are used for a metalanguage. There is an infinite number of layers of model-language-metalanguage relationships[15][17]. There are four layers defined by OMG standards. These layers are called M0, M1, M2 and M3. This is presented in the following paragraphs.

The Meta Layers of MDA

Layer M0: M0 layers forms the bottom layer in the hierarchy of the OMG Standard of the Meta layers. The Layer M0 depicts the instances of the model elements defined in a model. It represents the real world objects.

Layer M1: M1 layer contains models, which describe a system. M1 layer elements are classes, attributes and other model elements. An example for M1 layers is UML model. The model elements of the M1 layer are generalized classes of instances at the M0 layer. Also, every element of the M0 layer is an instance of the elements present in the M1 layer. The relationship between the layers M0 and M1 are presented in figure 3.3

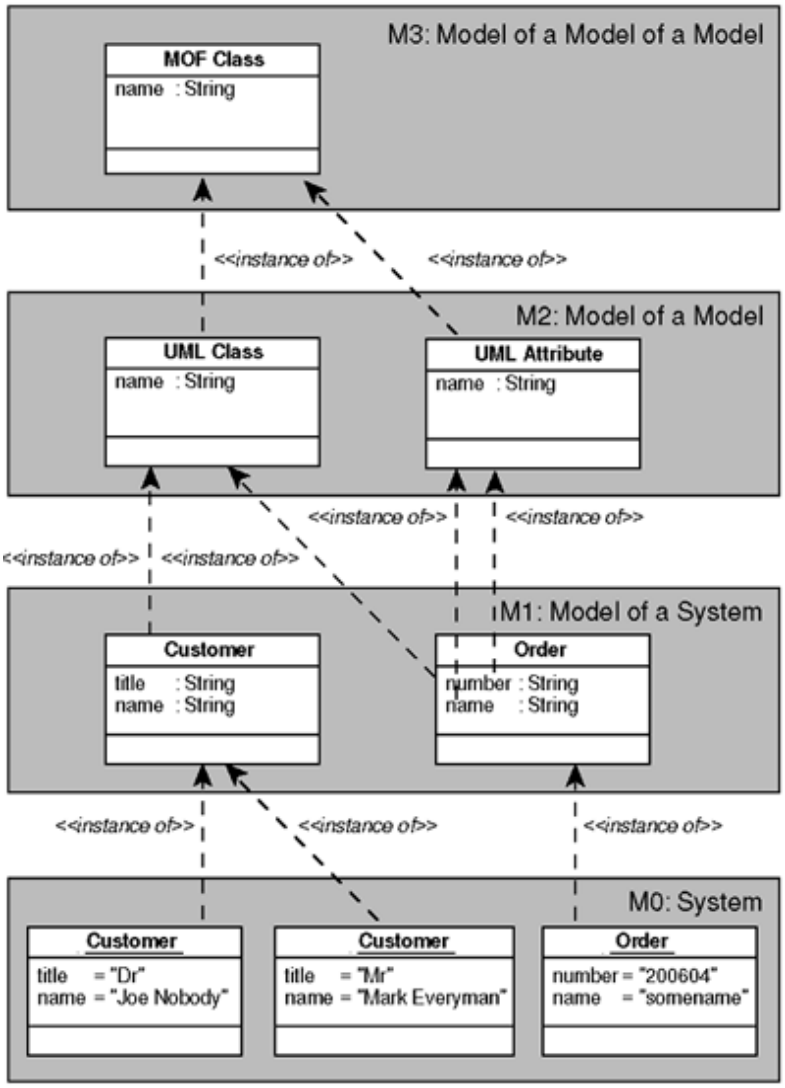

Figure 3.3: Overview of Layers M0 to M3 [17]

Layer M2: The metamodel, which is a model of a model lies in the level M2. This metamodel describes the language using which the model (in Layer M1) is developed. It describes the semantics of the modeling language.

Layer M3: The Metametamodel is a model of metamodel. The OMG standard level of metametamodel is M3. Meta Object Facility (MOF) is a metametamodeling language and is situated at this level. MOF describes how a metamodel is developed and is situated at the top of the hierarchy of the layers.

The overview of the layers M0 to M3 are presented in the figure 3.3.

3.5 Model Transformation

Model Transformation is a process of transforming a model from a given model into a desired model or working software or a schema, generally in an automated way. A program takes a model as an input and based on the transformation rules specified in it, it provides an output model which is consistent with the input model. A transformation may be direct or may involve intermediate transformations and intermediate models.

A transformation definition specifies how a model is to be transformed. It is a set of transformation rules which describes how a model conforming to a given metamodel is transformed into another model which conforms to a different metamodel. The transformation rules describes how the elements and constructs of the source model are to be mapped to the elements of the target model and also their relationship.

The process of Model Transformation is presented in the figure 3.4. At the input of the transformation is the source model shown in figure as Model A. The Model A conforms to a metamodel named Meta Model 1. Based on the transformation rules specified, the transformation programme transforms the source Model A into a target model which is shown as Model B. Model B conforms to metamodel named Meta Model 2. The resultant target model is consistent with the corresponding source model after the transformation is performed.

Two kinds of Model transformation are defined. One is model to model transformation and the other is the model to text transformation.

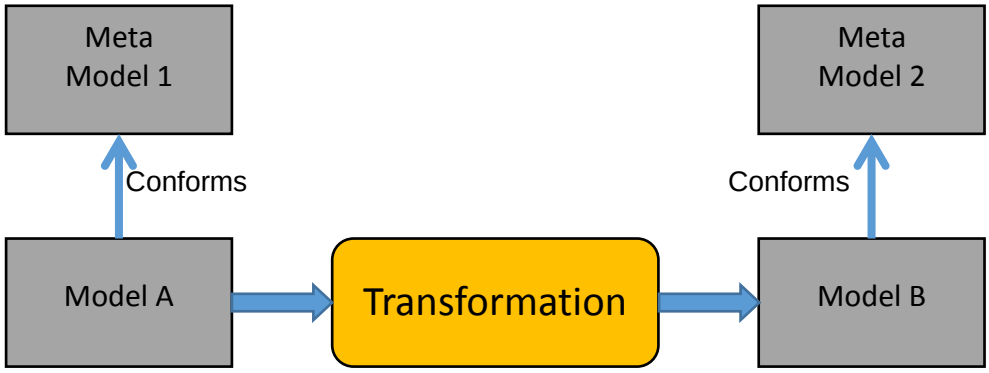

Figure 3.4: Model Transformation

Mens et al. [18] provides taxonomy for model transformation: Based on the languages of the source and the target models, it can be classified as endogenous or exogenous. If source and target models are expressed using the same language or in other words the source and the target metamodels are the same (in-place transformation), then it is endogenous transformation. If the source and target models are expressed using different languages, or in other words the source and target metamodels are distinct, then it is exogenous transformation.

A transformation can be classified as a horizontal or vertical based level of abstraction the models reside at. When the source and the target models reside at the same abstraction level then it is a horizontal transformation, when they are residing at different levels of abstraction then it is a vertical transformation.

3.5.1 Model to Model Transformation (M2M)

Model to Model transformation involves transformation of an input model into a target or an output model. The model elements of an input model are mapped with the model elements of the output model to generate an output/ a target model.

3.5.2 Model to Text Transformation (M2T)

In model to text transformation a model is transformed into a text form, mainly for code generation or for the purpose of documentation. When a working software is required or a schema is required, then Model to Text Transformation would be more suitable. Typically, in a M2T transformation, the model is transformed into some general-purpose programming language like Java, html, C, C++ etc. There are many code generators available which use M2T Transformation

In the present work model to model transformation is being carried out. This transformation can be classified as a horizontal transformation. This facilitates the inter-operability between Simulink and Scicos. The results are discussed in Chapter 8.

3.6 Summary

The concepts behind Model transformation have been elaborated. An overview of Model Driven Engineering, Model Driven Architecture and Model Based Design has been presented in this chapter. Model Driven Engineering is a methodology of development of Systems. Systems are designed using models. It uses models for representing units or modules of a system. Model Driven Architecture is an approach of MDE defined by OMG. It focuses on development of Software Systems. Model Based Design is an approach of MDE to develop Embedded and Control Systems. It presents the popular V-model in context of Model Based Design.

The concept of metamodel is introduced. A metamodel is a model of a model. It describes the language which is used to develop a model. It can be expressed in many different ways. OMG defines different Metalayers of MDA. A metamodel lies at the M2 layer in the metalayer hierarchy.

An overview of the process of model transformation using the metamodel are presented. The terms related to transformation are discussed.

Chapter 4
Tools and Technologies

The related concepts behind model transformation were discussed in chapter 3. The concepts of Model Driven Engineering, Model Driven Architecture, Model Based Design, the meta-layers of MDA were presented in the previous chapter. The concepts of Metamodels and the Model Transformation were also presented there.

This chapter introduces some of the tools, technologies and languages that are used for the development of metamodels and model transformation.

4.1 Tools used in the development of candidate model

The current work on transformation is carried out between models developed in Scicos and Simulink. These model are called as candidate models. These models can be developed and viewed using their corresponding tools. The following sections provide an overview of these tools that are necessary for developing the candidate models.

4.1.1 Scilab/Scicos

Scilab is an open source software for scientific numerical computation [19]. Scicos is an important toolbox which comes along with Scilab. It provides a block-diagram based graphical editor for development and simulation of dynamical systems.

Scicos

Scicos is an open source, graphical system modeler and simulator, developed in the Metalau projects by INRIA. It is an important toolbox which comes along with Scilab [19]. It provides a block-diagram based graphical editor for development and simulation of dynamical systems. Scicos is considered to be an open source alternative for Simulink.

Scicos allows simulation, compilation and debugging, and code generation. It has several advantages. It provides a means and an environment to develop systems in a modular fashion. It provides several elementary and complex blocks needed for development of a model. It allows users to develop their own reusable blocks if needed. This is useful when a large system is designed as different modules by different teams

working on it. A large number of built-in blocks are available in the Palette. These blocks provide the basic operations that are required to construct models of the systems. Users rarely need to build new blocks from scratch.

Code Generation is an important part of Embedded System Design. There is an internal code generator for Scicos, which generates code in C. There are several external code generators too available for generating codes into different languages.

In the present work, during the forward transformation a Scicos model acts as the target model. Scilab/Scicos tool is used to view and to modify the target model. During the reverse transformation the Scicos model acts as the source model and is developed using Scilab/Scicos tool.

4.1.2 Matlab/Simulink

Matlab is a high level, multi paradigm, fourth generation language for numerical computation, visualization, and development of applications [20]. It is a proprietary programming language developed by MathWorks. It provides an interactive environment for iterative exploration, design and problem solving. It has several mathematical functions for several operations including for linear algebra, Fourier analysis, filtering, optimization, numerical integration, and for solving ordinary differential equations (ode). It provides means to visualize data using built in graphics and tools for creation of plots including custom plots. It provides functions for integrating Matlab algorithms with other external applications. It provides several toolboxes for the use of several applications from different engineering backgrounds. Simulink is one such tool provided by Matlab for model based design, which provides an environment for modelling and simulation. The installation of Matlab is a pre-requisite for the use of Simulink.

Simulink tool

Simulink is a model based design tool which allows modelling and simulation of systems [21]. It is highly integrated with MATLAB and enables the use of MATLABs algorithms in it. It also allows the exporting of Simulation results to MATLAB. Along with modelling and simulation of the embedded systems, it also supports automatic code generation, testing and verification.

In the present work, the source model is a Simulink model during the forward transformation and is developed using Simulink tool of Matlab. In the reverse

transformation the Simulink model acts as the target model. The Matlab/Simulink tool is required to view and modify the target model.

4.2 Tools and technologies for the transformation

The process of model transformations entails the use of several tools, technologies and languages. The following sections introduces the language of QVT, the tools such as Eclipse and other technologies such as EMF Ecore which have been used in the current work. QVT comes under the elements of the MDA framework defined by OMG. The other elements used in the work are MOF and OCL.

4.2.1 Object Constraint Language (OCL)

Object Constraint Language (OCL)[27] is a key component of the QVT and is an expression language. OCL can be used for MOF and UML models. OCL can be translated to programming languages. OCL is helpful to specify initial attribute values, to get the derivation rules for the attribute or the associations, to get query operations for the body, to specify the targets for messages being sent, to describe the guard conditions on state charts, to specify end user queries on a UML model, to give added precision and expressiveness to the developers and to specify constraints on operation.

4.2.2 Meta Object Facility (MOF)

Meta Object Facility (MOF)[22] is an OMG Standard that defines the language to define modeling languages and also enables the building of tools for defining it. MOF is also used to define a stream or file based interchange format for M1 models. The interchange format is based on XML and is called XMI (XML Metadata interchange). MOF is defined using itself, and hence to generate standard interchange formats for metamodels, XMI can be used.

4.2.3 Query, View, Transform (QVT)

QVT or Query, View, Transform is a language used for transformation of models [6][23]. It is a language which operates on models based on metamodels conforming to Meta- Object facility (MOF) 2.0. QVT is defined by the OMG.

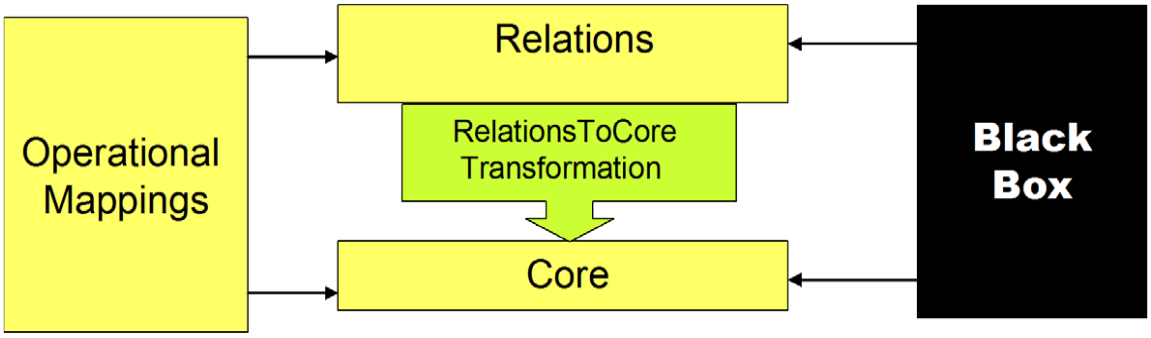

Figure 4.1: QVT language and the relationship between their metamodels [6]

There are three parts in QVT as the name itself suggests. The first being named **Query**, as queries can be applied on source model, which is an instance of a source metamodel. The second is **View** which describes the way the output or the target model should look like. The third one is **Transform**, wherein the results of the queries are projected on the views to obtain a target model. The QVT specifications are based on the MOF and OCL (Object Constraint Language) Specifications.

There are different domain specific languages defined in QVT. The first category belongs to the declarative languages and includes QVT Relations and QVT Core. In addition to QVT Relations and QVT Core, there are mechanisms for invoking imperative implementation of transformation from QVT Relations or QVT Core. There are two such mechanisms and includes one standard language QVT Operational Mappings and the other non-standard *Black-box MOF Operation* implementation. Each of these, i.e. QVT Relations, QVT Core and QVT Operational Mappings (QVTo) can be combined along with a black-box operation. QVT-Relations and Core permits both unidirectional and bidirectional model transformations to be written while QVT Operational Mappings is designed for writing unidirectional transformations.

QVT Relations

In QVT Relations, a set of relations are specified between the models that are being transformed. These relations must hold for having a successful transformation. Here, a transformation is invoked either to modify one model to enforce model consistency or to check the two models for the consistency.

QVT Core

The QVT Core language is a simpler language but as powerful as the QVT Relations. In QVT Core conditions are evaluated over a flat set of variables against a set of models and pattern matching is supported over these flat set of variables. It can be implemented directly for transformation or may be used as a reference to the QVT Relations.

QVT Operational Mappings (QVTo)

QVT Operational Mapping (QVTo) provides an implementation of imperative language of QVT. It populates the same trace models as the Relations language. It provides OCL extensions allowing a procedural style, and a concrete syntax which looks similar to the procedural languages familiar to programmers. Operational Mappings can be transformed to Relations and Core.

Black Box implementation

Many algorithms are very difficult to implement using OCL and some cannot be expressed at all. In order to overcome this a black box implementation is provided in QVT. A Blackbox allows programmers to code complex algorithms using any supported programming languages.

4.2.4 Eclipse Modeling Framework (EMF)

Eclipse Modeling Framework (EMF)[24] is a modeling framework and code generation facility. It is a framework for building tools and other applications based on a structured data model. Models can be created using EMF in many different ways. It provides the user with an editor (tree structure with properties), it also allows user to import annotated Java classes. If the XSD component is installed, one can import an XSD file. Models can be created using UML or Ecore diagram editor may be used to develop a model. Certain plugins allow user to model using textual syntax as well.

Whatever the method being used to work with the domain model, a file with an extension `.ecore` is present in the workspace i.e., the model is saved with this extension. This file can be opened using a text editor, wherein the model can be viewed in XMI serialized format. EMF allows user to work with this model file using a basic tree editor with Properties view.

Ecore

EMF Core or Ecore is the model that is used in representing models. It is itself an EMF model, and hence its own metamodel. Hence, it can be called as a meta-metamodel. Ecore forms the central picture of the EMF world. An Ecore model can be created by a UML model, or an XML Schema, or even an annotated Java interface. Different forms of model can be generated, including Java implementation code using an Ecore model. An XMI serialization can be generated using an Ecore model.

4.3 Summary

Scilab is an open source software for scientific numerical computation. Scicos is an important toolbox which comes along with Scilab. It provides a block-diagram based graphical editor for development and simulation of dynamical systems. Simulink is a model based design tool which allows modelling and simulation of systems. It is highly integrated with MATLAB and enables the use of MATLABs algorithms in it. The model developed using Scicos and Simulink are candidate models for the model transformation.

Object Constraint Language (OCL) is a key component of the QVT and is an expression language. It is used for MOF and UML models. It helps to create models which are more extensive and precise.

QVT or Query, View, Transform is a language used for transformation of models. It is a language which operates on models based on metamodels conforming to Meta-Object facility (MOF) 2.0, which is a standard set by the OMG (QVTr, QVTo, QVTc). Eclipse Modeling Framework (EMF) is a modeling framework and code generation facility. EMF Core or Ecore is the model that is used in representing models. Ecore forms the central picture of the EMF world.

Chapter 5
Metamodel Development

Different tools, technologies and languages used in the transformation between Simulink and Scicos was discussed in chapter 4. A brief description of Scicos, Simulink, QVT and EMF was also presented in the previous chapter.

As mentioned in the Chapter 1, the first objective of the current work is to develop the metamodels for Simulink and Scicos. This chapter introduces the development of metamodels. A metamodel defines the elements and, also the relation and the interactions between the elements of a model. Since, a metamodel is also a model, it has to be well defined using a language. The language used in developing a metamodel is called a metalanguage. The metalanguage used in the present work is EMF Ecore's Graphical Modeling Language. The relationship between a model, its language and the metalanguage is as presented in figure 5.1.

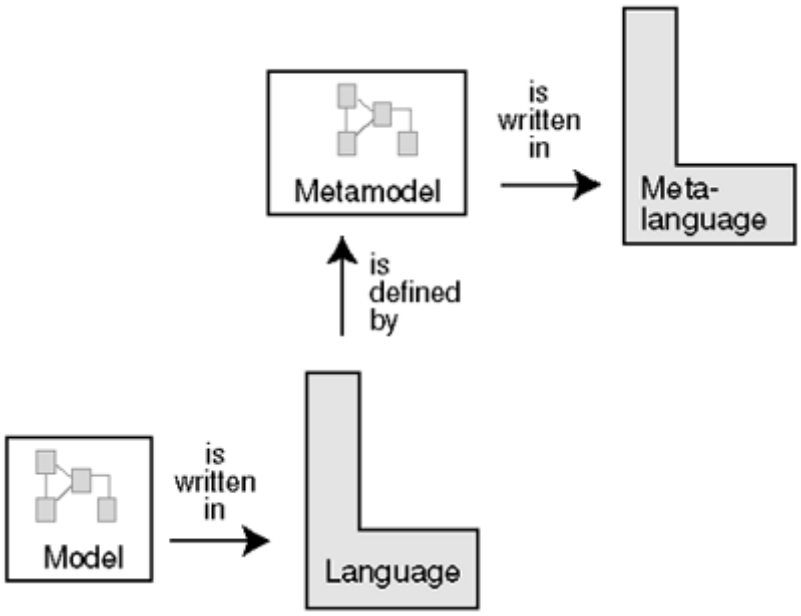

Figure 5.1: Relationship between a language and its metalanguage [17]

Here, the transformation is being done between Simulink and Scicos. This necessitates metamodels for Simulink and Scicos to be used. The following sub-sections explains more on these metamodels.

5.1 Requirement of metamodel development

Metamodel development or metamodeling, is developing a model of a modeling language in order to define that language. Modeling a metamodel has two important utility.

1. A modeling language is unambiguously defined
2. The transformation rules describe how a model in a source language can be transformed into a model in a target language.

In the current work the modeling languages are already present. Hence, this work does not define any new language. The metamodels of Scicos and Simulink are required for the transformation of the models, but these are unavailable in the public domain especially the metamodel of Simulink. The Scicos metamodel required for the transformation using QVTo should be MOF based metamodel. This requires the development of both the metamodels i.e. metamodel for Scicos and Simulink.

5.2 Metamodeling

A metamodel can be represented in many different forms. It can be represented using textual syntax using certain existing tools. It can also be represented as a set of Annotated Java classes or using UML2 or using EMF's Graphical Modeling Tool. There are several tools that are available in the market for the development of metamodels. Most of these tools provide a means of representing the metamodels in a graphical form. Eclipse Modeling Framework (EMF), which is introduced in chapter 4, is one of the frameworks that is used in the development of a metamodel. As explained, the metamodel developed using EMF is saved in the Ecore format with file extension of .ecore. It can be further retrieved in different formats supported by EMF. A pre-requisite for model transformation using QVTo is the availability of the MOF based metamodels for the source and also the target models. In the present work, metamodels are developed

using the graphical editor provided by EMF, which has similar construct as that of class diagrams. The metamodels developed using EMF are MOF based metamodels.

A metamodel representation developed using EMF's Graphical Modeling Tool is presented in the figure 5.2. This is a class diagram representation and makes it easier for the developer to visualize a model. It shows different elements of a metamodel designed as classes such as *ScicosModel*, *Object*, *Block*, *Link*, *Graphics*, *Diagram*, *SuperBlock* and *Properties*. The properties of the elements are modeled as attributes. For e.g. the property of the element *ScicoModel* is *Version*, the property of *Block* is *Name* and that of the *Properties* are *Name* and *Value*. On similar lines the other properties are modeled. It also shows the relationship between the elements. The elements *ScicosModel* and *Object* have the Composition relationship, which means that the *ScicosModel* composes the element Object within it. This relationship is called as Containment reference. The element SuperBlock inherits the element ScicosModel. This relationship is called Inheritance. Other kinds of relationship includes uni-directional and bi-directional references.

The ecore metamodel developed using the graphical modeling tool can be viewed and modified using a text or an XML file editor. The figure 5.3 shows a screen shot of the ecore metamodel in XML format.

There are several tools that are available in the market for the development of metamodel. Most of these tools provide a means of representing the metamodels in a diagrammatic form.

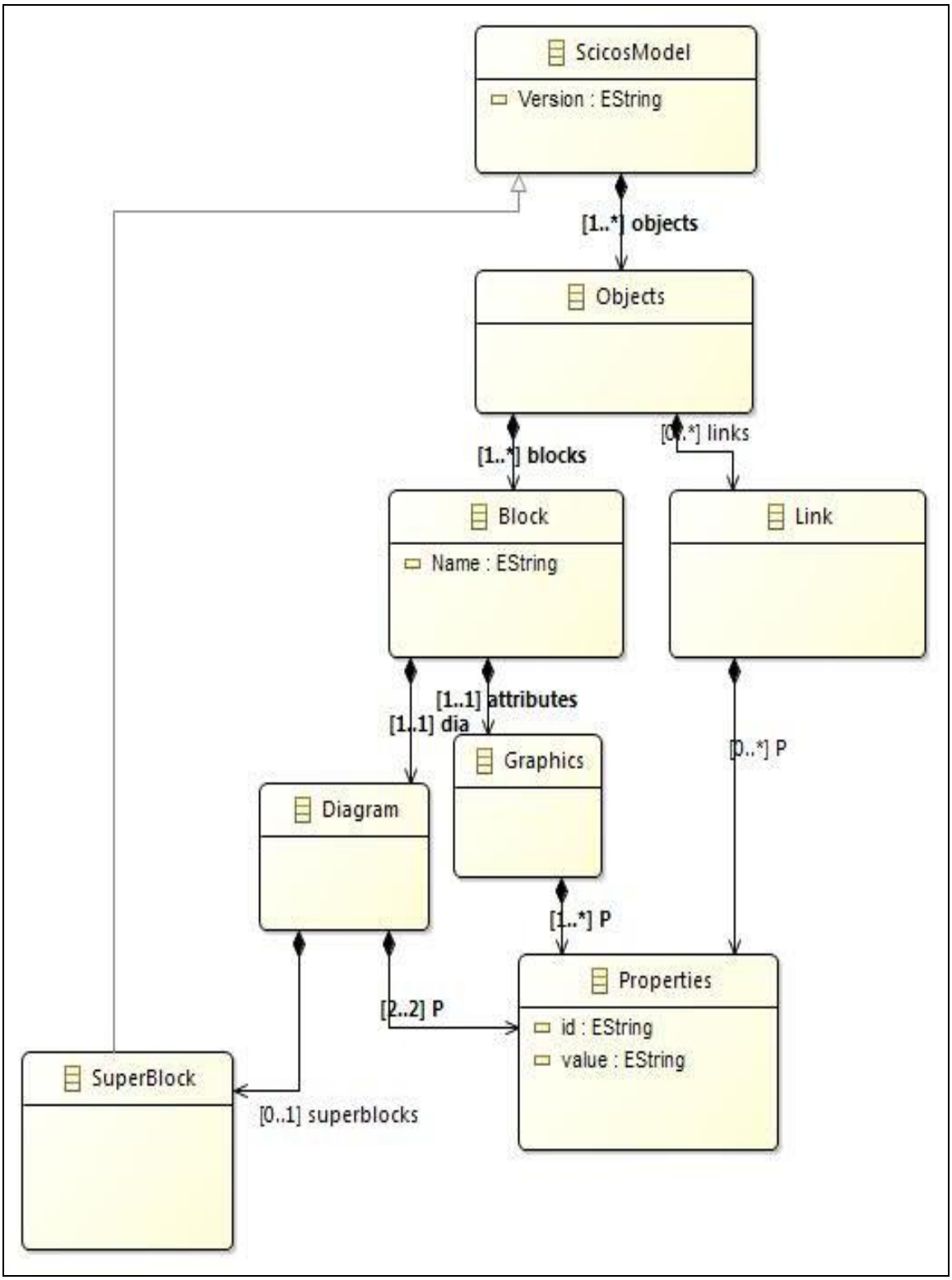

Figure 5.2: Representation of a metamodel using EMF Graphical Modeling Tool

```
<?xml version="1.0" encoding="UTF-8"?>
<genmodel:GenModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:ecore=
"http://www.eclipse.org/emf/2002/Ecore"
    xmlns:genmodel="http://www.eclipse.org/emf/2002/GenModel" modelDirectory="/SCM/src"
    editDirectory="/SCM.edit/src" editorDirectory="/SCM.editor/src"
    modelPluginID="SCM" modelName="Scicos" modelPluginClass="" rootExtendsClass=
    "org.eclipse.emf.ecore.impl.MinimalEObjectImpl$Container"
    importerID="org.eclipse.emf.importer.ecore" complianceLevel="8.0" copyrightFields=
    "false"
    editPluginID="SCM.edit" editorPluginID="SCM.editor" operationReflection="true"
    importOrganizing="true">
  <foreignModel>Scicos.ecore</foreignModel>
  <genPackages prefix="Scimm" disposableProviderFactory="true" ecorePackage=
  "Scicos.ecore#/">
    <genClasses ecoreClass="Scicos.ecore#//ScicosModel">
      <genFeatures createChild="false" ecoreFeature="ecore:EAttribute
      Scicos.ecore#//ScicosModel/Name"/>
      <genFeatures property="None" children="true" createChild="true" ecoreFeature=
      "ecore:EReference Scicos.ecore#//ScicosModel/objects"/>
    </genClasses>
    <genClasses ecoreClass="Scicos.ecore#//Objects">
      <genFeatures createChild="false" ecoreFeature="ecore:EAttribute
      Scicos.ecore#//Objects/Name"/>
    </genClasses>
    <genClasses ecoreClass="Scicos.ecore#//Link">
      <genFeatures createChild="false" ecoreFeature="ecore:EAttribute
      Scicos.ecore#//Link/id"/>
    </genClasses>
    <genClasses ecoreClass="Scicos.ecore#//SuperBlock"/>
  </genPackages>
</genmodel:GenModel>
```

Figure 5.3: Representation of a metamodel viewed as an XML file

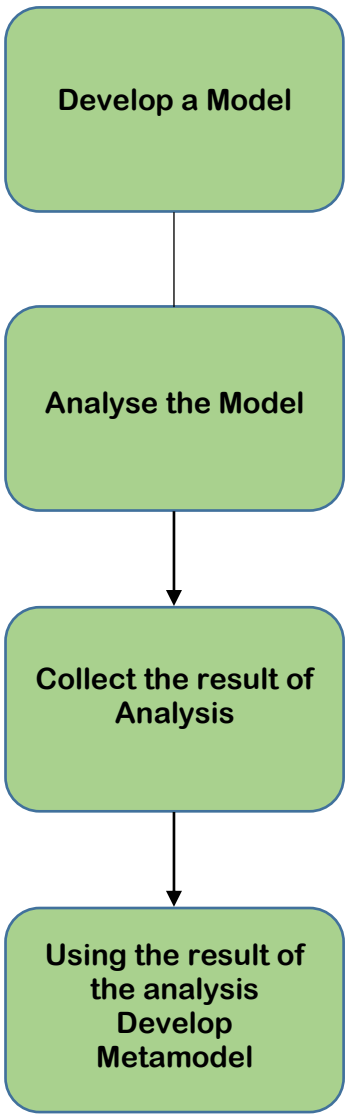

Figure 5.4: Steps to develop metamodel by reverse engineering a model

Metamodel in the present work is developed by reverse engineering a model. The simple steps followed for the development of metamodel are as presented in figure 5.4. First a model is developed. This model is then analysed for the elements, their properties, their constraints and the relationships. The analysed results are collected. This result is then used in the development of the metamodel.

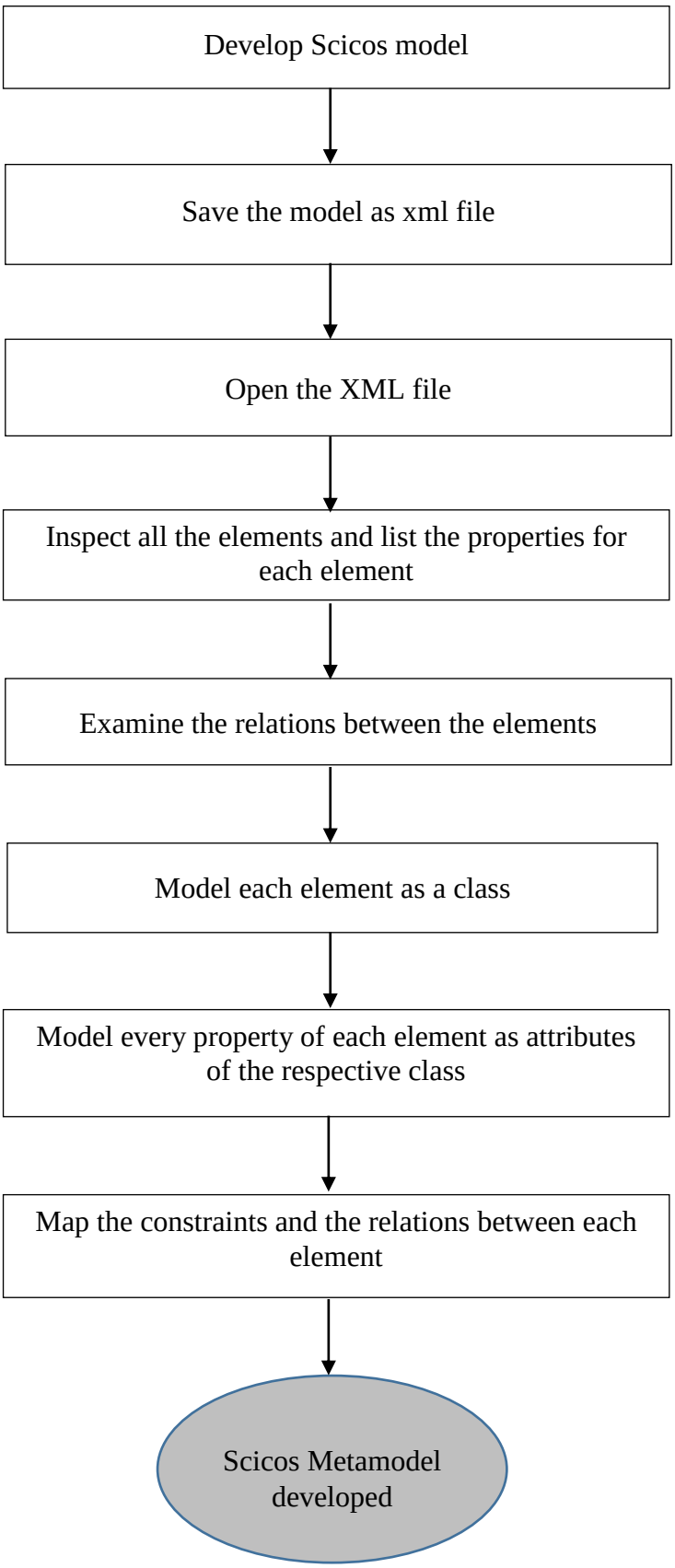

Figure 5.5: Scicos Metamodel Development

5.3 Scicos Metamodel

Scicos metamodel is required for transformation. Scicos metamodel is developed here by reverse engineering a Scicos model. Scicos model is developed using the Scicos tool. In the current transformation a Simulink model forms the input to the forward transformation and a Scicos model is obtained as result of this transformation. Similarly, Scicos model forms the input and Simulink model is obtained as a result of reverse transformation. For the transformation definition, the language QVTo requires MOF based metamodels. The metamodel of Scicos model forms the target metamodel for the transformation definition during the forward transformation. During reverse transformation, the Scicos metamodel acts as the source metamodel. Since, the metamodel required has to be MOF based, it has been developed separately from the beginning.

Flowchart for the development of Scicos metamodel

Steps involved in the development of the Scicos metamodel is presented in the flowchart as depicted in the figure 5.5. Each step is clearly defined and the flowchart is self-explanatory.

5.4 Development of Scicos metamodel

The first step to develop a Scicos metamodel is to develop a model using the Scicos tool. Then the model is studied and the constructs, the elements, the relationship between the elements and the constraints are analysed. Scicos tool provides the option of saving the Scicos models either as a model in Scicos format with .cos extension or as an xml file. The .cos is a binary file and cannot be opened in a text editor and needs the Scicos tool, but a Scicos model saved as an xml file can be opened in a text editor. These xml files can be studied and analysed to develop a metamodel for Scicos. The Scicos model saved as .cos file can be studied and analysed using the Scicos tool. However, using the xml file would be advantageous as it would make the intermediate transformation easier.

In the present work a simple model was developed for the purpose of analysis, using Scicos. This model is saved as an xml file. The saved xml file is then opened in a text editor. Every element of the file was inspected. For a reconstruction of a model all the

details necessary for the model are to be included in the metamodel. Hence, each element was analysed to see if the model could be reconstructed without the element. After the analysis, the elements of the model necessary for the reconstruction was listed. The relationship between these elements and the constraints were also noted. This information obtained as a result of the analysis is used in development of the MOF based metamodel as required by QVTo for the transformation definition.

5.4.1 Scicos Model Analysis

The analysis of the Scicos model resulted in the identification of the elements of the model that are necessary for reconstructing the model. After an initial analysis it was found that certain information of the file was not necessary for the reconstruction of the model and some information in the model were constant or had no effect on the functioning of the model if their values changed. The elements necessary are used to develop the Scicos metamodel which is later used in the transformation definition developed using QVTo. It is used for both the forward and reverse transformation. Among the most important elements identified includes ***ScicosModel***, ***Object***, ***Block*** and ***Link***. The other elements required to reconstruct the model are ***CodeGeneration***, ***Parameters*** and ***Options***. The elements of the model contained several properties. Some of the properties of these elements are constant values, while the values of many other properties had no effect on the working of the models when they were changed. Any effect seen would be related to graphical display without affecting the functioning of the model.

The results of the analysis are mentioned in the following points:

1. The root element of the model is ***ScicosModel***.
2. This root element ***ScicosModel*** contains one element called ***Object***, one element called ***CodeGeneration*** and one element called ***Parameters***.
3. An ***Object*** contains at least one ***Block*** and may contain ***Link***.
4. A ***Block*** may contain another element similar to ***ScicosModel*** in its structure. Such ***Block*****s** are called ***SuperBlock***.
5. A ***SuperBlock*** may contain several ***Block*****s**, ***Line*****s** and all such elements as in ***ScicosModel***.
6. Each element has several properties.

7. The number of properties for a given element are fixed

Scicos is an open source tool and the structure of the Scicos model is simpler to understand. By analysing a Scicos model it can be understood that there are limited number of attributes in the model elements required for reconstructing a model. Hence, it becomes easier to develop a metamodel for Scicos.

5.4.2 Modeling the elements into the metamodel

This analysis is used in the development of the metamodel. The metamodel development is done using the graphical modeling tool of Ecore, which uses UML based graphical development environment. Each element in Scicos is modeled as a class. Hence, the elements ***ScicosModel***, ***Object***, ***Block***, ***Link***, ***CodeGeneration***, ***Parameters*** and ***Options*** are modeled as classes. The elements and their corresponding classes in the metamodel are summarised in the table 5.1

Scicos Element	**Class name in the metamodel**
ScicosModel	ScicosModel
Object	Object
Block	Blocks
Link	Links
CodeGeneration	CodeGen
Parameters	Parameters
Options	Options

Table 5.1: Scicos elements and corresponding classes in the metamodel

5.4.3 Modeling the properties into the metamodel

The root element ***ScicosModel*** contains ***Object***, ***Parameters***, and ***CodeGeneration*** as its child elements. The root element ***ScicosModel*** contains two properties including *Name,* and *Version*. The property *Name* corresponds to the name of the model and *Version* corresponds to the Scicos version used to build the model which is constant for a given version of a tool. The element ***ScicosModel*** is modeled as the class *ScicosModel* in the metamodel. The properties *Name* and *Version* are modeled as the attributes of this class.

As mentioned above the root element ***ScicosModel*** contains the element ***Object***. The element ***Object*** does not contain any properties but the child elements ***Block*** and ***Link***. The ***Object*** contains one or several instances of ***Block***s and ***Link***s depending the model. The element ***Object*** is modeled as the class *Object*. Each of the ***Block***s and ***Link***s are assigned a unique number by which they can be identified. This number forms a part of the attribute *Name***.** The value of the attribute *Name* always starts with 'OBJ_' followed the unique number assigned to it.

The element ***Block*** contained in the ***Object*** is modeled as class *Blocks* and may have several properties. These properties have the same names for different instances of a ***Block***. These properties are modeled as attributes of the class *Blocks*. The attributes are *gui*, *orig*, *sz*, *exprs*, *pin*, *pout*, *pein*, *peout*, *gr_i*, *in_implicit*, *out_implicit*, *flip*, *theta*, *id* and *Name.* These attributes are explained as below*:*

- *Name*: As already mentioned above the value of *Name* starts with 'OBJ_' followed by the unique value assigned to it. An instance of the ***Block*** can be identified against other instances using this attribute.
- *gui*: The attribute *gui* gives the name of the type of ***Block*** it represents.
- *orig*: The attribute *orig* provides the information related to the position where the ***Block*** is situated.
- *sz*: The attribute *sz* provides size with the length and the width of the ***Block***.
- *in_implicit*: The number of input pins of a block are represented by the attribute *in_implicit*.
- *out_implicit*: Similarly, the number of output pins are represented by the attribute *out_implicit.*

- *pin* and *pout*: There can be several ***Link***s that form the input and output to a ***Block***. All the unique numbers assigned to the input ***Link***s are included in the attribute *pin*, and those which are assigned to the output of the ***Block*** are included in the attribute *pout*.
- *id*: The user name specified, if any, by the user to a ***Block***, forms the value of the attribute *id*.
- *flip*: The attribute *flip* indicates if the ***Block*** is flipped.
- *theta*: The attribute *theta* gives the information regarding the angle of rotation of the ***Block***.
- *exprs*: The attribute *exprs* contains the details specific to a given instance of a ***Block***. For example, if the ***Block***, as indicated by the attribute *gui*, is summation unit, then the attribute *exprs* gives the number of inputs that the block requires and the number of outputs it provides. It also contains the information of each of these inputs, including whether they are negative or positive inputs. However, a user can change these based on his requirements.
- *gr_i*: The attribute *gr_i* is unique to each type of ***Block***. It provides the information of the function of the ***Block***. It also gives the information of the display aspects of the ***Block***. The information is used by the Scicos tool for processing and may not be changed by the user. For a given type of ***Block***, this information remains unchanged.
- *pein* and *peout*: There can be several ***Link***s of the type e*vent* that form the input and output to a ***Block*** containing an event port. All the unique numbers assigned to the input *event* ***Link***s are included in the attribute *pein*, and those which are assigned to the output *event* ***Link***s of the ***Block*** are included in the attribute *peout*.

Similar to the element ***Block***, the element ***Link*** too is contained in the element ***Object*** and modeled as the class *Links* in the metamodel. It also has fixed number of properties. The properties are modeled as the attributes of the class *Links*. The attributes of the class *Links* include *xx*, *yy*, *id*, *thick*, *ct*, *from*, *to* and *Name*. An instance of the ***Link*** can be identified by the attribute *Name*. The attributes of the ***Link*** are explained as below:

- *Name*: As already mentioned above the value of *Name* starts with 'OBJ_' followed by the unique value assigned to it. An instance of the ***Link*** can be identified against other instances using this attribute.
- *xx*: It contains an array of values in x-axis. The first value represents the starting point of the *Link* in the x-axis and the last value provides the end point. The starting point is always from the ***Block*** from where the ***Link*** originates and the end point is the ***Block*** where the ***Link*** ends. The other values in the array represent the intermediate points where the path of the ***Link*** changes its direction.
- *yy*: Similar to the attribute *xx*, It contains an array of values in y-axis. The first value represents the starting point of the *Link* in the y-axis and the last value provides the end point. The starting point is from the ***Block*** from where the ***Link*** originates and the end point is the ***Block*** where the ***Link*** ends. The other values in the array represent the intermediate points where the path of the ***Link*** changes its direction. The number of values in the array of *xx* and that of *yy* are equal. Together with the array of values in *xx*, the array of values in *yy* provides the exact coordinates of the ***Link***.
- *id*: The name specified, if any, by the user to a ***Link***, forms the value of the attribute *id*.
- *thick*: The attribute *thick* provides graphic related information about the thickness and the type of the line representing the ***Link***.
- *ct*: The attribute *ct* provides the graphic related information on the colour of the line representing the ***Link***.

A ***Link*** is a connection between two ***Block***s. It facilitates the flow of information, signal, event or data from one ***Block*** to another ***Block.***

- *from*: The attribute *from* contains the unique identifier assigned to the ***Block*** and also the port number from where the ***Link*** gets the flow of data, signal or information.
- *to*: Similar to the attribute *from*, the attribute *to* contains the unique identifier assigned to the ***Block*** and also the port number to where the ***Link*** provides the flow of data, signal or information.

The elements ***CodeGeneration*** and ***Parameters*** are modeled as classes *CodeGen* and *Parameters* respectively. The class CodeGen have the following attributes: *silent*, *cblock*, *rdnom*, *rpat*, *libs*, *opt*, *enable_debug*, *scopes*, *remove* and *replace*. These attributes have very less effect on the functioning of the model. Hence, their values are kept constant or null.

The element ***Parameters*** is modeled as class *Parameters* and has *wpar*, *title*, *tol*, *tf*, *context*, *void1*, *void2* and *void3* as its attributes. The change in the values of these attributes have very less effect in the functioning of the model as most of these corresponds to the values needed to display the model on the screen.

The element ***Parameters*** contains the element ***Options*** with the attributes *3D*, *Background*, *Link*, *ID* and *Cmap*. Similar to ***Parameters***, the values of the attributes of ***Options*** also are used for the display of the model and have no influence on the functioning of the model.

The results of the development of the Scicos metamodels are discussed in the next chapter.

5.5 Simulink Metamodel

In the current work Simulink model is a model that is being transformed into Scicos model during the forward transformation. Simulink model forms the input for the forward transformation. The transformation definitions are developed using QVTo. Along with the Scicos metamodel, Simulink metamodel is also required by QVTo for the development of transformation definition. There are no publicly available metamodels for Simulink [5]. Therefore, a metamodel has to be developed for successfully developing the transformation rules. To develop a Simulink metamodel, the design details of Simulink is required. Since, Simulink is a proprietary tool, the details of the Simulink design is also unavailable publicly. This is a challenge posed by most proprietary tools. In order to overcome this challenge a metamodel is developed by reverse engineering a Simulink model.

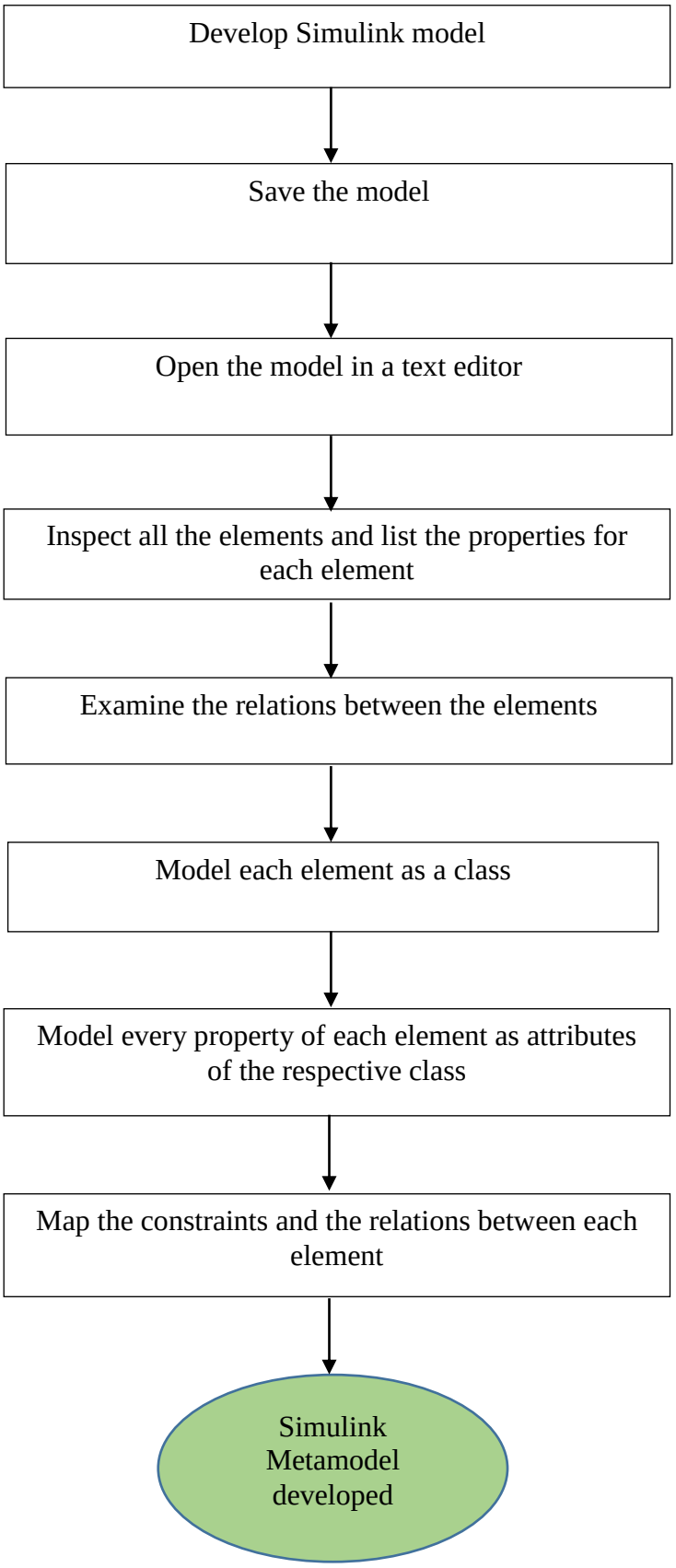

Figure 5.6: Simulink Metamodel Development

5.6 Development of Simulink Metamodel

The first step in developing a metamodel for Simulink is to develop a Simulink model. In order to develop a Simulink model, a system or a model is designed. The requirement specifications of this system are gathered. Based on the specifications, a model is developed. Then a study and an analysis of the Simulink model performed. The Simulink model file is saved as an .mdl file. When using certain versions of Simulink, the file can also be saved as an xml file. The .mdl file can be opened in any text editor for analysis. The xml file can also be opened using any text editor. This xml file basically contains the same information as in .mdl file but in a different format. Hence, the analysis of any one of these files is sufficient to develop a metamodel. A Simulink model in xml format would be more advantageous as it would later help in using xml file for intermediate transformation.

A simple model is being developed for the purpose of analysis using Simulink. This model is saved as an .mdl file, which is then opened using a text editor. Once the Simulink model is opened in a text editor, detailed analysis can be performed for each and every element of the model. The elements present in the model are identified. Every element of the file is inspected for their properties.

Here, a language is not being defined. Hence, the metamodel being developed is not a metamodel for defining a language. Therefore, only those elements of the Simulink model could be considered that would be useful in the model transformation underway. Since, the Scicos metamodel was already developed, only those elements and properties were being considered initially that would be useful in the forward transformation. The other elements and their attributes are being ignored.

Later, further analysis is performed and many more elements and their properties were considered for developing the metamodel required for reverse transformation. During this analysis, each element is again analysed to see if it is required for the reconstruction of the model from the metamodel. It is found that certain information of the file is unnecessary for the reconstruction of the model from the metamodel. Such information is ignored. The result of the analysis of the Simulink model is explained in the next section.

5.6.1 Simulink Model Analysis

The analysis of the Simulink results in the identification of the elements of the Simulink model which are necessary for developing a metamodel. In the first step the elements required for forward transformation were listed and used in the metamodel development.

The elements identified include *Model*, *System*, *Block*, *Line* and *Branch*. Later the other elements required for the reconstruction of the model were identified which includes *GraphicalInterface*, *Outport* and *Inport*. Each element may contain from a few properties to several properties.

The results of the analysis are mentioned in the following points:

1. The root element of the model is *Model*.
2. This root element *Model* contains at least one element called *System*.
3. A *System* contains at least one *Block* and may contain *Line*s.
4. A *System* contains one element called *GraphicalInterface*.
5. A *Line* may contain *Branch*es.
6. A *Block* may contain a *System*. Such a *System* in a *Block* is called a Subsystem.
7. The *GraphicalInterface* contains zero to many *Outport* and *Inport*
8. Each element may have several properties.

This analysis is used in the development of the Simulink metamodel. The metamodel development is done using the graphical modeling tool of Ecore, which uses UML based graphical development environment. Each element in Simulink is modeled as a class. Hence, the elements ***Model***, ***System***, ***Block***, ***Line***, ***Branch***, ***GraphicalInterface***, ***Outport*** and ***Inport*** are modeled as classes. The required Simulink elements and their corresponding classes in the metamodel are summarised in the table 5.2.

As mentioned above each element may have several properties or attributes. Some of the properties are common to different instances of the same type of element. When such elements exist, then these common properties are modeled as *attributes* of the class corresponding to the elements. Also, only those properties that are required, in the transformation of Simulink and Scicos, are modeled as *attributes* of the metamodel classes.

Simulink Element	Class name in the metamodel
Model	Model
System	system
Block	blocks
Line	lines
Branch	branches
GraphicalInterface	graphicalInterface
Outport	outport
Inport	inport

Table 5.2: Subset of Simulink element and corresponding classes

5.6.2 Modeling the elements into the metamodel

The root element ***Model*** contains several child elements including the element ***System***. All other children elements like *ConfigManagerSettings*, *EditorSettings*, etc. are ignored as these elements are not required for the transformation. The root element ***Model*** also contains several properties including *Name*, *Version*, *SavedCharacterEncoding*, *ScopeRefreshTime* and several others. However, only *Name* and *Version* are used in the current work. As explained earlier, the element ***Model*** corresponds to the class *Model* in the metamodel. The properties *Name* and *Version* are modeled as the attributes of this class.

As mentioned above the root element ***Model*** contains the element ***System***. The element ***System*** contains several properties and child elements. The elements contained in the ***System*** are several instances of ***Line***s and ***Block***s and several properties including *Location*, *ModelBrowserWidth*, *TiledPageScale*, *ZoomFactor* and many more. The

element ***System*** is modeled as the class *system* and the required properties *Name* and *Location* as the attributes.

The element ***Block*** contained in the ***System*** is modeled as class *blocks* and may have several properties. These properties vary for different instances of a ***Block***. However, there are several properties common over several instances of the ***Block*** and these properties are modeled as attributes of the class *blocks*. They include *BlockType, Name, SID, Ports, Position* and *BlockMirror*. Generally with one or a combination of these *attributes*, an instance of the ***Block*** can be identified against other instances.

The common properties of the ***Block*** are explained as below:

- *BlockType*: The attribute *BlockType* tells the type of the Block.
- *Name*: It is the name of the ***Block*** specified by the user.
- *SID*: *SID* stands for Simulink Identifier. It is a unique identifier which is automatically assigned by Simulink to the ***Block***. It can be used to identify the ***Block*** against the other instances of the ***Block***. Even if the *Name* of the ***Block*** changes, *SID* remains unchanged. An *SID* may not be modified by the user.
- *Ports*: The attributes *Ports* indicates the number of input ports and the number of output ports the ***Block*** has. If a ***Block*** has either only input port or only output port then only a single value is present. If the ***Block*** has both input and the output ports then it has an array with two values. The first one indicates the number of input ports and the second value indicates the number of output ports present in the ***Block***.
- *Position*: The attribute *Position* is an array or vector containing four values. The values provide the information of the pixel position of the model from the origin. The first two values provide the top-left pixel position and the next two provide the bottom right position of the pixel. The origin is located at the top-left position in a Simulink window
- *BlockMirror*: When the value of this attribute is ‘on’, then it indicates that the ***Block*** is flipped

There were several other properties of the ***Block*** that have not been modeled as attributes of the class *block* and hence not listed here. One such property is *Value*, which

provide the value information of certain kinds of ***Block*** like the *BlockType* 'Constant'. However, these were modeled using a separate class which is explained later.

Similar to the element ***Block***, the element ***Line*** is also contained in the element ***System*** and modeled as the class *lines* in the metamodel. It may also have several properties and include several common properties over many instances of the element ***Line***. These common properties are modeled as the attributes of the class *lines*. The attributes of the class *lines* include *SrcBlock*, *SrcPort*, *Points*, *DstBlock* and *DstPort*. Again, an instance of the ***Line*** can be identified using a combination of these attributes. The properties modeled as attributes in the class *lines* are explained below:

Line is an element in the Simulink which is used to connect two ***Block***s to facilitate the flow of signal, data or information

- *SrcBlock*: The attribute *SrcBlock* contains the Name of the ***Block*** from where the ***Line*** originates.
- *SrcPort*: The attribute *SrcPort* identifies the port number of the output port in a ***Block*** from where the ***Line*** originates.
- *Points*: The attribute *Points* give the relative pixel coordinates of the ***Line*** with respect to the *SrcBlock* and the *DstBlock*.
- *DstBlock*: The attribute *DstBlock* contains the Name of the ***Block*** where the ***Line*** ends.
- *DstPort*: The attribute *DstPort* identifies the port number of the input port in a ***Block*** where the ***Line*** ends.

On similar lines, the element ***Branch*** is modeled as the class *branches*. The instance(s) of this element if present is contained in the element ***Line***. The common and required properties of this element are *DstBlock* and *DstPort* and are modeled as the attributes of the metamodel class *branches*. The attributes *DstBlock* and *DstPort* have the same meaning as that of the element ***Line***.

There are several properties of the elements ***Block***, ***Line*** and ***Branch*** which are specific to certain instances of its type. These properties are also needed for the transformation and have to be modeled in the metamodel. For example the property *Value* is present only for certain instance of the element ***Block***, like the ***Block*** instance with the *BlockType* as '*Constant'*. However these properties are not modeled as attributes

of the class. In order to model such properties in the metamodel a separate class in the metamodel was designed. This class is named as *properties*. The class *properties* has only two attributes *Name* and *Value.* The attribute *Name* is designed to contain the Name of the property and the attribute *Value* is designed to contain the value of the property.

The elements of the model described above are sufficient for developing the transformation definition for the forward transformation from Simulink to Scicos. However, for reconstructing a Simulink model from its metamodel requires many other elements to be present in the metamodel, such as ***GraphicalInterface***, ***Outport*** and ***Inport***. These elements are also modeled as classes in the metamodel. The ***GraphicalInterface*** is modeled as the class *graphicalInterface*. The properties which are required include

- *NumRootInports*
- *NumRootOutports*
- *ParameterArgumentNames*
- *ComputedModelVersion*
- *NumModelReferences*
- *NumTestPointedSignals*

These properties are modeled as attributes of the class *graphicalInterface*.

The element ***GraphicalInterface*** contains the elements ***Outport*** and ***Inport***. The number of instances of elements ***Outport*** and the number instances of elements ***Intport*** present is mentioned in the properties *NumRootInports* and *NumRootOutports* respectively.

The element ***Outport*** is modeled as class *outport*. It has the porperties *BusObject*, *BusObjectAsStruct* and *SignalName* modeled as its attributes. The element ***Inport*** is modeled as class *inport.* It has the properties *BusObject* and *SignalName* modeled as its attributes.

Attribute names have a value which may be text, int or any kind of datatype corresponding to the EMF datatypes.

The results of the metamodeling are discussed in the next chapter.

5.7 Summary

In this chapter, the process of development of MOF based metamodels has been discussed. First a Scicos model is developed. Based on this model a metamodel for Scicos is developed. The transformation definition requires a metamodel for Simulink which is not available publicly. Here, a Simulink model is developed. This Simulink model is analysed and a metamodel for Simulink is developed. The language used for the development of metamodel is EMF Ecore. The tool used for the metamodel development is Ecore's Graphical Modeling Tool. The flow charts for the development of the metamodels are presented. These developed MOF based metamodels are proposed for the use in the transformation definition of the current work. The results are discussed in the chapter 6.

Chapter 6
Results of Metamodel Development

A detailed process of the development of the metamodels for Scicos and Simulink was discussed in chapter 5. This chapter discusses the results of the development of metamodels of Scicos and Simulink.

6.1 Results of the development of Scicos metamodel

The result of the analysis of the Scicos model has resulted in retrieval of the information required for the development of an EMF based metamodel for Scicos. Based on the analysis, the classes ScicosModel, Object, Blocks, Links, CodeGen, Parameters, Options and SuperBlock are modeled. The relationships between them are shown using the containment references and inheritance. The class ScicosModel contains the classes Parameters, CodeGen and Object. The class Parameter contains the class Option. The class Object contains the classes Blocks and Links. The class Blocks contains the class SuperBlock which inherits the properties from the class ScicosModel. The EMF based metamodel developed for Scicos is presented in Figure 6.1.

6.2 Results of the development of Simulink metamodel

The result of the analysis of Simulink is used in the development of the metamodel. The Simulink metamodel starts with the root element, Model. This root element has a composition reference *System* to the class system. The system in turn has composition references *GraphicalInterface*, *Block*, and *Line* to the classes graphicalInterface, blocks and lines respectively. The class graphicalInterface contains the classes outport and inport. The class *blocks* has references to system and properties. The classes *lines* and branches have composition references to the class properties. The class *branches* has composition reference to itself. The metamodel developed is presented in Figure 6.2. The figure shows the Graphical representation of the MOF based metamodel for Simulink.

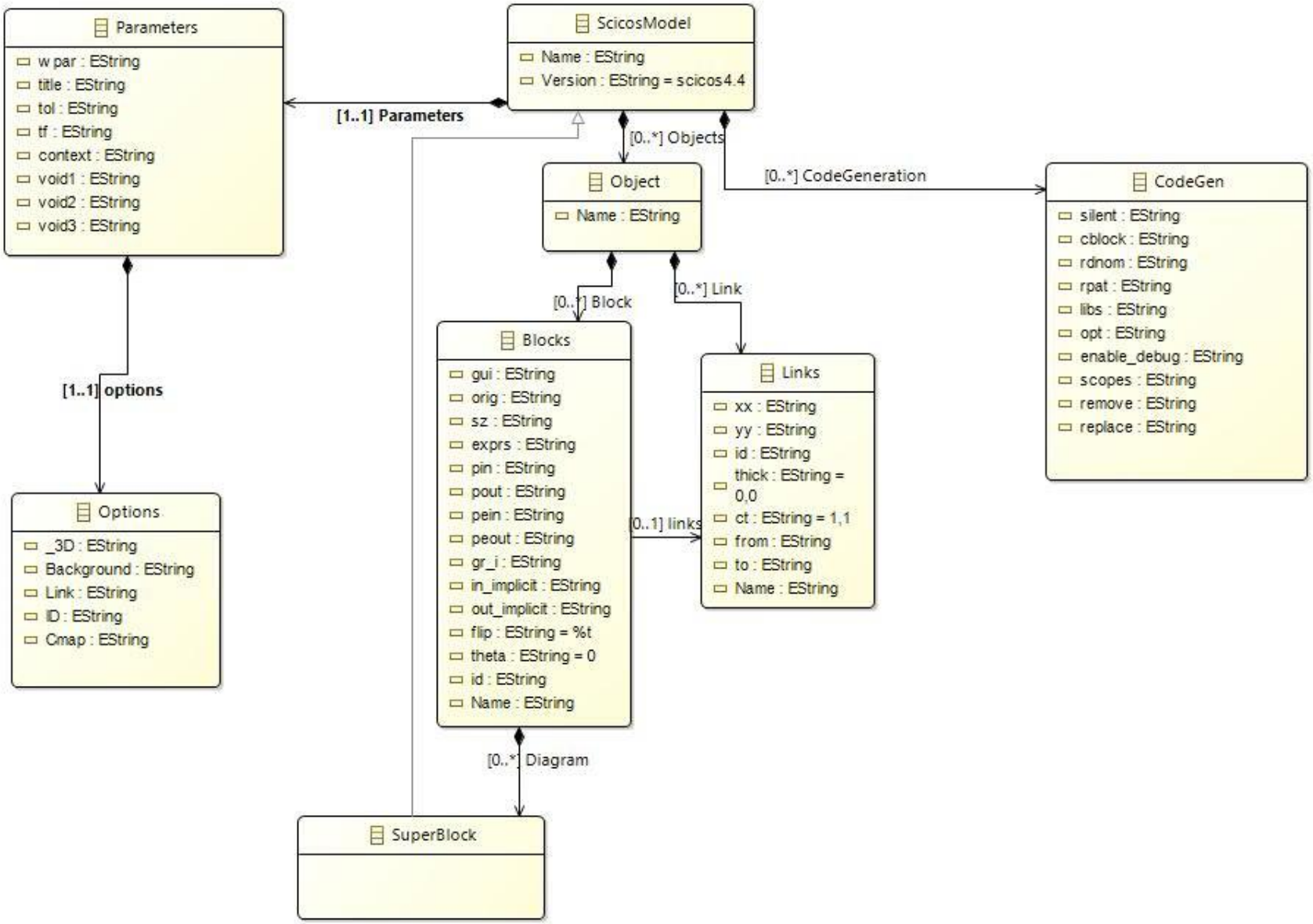

Figure 6.1: Scicos Metamodel

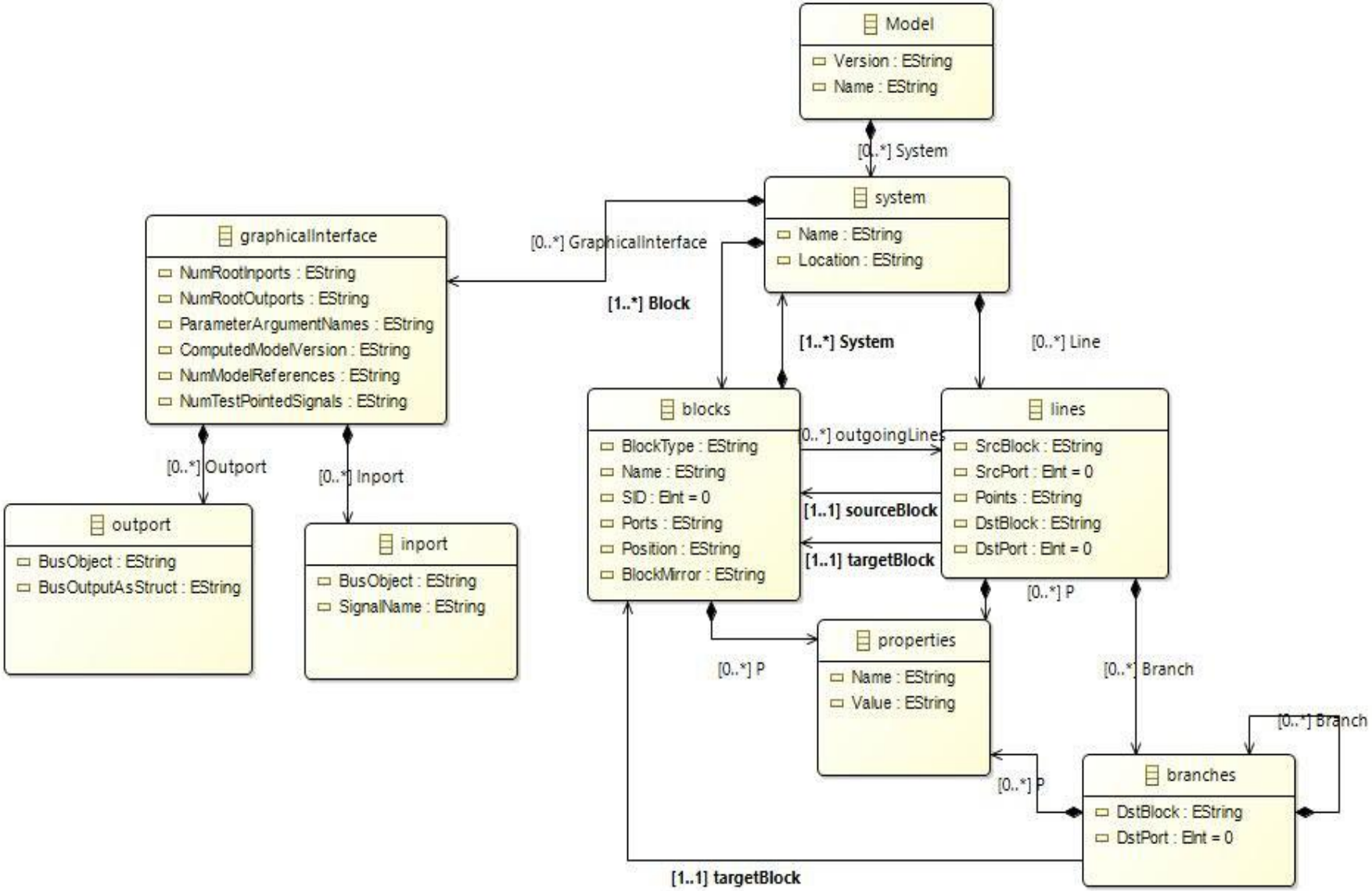

Figure 6.2: Simulink Metamodel

6.3 Verification and Validation of the metamodels

The metamodels developed here are for the purpose of developing a transformation definition. Hence, the validation of the metamodels can be done only after verifying a transformation of model. However, the metamodels are verified against the specifications using which the metamodel is developed. In order to verify the metamodels, a model is created using the tools Simulink and Scicos. Similarly, the model instances are created from the metamodels. The model developed using Simulink is compared against the instance created using the Simulink metamodel and the model developed using the Scicos tool is compared against the instance created using the Scicos metamodel. If the model instances are able to reflect all the elements, their properties and the relationships, then the metamodel can be assumed to be correct and verified.

6.3.1 Simulink Metamodel

A model of a subsystem of the cross wind assistance system is used in the verification of the metamodel. The figure 6.3 presents a Simulink model which has been used in the verification of the Simulink metamodel. The instance model developed using the metamodel developed is presented in the figure 6.4.

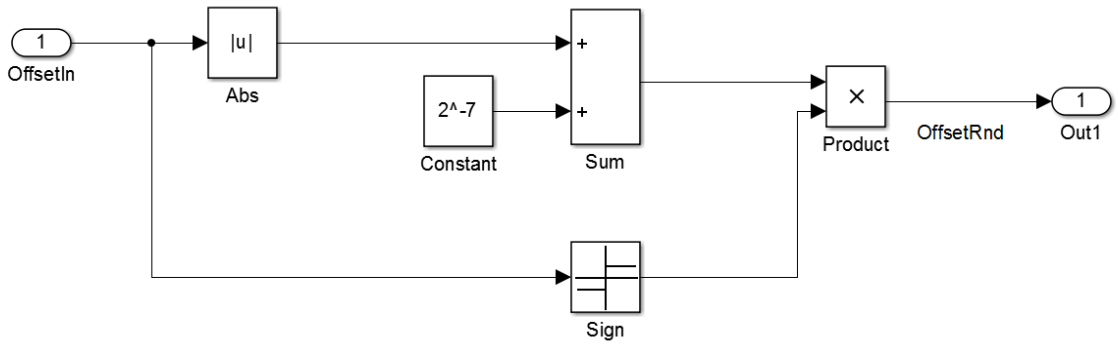

Figure 6.3: A subsystem designed using Simulink

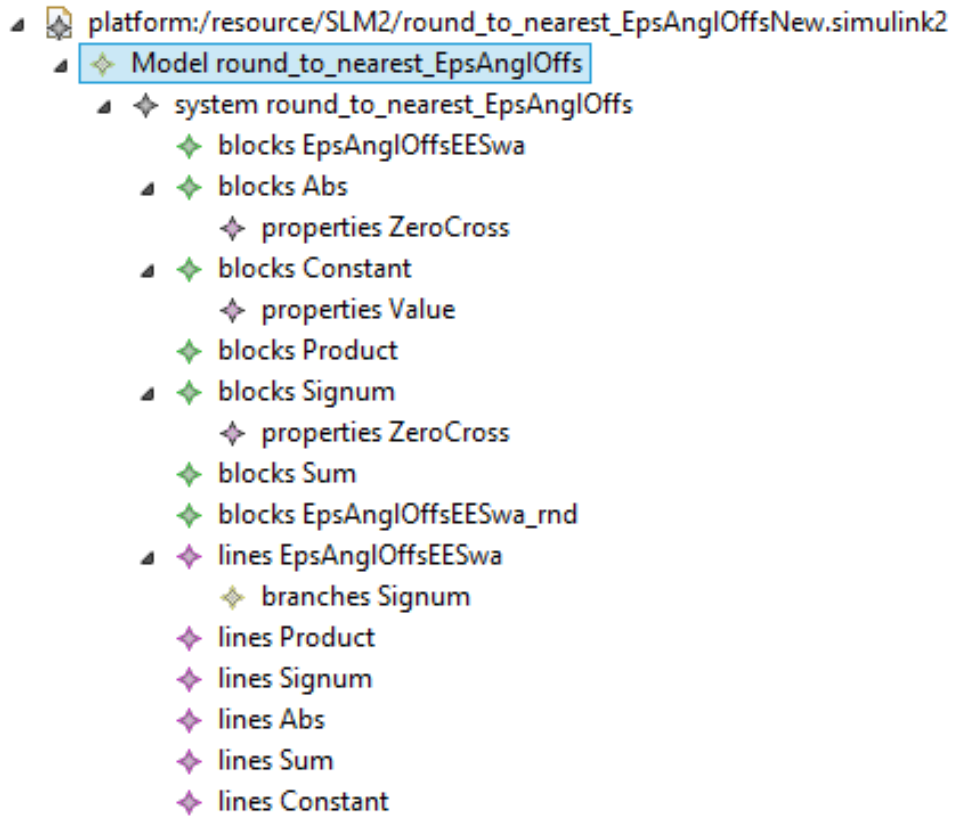

Figure 6.4: Model instance created using the developed Simulink metamodel

Comparison with the instance model

The metamodel instance presented in the figure 6.4 reflects all the properties of the model presented in the figure 6.3, which is required for performing the transformations. This verifies the Simulink metamodel developed. The metamodel can be validated with a successful transformation of the Simulink metamodel instance into a Scicos metamodel instance.

6.3.2 Scicos metamodel

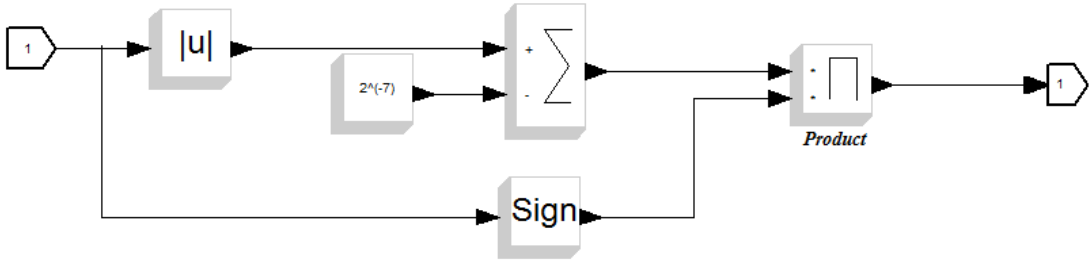

Figure 6.5: A subsystem designed using Scicos

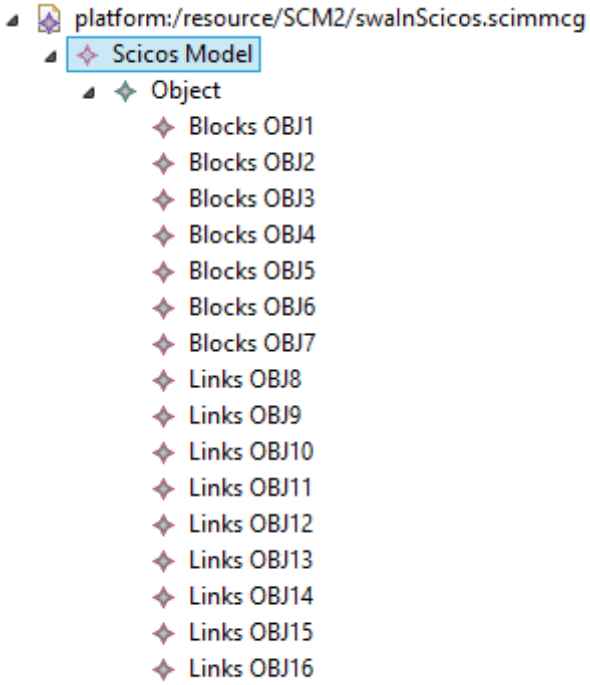

Figure 6.6: Model instance created using Scicos metamodel

A model of a subsystem is used in the verification of the metamodel. The figure 6.5 presents a Scicos model which has been used in the verification of the Scicos metamodel. The instance developed using the metamodel developed is presented in the figure 6.6.

Comparison with the instance

The metamodel instance presented in the figure 6.6 reflects all the properties of the model presented in the figure 6.5, which is required for performing the transformations. This verifies the Scicos metamodel developed. The metamodel can be validated by reconstructing a Scicos model using an instance of the metamodel. This has been discussed along with results of the transformation in next chapters.

6.4 Summary

In this chapter the results of metamodel development for Simulink and Scicos have been discussed. The instances of the model are developed using the metamodel. The results are verified by comparing the instance against the models developed using the respective tools. Thus, verifying the metamodels.

Chapter 7
Implementation of Transformation

A detailed process of the development of the metamodels for Scicos and Simulink were discussed in chapter 5 and their results were discussed in chapter 6. A Scicos metamodel was constructed based on the developed Scicos model. A Simulink metamodel was constructed using reverse engineering a Simulink model. This metamodel was developed using EMF Ecore tools. This chapter gives an insight into the implementation of the transformation.

Transforming a model from another model is same as generating a model by using information from the other model. In order to perform the transformation the elements in the source model must be mapped to the elements in the target model. This mapping can be done using the metamodels of the source and the target models. The metaclass of the source model elements must be related to the metaclass of the elements in the target model.

7.1 Transformation rule requirements

A transformation definition provides the rules for the transformation. It has to contain all the information needed for the transformation and requires:

- A source metamodel conforming to the source language
- A target metamodel conforming to the target language
- A set of elements from the source metamodel having names
- A set of elements from the target metamodel having names
- Conditions that must hold in the source metamodel without which the transformation rule will not be applied
- Conditions that must hold in the target metamodel without which the transformation rule will not be applied
- A rule for mapping the elements in the source model to the elements in the target model

7.2 Development of Forward Transformation

In the present work, the transformation from Simulink to Scicos forms the forward transformation. The definition of the transformation is written using QVTo. QVTo requires MOF based source and target metamodels. The metamodels were developed as explained in the previous chapter and it is being used here for the transformation.

The process of forward transformation carried out is as explained in this section.

7.2.1 Analysis

In order to develop the transformation definition and to understand the rule for transformation, further analysis was carried out. These analyses were especially required to understand the mapping relationship between the elements of the target metamodel and the source metamodel. The analysis was extended for the transformation as well. Some points useful for the transformation are listed below:

- A Simulink model can have only one element called '*Model*'. It implies that a transformation rule must restrict the transformation to those models having only one '*Model*'.
- A '*Model*' of the Simulink model is mapped to a '*ScicosModel*' of the Scicos model.
- A '*System*' is mapped to an '*Object*'
- There can be only one element '*System*' as an immediate child element of '*Model*'. However, a model may contain many systems called Subsystems within the '*Block*'s. Also it can be noted that only one Subsystem may be present in a given '*Block*'.
- A '*Block*' in a Simulink model can be mapped to a '*Block*' in a Scicos model.
- A '*Line*' of a Simulink is mapped to a '*Link*' of a Scicos model.
- The element '*Branch*' of the Simulink model has no equivalent element of the Scicos model. However, a Scicos model has a '*Block*' called 'Split' which can be used for branching. Hence, a '*Branch*' can be mapped to a 'Split' '*Block*' and two '*Link*'s.

- A '*Block*' of type subsystem is mapped to a '*Block*' of Scicos containing a '*SuperBlock*'.

In order to reconstruct a model in Scicos, all the required attributes of a class representing the class have to be mapped. While mapping or assigning values to the attributes it may be noted that Simulink has more attribute values than that would be required by the Scicos model. Hence, it would be helpful to consider the attributes of the classes in Scicos metamodel first. Based on these attributes the attributes of the Simulink metamodel may be mapped/ assigned to the attributes of the Scicos metamodel.

The root class of the Scicos metamodel is *ScicosModel*. It has attributes *Name* and *Version*. *Name* is name of the model and the name of the Simulink model may be assigned during the transformation. *Version* being the version of the Scicos model using which the model was developed. Since, in the development of the current transformation rule, Scicos version 4.4 is used, the value of the *Version* is set as 'Scicos4.4'.

The attributes of the Scicos '*Block*' are extracted using the analysis and processing of the source model. Some of the attributes can be deduced from the attribute values of the source model, but many others have to be matched with the corresponding block instances of the source and target models. *gui* can be obtained from the *BlockType* attribute of the Simulink model which has to be mapped based on corresponding block instances. As mentioned in the previous chapter *orig* is the position of the *Block* from the origin and can be obtained by using the attribute *Position*. The attribute *sz* denotes the size of the *Block*. The values for the attribute *sz* can be calculated using *Position.* The attributes *pein*, *peout*, *pin* and *pout* references the *Link*s which were mapped from the *Line* and *Branch* elements of the Simulink. The *in_implicit* and *out_implicit* indicates the number of input and the output ports a *Block* contains and can be obtained using the attribute *Ports* of the Simulink *Block.*

The attribute *Name* of the Simulink *Block* provides the value for *id.* The value for *id* may be left blank if it has no Name given by the user. The attribute *flip*, indicates if the *Block* is flipped, this can be obtained by checking the *BlockMirror*. If the *BlockMirror* is set to *on*, then it indicates that the *Block* is flipped, otherwise it is not flipped. The attribute *theta* provide the angle of rotation of the *Block*, and in the current work it has been set to zero for all instances of the *Block*s. The attributes *exprs* and *gr_i* are specific

to the type of *Block*. The information on the type of Scicos *Block* can be obtained by using its own attribute *gui* or the value of the attribute *BlockType* from Simulink. However, value of *gr_i* is specific and defined by Scicos. Hence using the information on the type of the Block, the value of gr_i has to be obtained or matched. The value of *exprs* is specific to the type of the Block and also to the specific instance of the type of the Block. The format of *exprs* is used from the definition of the specific instance provided by Scicos tool and the value has to be obtained from different other attributes of the *Block*. These attributes are designed as the class *Properties* in the metamodel.

Some attributes of the Class *Link* are obtained from the source model, some others are defined in the target model itself. The *Name* attribute is generated during the transformation. The attributes *xx* and *yy* are calculated based on the *SrcBlock* and *DstBlock* attributes of the source metamodel, the position and size of the *Block*s which the *Link* connects, the port number which it connects and the number of ports in the *Block*s it connects. The attributes *to* and *from* are constructed using the attributes *DstBlock*, *DstPort*, *SrcBlock* and *SrcPort*. The attributes *thick* and *ct* are kept constant at (0,0) and (1,1) respectively.

7.2.2 Transformation Mappings using QVTo

As mentioned above the mappings form the most important part of the transformation in QVTo. In the following paragraphs, the QVTo mappings developed for the current transformation is explained.

```
transformation SL2SCTransformation(in Source: SLM, out Target: SCM);

modeltype SLM uses simulink2('com.example.simulink2') where {
      self.objectsOfType(Model)->size() = 1 };

modeltype SCM uses "com.example.scimmC";
```

The declaration for the forward transformation is as presented above. The declaration `modeltype` declares the metamodel references that are used in the transformation. The first statement declares a transformation and has the name provided as `SL2SCTransformation`. The keywords `in` and `out` are used to indicate which of the metamodel declared are the source and target metamodels.

The transformation begins with the `main()` function and is presented below. This function is used to set the environment variables and the first mapping of the transformation is called here. The `rootObjects()` selects the root of the Source metamodel. However, by using `[SLM::Model]`, only objects called *Model* are selected as the root object.

```
main()
{
	Source.rootObjects()[SLM::Model] -> map ModelToSciMod();
}
```

The first mapping in the current transformation is between the root objects *Model* and *ScicosModel* of the respective metamodels. A mapping declaration consists of a name of the mapping, class name of the element being transformed and the class name of the element of the resulting target model. While declaring a mapping it should be noted that the name of the mapping should be uniquely declared.

```
mapping Model::ModelToSciMod(): ScicosModel {...}
```

A mapping may have conditions using the key words **`where`** and **`when`**. This is followed by the body of the mapping. It may contain three sections **`init`**, **`population`** and **`end`**. The **`population`** section contains all the actual mappings needed for the transformation.

```
mapping Model::ModelToSciMod(): ScicosModel
{
	Name := self.Name;
	Version:= "scicos4.4";
	var NameString="OBJ";
	result.Objects:=self.System -> map Object2System(NameString);
}
```

In the mapping `ModelToSciMod()`, between the root objects, *Model* and *ScicosModel*, the attribute `Name` of the source object is copied into the attribute `Name` of the target object. As mentioned previously, the attribute `Version` is assigned a constant value `"scicos4.4"`. The mapping of the child elements is done here using the Composition Relation. It maps the System in Simulink to Object in Scicos. It may be noted here that the *Block*s and *Link*s have unique identifier in the Scicos model. This unique identifier is a number and always starts from one and increments sequentially. In QVTo a string can be associated to an increment counter called `incrStrCounter(String)` for generating a sequence of numbers. This counter can be

used to generate the unique identifier. A unique identifier once allotted cannot be used again which means that if an identifier is allotted to an instance of a *Block*, then that identifier cannot be assigned to any other instances of *Block* or *Link*. The String variable `NameString` shown in the code snippet is used to associate the increment counter so that the identifier is not duplicated. While mapping the System to Object, the variable `NameString` is passed as a parameter along with the mapping.

The mapping of `Object2System` contains calls for the mappings for *Block* to *Block*, *Line* to *Link*, *Branch* to *Block* and *Branch* to *Link*. The first among these mappings is from *Block* to *Block* declared `Block2Block(NameString)`.

Mapping Block to Block

The mapping `Block2Block(NameString)` maps the Simulink *Block* to Scicos *Block*. The first attribute is the `Name` which uses the unique identifier. The string parameter `NameString` is used to invoke the function *`incrStrCounter`*`(NameString)` which increments the value associated with the parameter and returns an integer. This value is concatenated with the String to form the vale for the attribute. The attribute `theta` is set to `"0"`. While assigning or reading the values of any attributes from the source model, the keyword **`self`** has to be used to indicate that the attribute value being accessed belongs to the source model. The statements shown below demonstrate the use of **`self`**.

```
id:= self.Name;
var btype := self.BlockType;
```

The values for the attributes `in_implicit` and `out_implicit` are constructed using the attribute `Ports` of the Simulink *Block*. Since every Simulink block has a `Name` provided by the user. This name is allocated to the attribute `id` of the Scicos *Block*. The attribute `Position` is used to calculate the attributes `orig` and `sz`. If the attribute `BlockMirror` is set to “on” then `flip` is set to “%f” else to “%t”. The values of the attributes `pin` and `pout` can only be determined after the *Line* is mapped to the *Link*. Hence, setting the values to these attributes is explained later along with the mapping of *Line* to *Link*.

The attributes `gui`, `exprs` and `gr_i` are dependent on the type of the instance of Block being transformed. Since there are several different types of *Block*s, these

attributes have to be mapped on a case to case basis. Hence, a switch statement is used to assign values to such attributes. The attribute `BlockType` tells us the type of *Block* being transformed. This directly corresponds to the information of `gui` and has to be mapped to the corresponding value for Scicos. The following points provide information on the transformation of the subset of the *Block*s:

- `Outport` and `Inport` : The `Outport` and `Inport` corresponds to OUT_F and IN_F of the Scicos model. The value of the `exprs` is constructed based on the number of output ports in an Outport Block and on the number of input ports in an Inport Block.

- `Sum`: The `Sum` corresponds to the addition block SUMMATION of the Scicos model. The value of the gui is `SUMMATION`. A summation block may be two input block or multi input block. The inputs some may be positive or negative. All these information is available in different properties of the Simulink block which are mapped to the *Properties* class of the Simulink metamodel. These are accessed using the containment reference from the class *Block*. The transformation traverses through the instances of the *Properties* and checks if there is any property with the `Name` "Inputs". On the presence of this property, the value for the attribute `exprs` is set. If there is no such property present then the default value is used, which is set as two positive inputs.

- `Product`: The `Product` corresponds to the multiplication block of the Scicos model. The value of the gui is `MATMUL`.

- `Constant`: The `Constant` corresponds to the Constant block of the Scicos model. The value of the gui is `CONST_m`. As explained for the addition block, the value for the constant block is also present as an attribute of the *Properties* class with the attribute called `Name` having the value "*`Value`*" and the attribute `Value` contains value for the constant block. Hence, the attributes of the *Properties* have to be traversed to find this value. Based on this, the value for the attribute `exprs` is constructed.

- `Bus Creator`: The `Bus Creator` corresponds to the BusCreator block of the Scicos model. The attribute id is set to "BUS". The value of the exprs is set based on the number of inputs the bus creator has which is obtained from the attributes of *Property*. If the value of the attribute `Name` is "Inputs" then the corresponding value of the attribute `Value` gives the number of inputs.
- `Bus Selector`: The `Bus Selector` corresponds to the `Bus Selector` block of the Scicos model. The attribute id is set to "DEBUS". The value of the `exprs` is set based on the number of outputs the bus selector has which is obtained from the attributes of *Property*. If the value of the attribute `Name` is "Outputs" then the corresponding value of the attribute `Value`, provides the number of outputs.
- `SubSystem`: The `SubSystem` corresponds to the Super Block of the Scicos model. The value of the `exprs` is not set and left blank. A super block may contain many *Block*s and *Link*s within it. Since these elements belong to the Super Block, their unique identifiers within the Super Block are also unique. They have the identifier of the Super Block as their prefix. In order to generate this unique identifier, the `Name` of the SuperBlock is used as the reference string. In order to map the *Block*s and *Link*s within the Super Block another mapping `map Subsystem2SuperBlock(blockName)` is called and the reference string is passed as the parameter to the mapping.
- `UnitDelay`: The `UnitDelay` corresponds to the Delay block of the Scicos model. Similar to the other Blocks mentioned above, the attributes of this block are also populated.

Mapping Line to Link

The mapping `Line2Link(NameString)` maps the Simulink *Line* to Scicos *Link*. As explained for the *Block*, even *Link* has a unique identifier and is associated to the same string `NameString`. Hence, the counter associated to `NameString` provides the unique identifier, using which the value for the attribute `Name` is formed. The attributes `thick` is set to "0,0" and `ct` to "1,1" as they are only needed for display purposes.

The Simulink *Line* has attributes *SrcBlock* and *SrcPort*. They contain the Name and the Port details of the Simulink *Block* from where this *Line* originates. This is used to

identify the Scicos *Block*, from which the *Link* originates. Similarly, the *Line* also has the attributes *DstBlock* and *DstPort* that provide information regarding the destination *Block*s. The process of identification of the Scicos *Block* is by using resolve constructs.

In order to identify a Scicos *Block*, it needs to be first accessed from the mapping and then **resolve** constructs can be used in the retrieval of the *Block*s. The access to the Scicos *Block*s can be made only by accessing the mappings that created the *Block*s. Hence, the function *container()* is used to traverse to the parent class which is a Simulink *system* here. The Simulink *Block*s are first accessed via *system*. These Simulink *Block*s are used by the **resolveIn** construct to retrieve the Scicos *Block*s already mapped. These Scicos *Block*s are then compared using the attribute *id* with the *SrcBlock* for source and *DstBlock* for the destination *Block*s, to identify and obtain the relevant Scicos *Block*.

```
var tempBlocks :=
self.container().oclAsType(system).Block.resolveIn(SLM::blocks::Block2B
lock, SCM:Blocks);
      tempBlocks->forEach(blkObjects){

            if(blkObjects.id=self.SrcBlock) then{
                  log("sourceModel: " + blkObjects.id);
                  sourceMode:=blkObjects;
            }endif;

            if(blkObjects.id=self.DstBlock) then{
                  log("targetModel: " + blkObjects.id);
                  targetMode:=blkObjects;
            }endif;

      };
```

The attributes `xx`, `yy`, `to` and `from` are calculated and obtained using the Scicos *Block*s identified as the source and the destination blocks. The attribute `orig` and `sz` of both the source and the destination *Block*s in addition to the SrcPort are used to determine the `xx` and `yy`. The unique identifier of theses blocks and their port numbers are used to formulate the values of the attributes `to` and `from`.

The attributes `pin` and `pout` of the Scicos *Block*s were not populated during the mappings of the *Block*s. Since, the Link is created in the mapping *Line* to *Link*, these *Link*s have to be now mapped to the attributes pin and pout.

Mapping a Branch

The elements *Line* and *Block* of Simulink have corresponding elements in the Scicos block and have a direct mapping between the classes. However, a branch which is contained in a line in a Simulink model has no corresponding class in the Scicos model. Hence, it has to be mapped to a class that can substitute it in the Scicos model. During the analysis, it has been found that Scicos uses a block called Split for branching its links. Hence, a branch in a Simulink model has to be mapped to this Split. Also, a split can have two or three outputs, but in the present work splits with only two outputs have been used for mapping branches. Hence, a branch should also be mapped to two links which are connected to the output port of the split. The mappings of the branch to block and branch to the links have been explained in the following paragraphs.

A branch is contained in a line and also, every line is mapped to a link which contains a reference to the unique id of the block from where it originates and to where it terminates. Hence, during the mapping of a line to link, the unique id of the block mapped from the branch is required. Hence, a mapping from branch to block is invoked within the mapping from line to link. This creates a split block for each branch and all those attributes are also mapped for which information is available. The remaining are mapped when the mapping is invoked from the containment reference.

Mapping Branch to Block

As explained above, a branch is mapped to a block. The attribute `Name` of the Split block also uses the unique identifier sequence. Hence, the same string counter as used for the other elements is used to obtain it. Most of the attributes for a split block are constant for all the instances. Among them are exprs which has a blank value, gui has the value `SPLIT_f`, the attribute sz which denotes the size is set to [0.33331, 0.33331], flip to “%t” and theta to 0. Since, the number of inputs is only one, and in the current transformation split of only two outputs is used, the in_implicit and out_implicit are assigned a value to denote one input and two outputs respectively. The attribute `gr_i` is also assigned a fixed value.

The attributes pin and pout refers to the incoming links and the outgoing links in the split block. Since the *Line* which contains the *Branch* is already mapped, its

corresponding unique id of the Scicos link is used to populate the attribute `pin`. The challenge is to populate the attribute `pout` which contains the references to the outgoing links. A variable is used to extract the mapping information of the Branch2link as presented below.

```
var pin1:=self.map Branch2Link(NameString);
```

The actual working of the statement presented above is to retrieve the mapping of the Branch to Link. However, if the mapping is not present then a mapping is done at that point from Branch to Link, with possible attributes populated. The remaining attributes are populated later when the same mapping is attempted again.

Mapping Branch to Link

The mapping from *Branch* to *Link* involves two mappings. A split requires two *Link*s as outputs and hence two *Link*s for every *Branch* is being mapped. The attributes `ct` and `thick` are kept constant at "1,1" and "0,0" respectively. These are line display attributes as mentioned previously. Again the attribute `Name` uses the unique identifier and the namestring. This mapping is in sequence to the *Block* from where the two *Link*s originate. The unique identifier number preceeding this, will be the source block. Using this information the attribute `from` is constructed. The attribute `DstBlock` of the branch is used along with the **`resolve`** construct to retrieve the destination Scicos block, which is used to construct the value for the **`to`** attribute. The attributes `orig` and `sz` of the Scicos block and the `DstPort` of the Simulink branch is used in the calculation of the `xx` and `yy` coordinate values. This mapping is done twice for every branch.

Mapping Subsystem to SuperBlock

SuperBlock is an inherited subclass of the *ScicosModel*, hence mapping a *Subsystem* to a *SuperBlock* becomes a challenge. The solution is to map the *Subsystem* in line with the parent class *ScicosModel*. Here, a Subsystem is actually the element system of the Simulink model. Hence, to map the subsystem to superblock, a mapping is actually done on system to Superblock. The contained classes are then mapped as per the superclass references.

7.3 Development of Reverse Transformation

The transformation of model from Scicos to Simulink constitutes the reverse transformation. In the reverse transformation, a Simulink model has to be constructed using the information from Scicos model. This transformation as in forward transformation is also performed using their respective MOF based metamodels developed using EMF Ecore in this work.

In order to perform reverse transformation, the information required to reconstruct a Simulink model has to be present in the Simulink metamodel. There were many classes in the Simulink metamodel that were not used in the forward transformation as the required information was already present in other elements of the metamodel. These classes are *graphicalInterface*, *outport* and *inport*. Hence, these classes also need to be mapped from the Scicos model in order to perform the transformation.

7.3.1 Transformation Mappings using QVTo

Similar to the forward transformation the reverse transformation also starts from the declaration of the transformation and modeltype. This is followed by the main() function from where the mapping begins. In the Scicos model, the element `ScicosModel` is the root element. Hence, the mapping is started from the mapping this root element of Scicos model to the root element of the Simulink model.

Mapping ScicosModel to Model

The mapping `SciModToModel()`, which maps the `ScicosModel` to `Model` is invoked in the main() function. The attributes `Name` of the `Model` is assigned the value of the attribute `Name` of the `ScicosModel`. The current work has been carried out using the Simulink version 7.8. Hence, the attribute `Version` is assigned the value `7.8`. In Simulink every *Block* has a unique name. A string is used along with the function `incrStrCounter()` to generate unique number for each *Block*. This unique number is appended to the `BlockType` of the *Block* to generate a unique name for each. There are two other mappings which are invoked within this mapping. One is the

`Object2graphicalInterface()` and the other is `Object2System()`. The mapping `Object2graphicalInterface()` maps class *Object* to class *graphicalInterface*.

Mapping Object to graphicalInterface

Mapping of class Object to class graphicalInterface involves assigning the values of all the required attributes of the class graphicalInterface. Many of the attributes of the graphicalInterface remains unchanged for most instances of Simulink models. Hence, they are given a fixed value. The attributes `NumRootInports` and `NumRootOutports` contain the count of the number of Blocks of type *Inport* and *Outport*, which are accessible from the external environment. Each of these Inport and Outport are contained in the class *graphicalInterface*. Hence, another mapping operation is performed for each these *Inport*s and *Outport*s. The mappings `Object2Inport` and `Object2Outport` map the *Object* to *Inport*s and *Outport*s respectively, passing the `id` of the blocks as parameters. Within these mappings the attributes of the classes are populated.

Mapping Object to System

Mapping of Object to system contains the mappings from *Block* to *blocks* and the mappings from *Link* to *lines*. These mappings form the core of the reverse transformation.

Mapping Block to Block

The mapping from *Block* to *blocks* contains, the assignment of values to the attributes of the Simulink Block using that of the Scicos Block. The attributes that are populated first include `Name`, `SID`, `Ports`, `BlockType`, `Position` and `BlockMirror`. The name is the unique name of the Block. It is constructed using the unique identifier number appended to the `id`, if any given by the user, for the Scicos Block. If no `id` is specified then the `BlockType` is appended. `SID` is the unique identifier of the Simulink Block. Hence, the unique number generated is assigned to the Block. The attributes `in_implicit` and `out_implicit` are used for constructing the value for the attribute `Ports`. For every port a letter ‘E’ is used in the `in_implicit` and `out_implicit`. Counting the number of ‘E’ provides the number of input and output ports of a block.

This is used for populating the attribute `Ports`. The attribute `BlockType` is dependent on the type of Block. The attribute `gui` of the Scicos Block is used to determine the `BlockType`. The value of this attribute is defined by Simulink. Hence, `gui` must be mapped to the possible values of the Simulink model and then assigned to the attribute. The Position provides the co-ordinate values of the element in the Simulink model. The attributes `orig` and `sz` are used in the calculation of the values for the attribute `Position`. If the `flip` attribute of the Scicos model is "%f", then the `BlockMirror` is set to "on". The attributes required for the Simulink model are also mapped based on the individual elements using the case statement, which uses the `BlockType` as its parameter.

Mapping Link to Line

In the mapping from *Link* to *lines*, the attributes `SrcBlock`, `SrcPort`, `DstBlock`, `DstPort` and `Points` have to be populated. The attributes, `from` and `to` contain the information of the source and destination blocks respectively. The values for the `SrcPort` and `DstPort` can be directly extracted. The attribute `SrcBlock` and `DstBlock` require the name of the block. All the blocks are retrieved and the values in the attribute from and to are compared with the respective attributes of the blocks. The matching Scicos block is then used in identifying the Simulink block with the help of the **`resolveoneIn`** construct. The attribute `Points` is calculated using the `Position` attribute of the source and the destination blocks.

The results of the above transformation are presented in the next chapter.

7.4 Intermediate transformation

A model was developed using the Simulink tool. The Simulink model was saved as an xml file using the save `save_system()` function. The format of the obtained xml file is as defined by the Simulink tool. However, the current transformation definition requires the xml file in form defined by the metamodel of Simulink. Hence, an intermediate transformation was developed using the language python. The Simulink xml file was provided as input to the intermediate transformation program, which converts it into the format of the metamodel instance.

The obtained metamodel instance is used as the input for the forward transformation definition. The output of this transformation is a model instance conforming to the Scicos metamodel. Again this model instance cannot be accessed directly by the Scicos tool. It again requires an intermediate transformation to obtain the Scicos model. An intermediate transformation was developed using python to obtain the Scicos model from the intermediate model instance. This model obtained after the intermediate transformation can be accessed using the Scicos tool.

7.5 Summary

This chapter presents in detail the development of the transformation between Scicos and Simulink. The Metamodels developed in the previous chapter are used to develop the transformation definitions. The tools and languages mentioned in chapter 4 have been used in the development of the transformation. The transformation from the Simulink to Scicos is the forward transformation. The transformation from the Scicos to Simulink is the reverse transformation. The transformation definition is developed using QVT operational mappings. The elements of the source model are mapped to the elements of the target model. The constraints and the relationships are also mapped.

Chapter 8
Results of Model Transformation

An implementation of the transformation between the model instances of Simulink metamodel and Scicos metamodel was presented in chapter 7. The forward transformation from Simulink to Scicos was carried out using a few models in Simulink. The results obtained after the transformation are discussed in this chapter.

8.1 Results of Forward Transformation

The results of the forward transformation for the Simulink models, presented in figure 8.1 and figure 8.5, into Scicos models are discussed in the following sections.

Example 1: The transformations was applied on a Simulink model presented in figure 8.1

This model was developed using Simulink tool and was saved as an XML file using the MATLAB command `save_system()`. The file obtained is a Simulink model in XML format. This is used as an input for the intermediate transformation. The intermediate transformation is basically a restructuring of the XML tags, so that the model conforms to the Simulink metamodel. The output of the intermediate transformation is another XML file. This is called as intermediate model IM1 and is presented in figure 8.2.

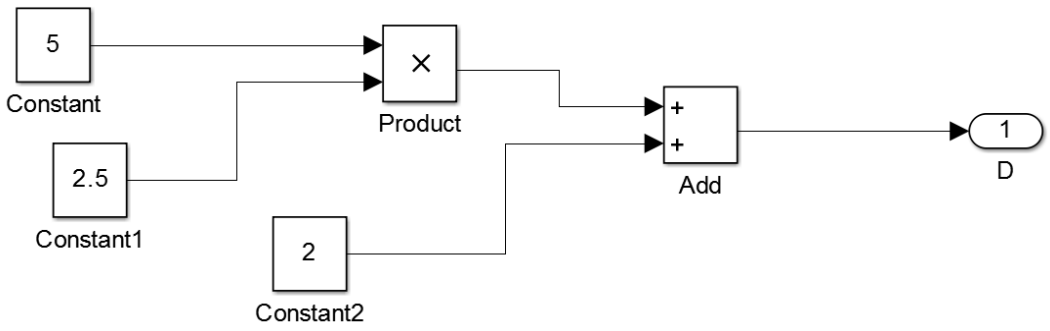

Figure 8.1: A Simulink model

The intermediate model IM1 conforms to the Simulink metamodel. IM1 is used as input to the transformation definition. When the transformation rule is applied to this model IM1, we get an output, which is another model. This model conforms to the Scicos metamodel and is named as Intermediate model IM2 here. IM2 is presented in the figure 8.3. The model IM2 is also in the form of XML file. But, it cannot be directly accessed by the Scicos tool as the IM2 does not have the exact formatting as required by the Scicos tool. Hence, another intermediate transformation is performed on IM2. The output of this intermediate transformation is a model file. This model file is an XML file in the format readable by the Scicos tool. The obtained Scicos model is presented in figure 8.4. The correctness of the model and the transformation is discussed in the section Validation and Verification.

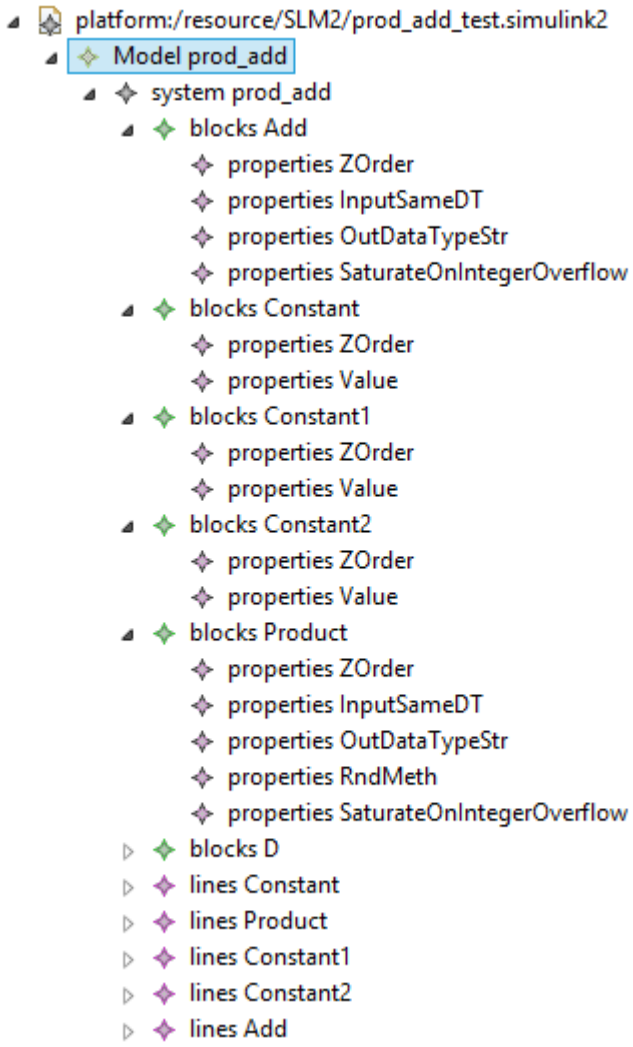

Figure 8.2: Intermediate model IM1

The intermediate model IM1 represented as an XML file is presented below:

```
<?xml version="1.0" encoding="UTF-8"?>
<simulink2:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:simulink2="com.example.simulink2" Version="2" Name="prod_add">
  <System Name="prod_add" Location="">
    <Block BlockType="Sum" Name="Add" SID="3" Ports="2,1"
Position="[330, 112, 360, 143]">
      <P Name="ZOrder" Value="3"/>
      <P Name="InputSameDT" Value="off"/>
      <P Name="OutDataTypeStr" Value="&quot;Inherit: Inherit via
internal rule&quot;"/>
      <P Name="SaturateOnIntegerOverflow" Value="off"/>
    </Block>
    <Block BlockType="Constant" Name="Constant" SID="8" Ports=""
Position="[65, 80, 95, 110]">
      <P Name="ZOrder" Value="8"/>
      <P Name="Value" Value="3"/>
    </Block>
    <Block BlockType="Constant" Name="Constant1" SID="9" Ports=""
Position="[65, 135, 95, 165]">
      <P Name="ZOrder" Value="9"/>
      <P Name="Value" Value="2"/>
    </Block>
    <Block BlockType="Constant" Name="Constant2" SID="10" Ports=""
Position="[170, 165, 200, 195]">
      <P Name="ZOrder" Value="10"/>
      <P Name="Value" Value="2"/>
    </Block>
    <Block BlockType="Product" Name="Product" SID="1" Ports="[2, 1]"
Position="[215, 87, 245, 118]">
      <P Name="ZOrder" Value="1"/>
      <P Name="InputSameDT" Value="off"/>
      <P Name="OutDataTypeStr" Value="&quot;Inherit: Inherit via
internal rule&quot;"/>
      <P Name="RndMeth" Value="Floor"/>
      <P Name="SaturateOnIntegerOverflow" Value="off"/>
    </Block>
    <Block BlockType="Outport" Name="D" SID="13" Position="[455, 123,
485, 137]">
      <P Name="ZOrder" Value="13"/>
      <P Name="IconDisplay" Value="Port number"/>
    </Block>
    <Line SrcBlock="Constant" SrcPort="1" Points="" DstBlock="Product"
DstPort="1">
      <P Name="ZOrder" Value="6"/>
    </Line>
    <Line SrcBlock="Product" SrcPort="1" Points="" DstBlock="Add"
DstPort="1">
      <P Name="ZOrder" Value="4"/>
    </Line>
    <Line SrcBlock="Constant1" SrcPort="1" Points="" DstBlock="Product"
DstPort="2">
      <P Name="ZOrder" Value="10"/>
    </Line>
    <Line SrcBlock="Constant2" SrcPort="1" Points="" DstBlock="Add"
DstPort="2">
      <P Name="ZOrder" Value="11"/>
```

```
    </Line>
    <Line SrcBlock="Add" SrcPort="1" Points="" DstBlock="D"
DstPort="1">
      <P Name="ZOrder" Value="12"/>
    </Line>
  </System>
</simulink2:Model>
```

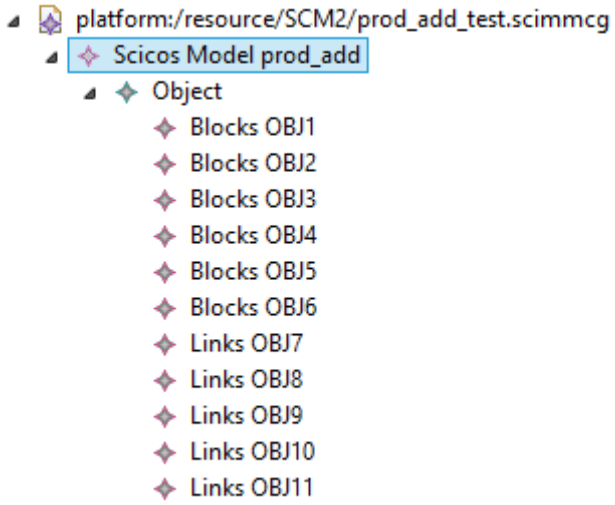

Figure 8.3: Intermediate model IM2

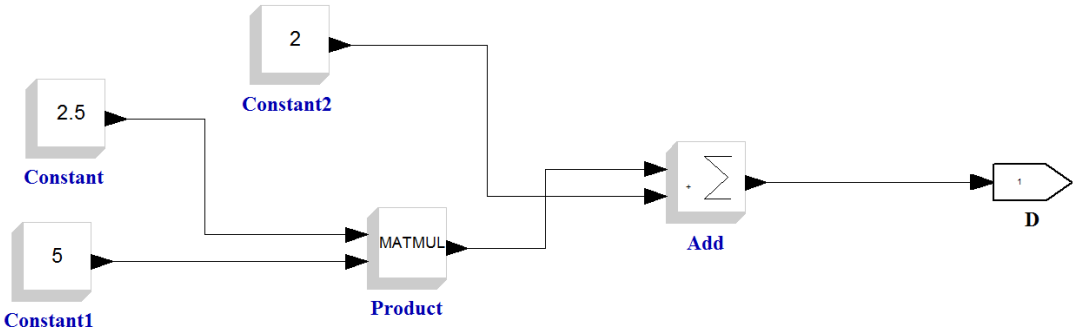

Figure 8.4: Resultant Scicos Model

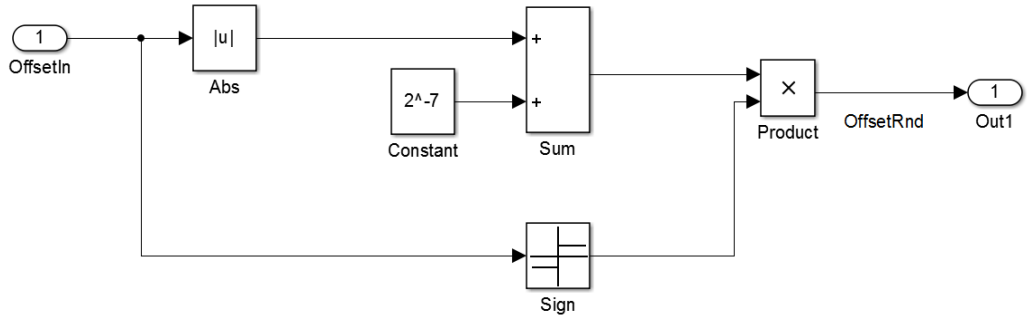

Figure 8.5: A subsystem of cross wind assistance system

Example 2: Simulink model of a subsystem of cross wind assistance system:

The model presented in figure 8.5 is a subsystem, part of cross wind assistance system, which is a driver assistance system. This model represents subsystem which is used to round off the offset added to the Electric Power Steering Angle. The input is the offset stored in memory. This offset value is obtained as a process of learning. The Steering Angle would be converted to integer during further processing. This introduces a round-off error. Hence, this subsystem is used to reduce the round off error by half its scale. The original offset before round off is applied as input to Simulink model. The output from the model represents the offset rounded off with error less than or equal to half the scale. Without a round off, the error would be upto one full scale. This value obtained is used for further processing in cross wind assistance system.

The transformation of the model is then carried out. First an intermediate model IMSubSystem1 is obtained. IMsubSystem1 is presented in figure 8.6. The transformation rule is then applied to the model IMSubSystem1. The resulting model is IMSubsystem2 and is presented in figure 8.7. Then the intermediate transformation is performed on IMSubSystem2 to obtain a Scicos model. The resultant Scicos model for the subsystem of cross wind assistance system is presented in figure 8.8.

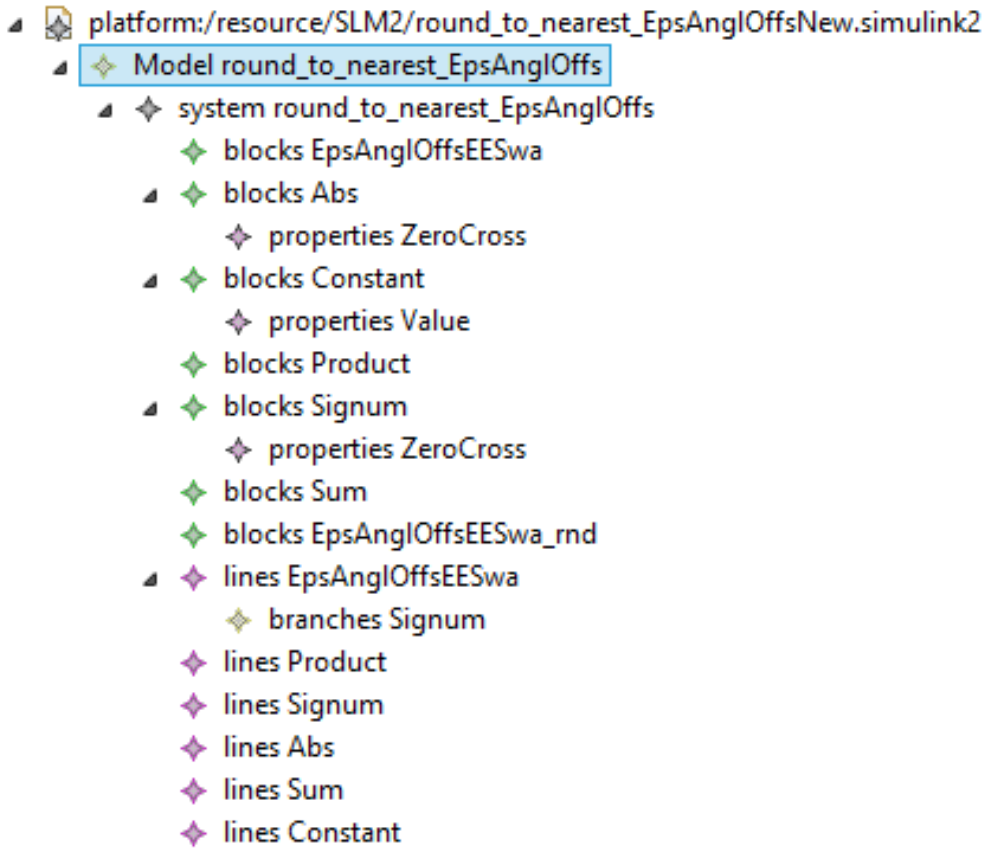

Figure 8.6: Intermediate model IMSubSystem1

The XML representation of the Intermediate model IMSubSystem1 is presented below:

```
<?xml version="1.0" encoding="UTF-8"?>
<simulink2:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:simulink2="platform:/resource/SLM2/model/Simulink.ecore"
Version="8.2" Name="round_to_nearest_EpsAnglOffs">
  <System Name="round_to_nearest_EpsAnglOffs" Location="[92, 69, 908,
608]">
    <Block BlockType="Inport" Name="EpsAnglOffsEESwa" SID="1" Ports=""
Position="[135, 113, 165, 127]" BlockMirror=""/>
    <Block BlockType="Abs" Name="Abs" SID="2" Ports="" Position="[365,
100, 405, 140]">
      <P Name="ZeroCross" Value="off"/>
    </Block>
    <Block BlockType="Constant" Name="Constant" SID="3" Position="[465,
141, 505, 179]">
      <P Name="Value" Value="2^-7"/>
    </Block>
    <Block BlockType="Product" Name="Product" SID="4" Ports="[2, 1]"
Position="[780, 130, 820, 170]"/>
```

```
    <Block BlockType="Signum" Name="Signum" SID="6" Position="[540,
230, 580, 270]">
      <P Name="ZeroCross"/>
    </Block>
    <Block BlockType="Sum" Name="Sum" SID="7" Ports="[2, 1]"
Position="[540, 102, 580, 178]"/>
    <Block BlockType="Outport" Name="EpsAnglOffsEESwa_rnd" SID="8"
Position="[1005, 143, 1035, 157]"/>
    <Line SrcBlock="EpsAnglOffsEESwa" SrcPort="1" Points="[160, 0]"
DstBlock="Abs" DstPort="1">
      <Branch DstBlock="Signum" DstPort="1"/>
    </Line>
    <Line SrcBlock="Product" SrcPort="1"
DstBlock="EpsAnglOffsEESwa_rnd" DstPort="1"/>
    <Line SrcBlock="Signum" SrcPort="1" Points="[120, 0; 0, -90]"
DstBlock="Product" DstPort="2"/>
    <Line SrcBlock="Abs" SrcPort="1" DstBlock="Sum" DstPort="1"/>
    <Line SrcBlock="Sum" SrcPort="1" DstBlock="Product" DstPort="1"/>
    <Line SrcBlock="Constant" SrcPort="1" DstBlock="Sum" DstPort="2"/>
  </System>
                          </simulink2:Model>
```

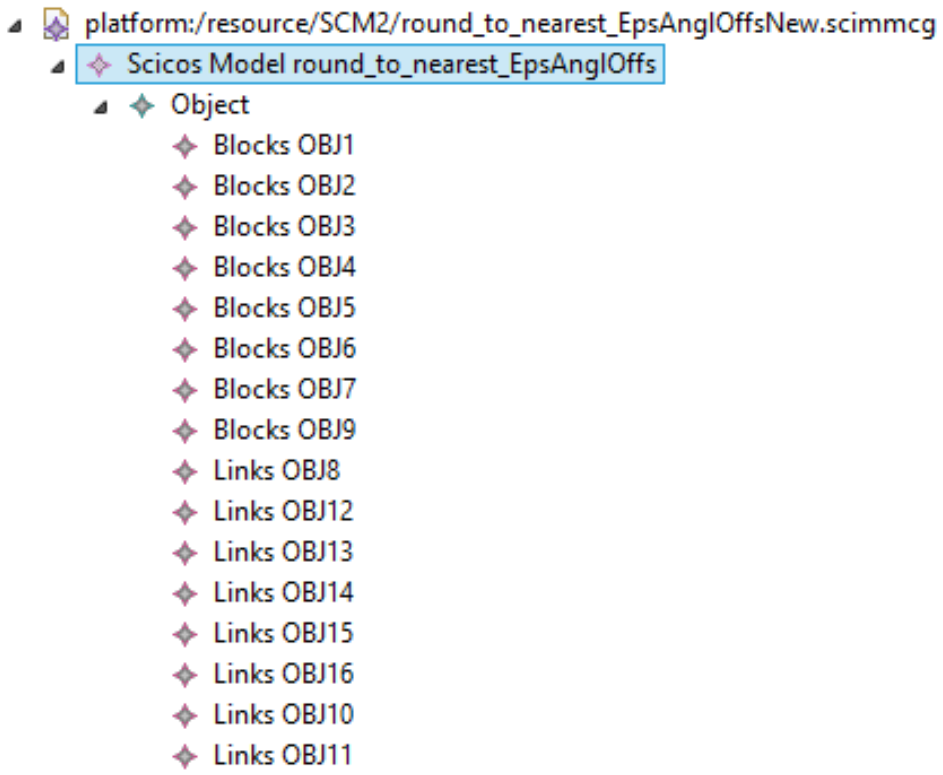

Figure 8.7: Intermediate model IMSubSystem2

The XML representation of the Intermediate model IMSubSystem2 is presented below:

```
<?xml version="1.0" encoding="UTF-8"?>
<scimmC:ScicosModel xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:scimmC="com.example.scimmC"
xsi:schemaLocation="com.example.scimmC model/Scicos.ecore"
Name="round_to_nearest_EpsAnglOffs">
  <Objects>
    <Block gui="IN_f" orig="[135,113]" sz="[30.0,14.0]"
exprs="[&amp;quot;1&amp;quot;;&amp;quot;-1&amp;quot;;&amp;quot;-
1&amp;quot;]" pout="8" gr_i="list(&amp;quot; &amp;quot;,8)"
out_implicit="&amp;quot;E&amp;quot;" id="OffsetIn" Name="OBJ1"/>
    <Block gui="ABS_VALUEi" orig="[365,100]" sz="[40.0,40.0]"
exprs="&amp;quot;0&amp;quot;" pin="11" pout="14"
gr_i="list([&amp;quot;txt=[&amp;apos;&amp;apos; |u|
&amp;apos;&amp;apos;];&amp;quot;;&amp;quot;xstringb(orig(1),orig(2),txt
,sz(1),sz(2),&amp;apos;&amp;apos;fill&amp;apos;&amp;apos;)&amp;quot;],8
)" in_implicit="&amp;quot;E&amp;quot;"
out_implicit="&amp;quot;E&amp;quot;" id="Abs" Name="OBJ2"/>
    <Block gui="CONST_m" orig="[465,141]" sz="[40.0,38.0]"
exprs="&amp;quot;2^-7&amp;quot;" pout="16"
gr_i="list([&amp;quot;dx=sz(1)/5;dy=sz(2)/10;&amp;quot;;&amp;quot;w=sz(
1)-2*dx;h=sz(2)-
2*dy;&amp;quot;;&amp;quot;txt=C;&amp;quot;;&amp;quot;xstringb(orig(1)+d
x,orig(2)+dy,txt,w,h,&amp;apos;&amp;apos;fill&amp;apos;&amp;apos;);&amp
;quot;],8)" out_implicit="&amp;quot;E&amp;quot;" id="Constant"
Name="OBJ3"/>
    <Block gui="MATMUL" orig="[780,130]" sz="[40.0,40.0]"
exprs="[&amp;quot;1&amp;quot;;&amp;quot;2&amp;quot;;&amp;quot;1&amp;quo
t;]" pin="13,15" pout="12"
gr_i="list(&amp;quot;xstringb(orig(1),orig(2),[&amp;apos;&amp;apos;MATM
UL&amp;apos;&amp;apos;],sz(1),sz(2),&amp;apos;&amp;apos;fill&amp;apos;&
amp;apos;);&amp;quot;,8)"
in_implicit="[&amp;quot;E&amp;quot;;&amp;quot;E&amp;quot;]"
out_implicit="[&amp;quot;E&amp;quot;]" id="Product" Name="OBJ4"/>
    <Block gui="SIGNUM" orig="[540,230]" sz="[40.0,40.0]"
exprs="&amp;quot;1&amp;quot;" pin="10" pout="13"
gr_i="list([&amp;quot;txt=[&amp;apos;&amp;apos;Sign&amp;apos;&amp;apos;
];&amp;quot;;&amp;quot;xstringb(orig(1),orig(2),txt,sz(1),sz(2),&amp;ap
os;&amp;apos;fill&amp;apos;&amp;apos;)&amp;quot;],8)"
in_implicit="&amp;quot;E&amp;quot;"
out_implicit="&amp;quot;E&amp;quot;" id="Signum" Name="OBJ5"/>
    <Block gui="SUMMATION" orig="[540,102]" sz="[40.0,76.0]"
exprs="[&amp;quot;1&amp;quot;;&amp;quot;[1,1]&amp;quot;;&amp;quot;0&amp
;quot;]" pin="14,16" pout="15"
gr_i="list([&amp;quot;[x,y,typ]=standard_inputs(o)
&amp;quot;;&amp;quot;dd=sz(1)/8,de=0,&amp;quot;;&amp;quot;if
~arg1.graphics.flip then dd=6*sz(1)/8,de=-
sz(1)/8,end&amp;quot;;&amp;quot;for
k=1:size(x,&amp;apos;&amp;apos;*&amp;apos;&amp;apos;)&amp;quot;;&amp;qu
ot;if size(sgn,1)&amp;gt;1 then&amp;quot;;&amp;quot;  if
sgn(k)&amp;gt;0 then&amp;quot;;&amp;quot;    xstring(orig(1)+dd,y(k)-
4,&amp;apos;&amp;apos;+&amp;apos;&amp;apos;)&amp;quot;;&amp;quot;
else&amp;quot;;&amp;quot;    xstring(orig(1)+dd,y(k)-
4,&amp;apos;&amp;apos;-&amp;apos;&amp;apos;)&amp;quot;;&amp;quot;
end&amp;quot;;&amp;quot;end&amp;quot;;&amp;quot;end&amp;quot;;&amp;quot
;xx=sz(1)*[.8 .4 0.75 .4
```

```
.8]+orig(1)+de&amp;quot;;&amp;quot;yy=sz(2)*[.8 .8 .5 .2
.2]+orig(2)&amp;quot;;&amp;quot;xpoly(xx,yy,&amp;apos;&amp;apos;lines&a
mp;apos;&amp;apos;)&amp;quot;],8)"
in_implicit="[&amp;quot;E&amp;quot;;&amp;quot;E&amp;quot;]"
out_implicit="[&amp;quot;E&amp;quot;]" id="Sum" Name="OBJ6"/>
    <Block gui="OUT_f" orig="[1005,143]" sz="[30.0,14.0]"
exprs="&amp;quot;1&amp;quot;" pin="12" gr_i="list(&amp;quot;
&amp;quot;,8)" in_implicit="&amp;quot;E&amp;quot;" id="OffsetRnd"
Name="OBJ7"/>
    <Block gui="SPLIT_f" orig="[235.0,113]"
sz="[0.33333333333333331,0.33333333333333331]" exprs="[]" pin="8"
pout="10;11" gr_i="list([],8)" in_implicit="&amp;quot;E&amp;quot;"
out_implicit="[&amp;quot;E&amp;quot;;&amp;quot;E&amp;quot;]"
Name="OBJ9"/>
    <Link
xx="[165.0;260.7142857142857;260.7142857142857;356.42857142857144]"
yy="[120.0;120.0;120.0;120.0]" from="[1,1,0]" to="[9,1,0]"
Name="OBJ8"/>
    <Link
xx="[820.0;908.2142857142858;908.2142857142858;996.4285714285714]"
yy="[150.0;150.0;150.0;150.0]" from="[4,1,0]" to="[7,1,0]"
Name="OBJ12"/>
    <Link
xx="[580.0;675.7142857142858;675.7142857142858;771.4285714285714]"
yy="[250.0;250.0;143.33333333333334;143.33333333333334]" from="[5,1,0]"
to="[4,2,0]" Name="OBJ13"/>
    <Link
xx="[405.0;468.2142857142857;468.2142857142857;531.4285714285714]"
yy="[120.0;120.0;152.66666666666666;152.66666666666666]" from="[2,1,0]"
to="[6,1,0]" Name="OBJ14"/>
    <Link
xx="[580.0;675.7142857142858;675.7142857142858;771.4285714285714]"
yy="[140.0;140.0;156.66666666666666;156.66666666666666]" from="[6,1,0]"
to="[4,1,0]" Name="OBJ15"/>
    <Link
xx="[505.0;518.2142857142858;518.2142857142858;531.4285714285714]"
yy="[160.0;160.0;127.33333333333334;127.33333333333334]" from="[3,1,0]"
to="[6,2,0]" Name="OBJ16"/>
    <Link
xx="[235.0;383.2142857142857;383.2142857142857;531.4285714285714]"
yy="[113.0;113.0;250.0;250.0]" id="b2l_1_10" from="[9,1,0]"
to="[5,1,0]" Name="OBJ10"/>
    <Link
xx="[235.0;295.7142857142857;295.7142857142857;356.42857142857144]"
yy="[113.0;113.0;120.0;120.0]" id="b2l_2_11" from="[9,2,0]"
to="[2,1,0]" Name="OBJ11"/>
  </Objects>
</scimmC:ScicosModel>
```

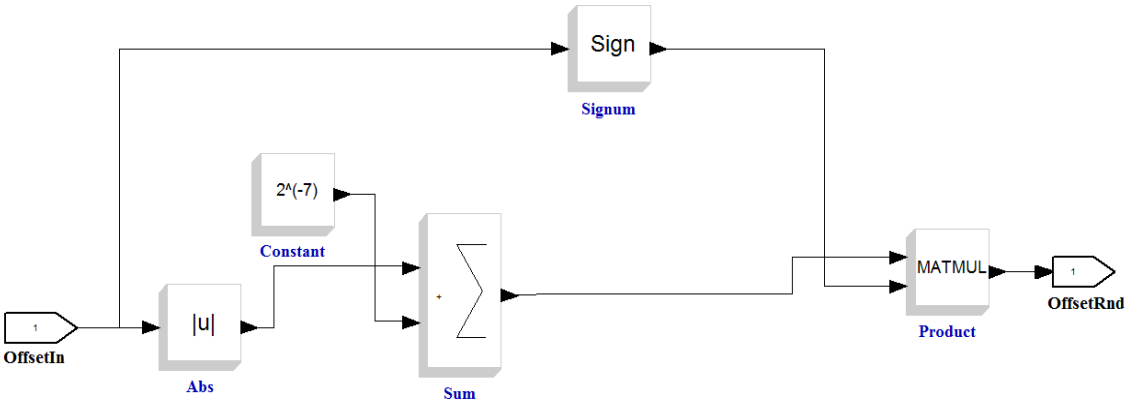

Figure 8.8: Resultant Scicos model for the subsystem of cross wind assistance system

8.2 Results of Reverse Transformation

The aim of the project was to develop metamodels for Scicos and Simulink and also to develop forward transformation between Simulink and Scicos. However, an attempt has been made to achieve reverse transformation also.

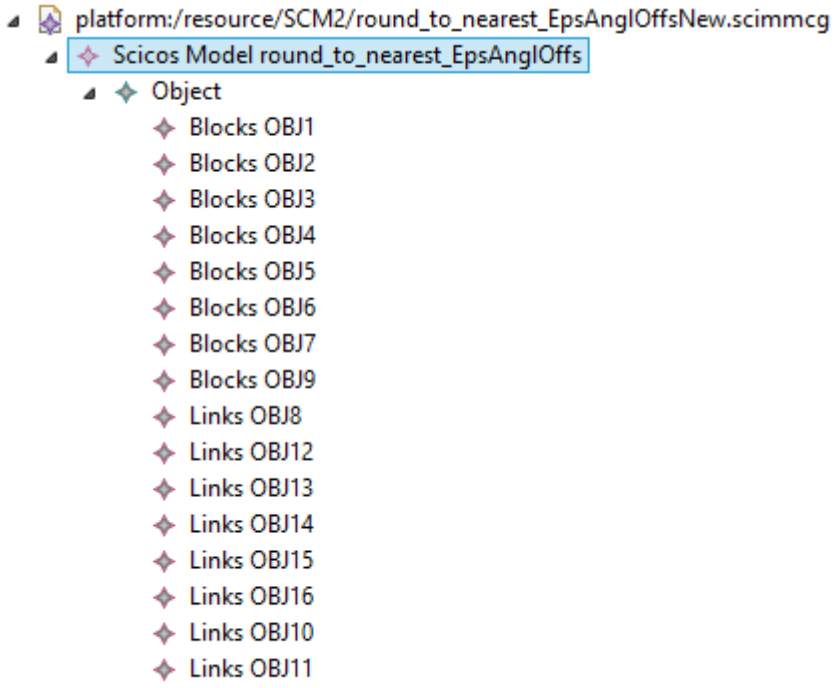

Figure 8.9: Intermediate model IMR1

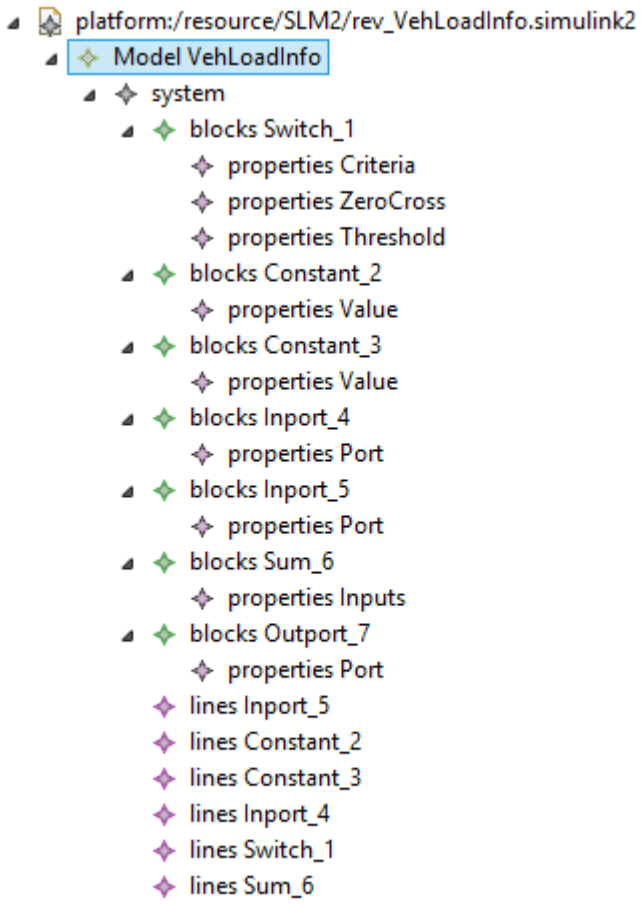

Figure 8.10: Intermediate model IMR2

The results of the reverse transformation from Scicos model to Simulink model is discussed as follows.

The reverse transformation is developed using QVTo for the intermediate models. The intermediate model IMR1 is presented in figure 8.9. It is a model of a subsystem VehicleLoadInfo of the Cross Wind Assistance System. The model IMR1 is provided as the input to the transformation definition. The output of the transformation is the intermediate model IMR2 which conforms to the Simulink metamodel and it is presented in figure 8.10. The model IMR2 was verified by using the XML file of the model. It represented all the characteristics of the Simulink model that was used for developing the model IMR1.

8.3 Verification and Validation

Verification of a model is essential for ensuring the correct functioning of a model. Here, verification and validation has been carried out as a process wherein the models are executed to ensure that the desired behaviour is produced by them.

The Simulink model presented in figure 8.2 is considered for validation. It is given a set of values as inputs and certain output obtained. This model is transformed into a Scicos model as presented in figure 8.4. The same inputs that were given to the Simulink model, are provided to the resultant Scicos model. The output obtained by Simulink model is compared with the output obtained from Scicos model. It is observed that the outputs of both the models match, thus verifying the transformation.

Another example, a subsystem of a cross wind assistance system as presented in the figure 8.5 is also subjected to transformation and the resultant Scicos model is presented in figure 8.6. Here again, the outputs obtained by both the models are compared. The outputs of both the models match. Hence, the transformation is verified.

In the above figure, blocks of constants, product and sum were used. Constants values, *Constant*, *Constant1* and *Constant2* are given various values. The intermediate results and the final results are first recorded for the Simulink model itself. After verifying the results, the same inputs are applied to the Scicos model.

Input Parameters	Constant	Constant1	Constant2	Output of Product block	Sum(Final Output D)
Test 1	2.5	2	2	5	7
Test 2	1	2	3	2	5
Test 3	-0.7	17	3	-11.9	-8.9
Test 4	-0.9	25	-5	-22.5	-27.5

Table 8.1: Range of Input values and corresponding output values for Example 1

Input Parameters	EPSAngOFFsEESwa	|u|	Signum	Sum	Product (Final Output at Port D)
Test 1	0	0	0	0.0078	0
Test 2	0.5	0.5	1	0.5078	0.5078
Test 3	-0.5	0.5	-1	0.5078	-0.5078
Test 4	1	1	1	1.0078	1.0078
Test 5	-1	1	-1	1.0078	-1.0078
Test 6	-0. 0312	0. 0312	-1	0.039012	-0. 039012
Test 7	0.0312	0.0312	1	0.039012	0. 039012
Test 8	0.0156	0.0156	1	0.023412	0.023412
Test 9	-0.0156	0.0156	-1	0.023412	-0.023412

Table 8.2: Range of Input values and corresponding output values for Example 2

A range of values were provided as inputs for example 1. Example 1 corresponds to figure 8.1 and figure 8.4. The input and output values are presented in table 8.1. The same is followed for example 2, which corresponds to figure 8.5 and figure 8.8. The input and corresponding output values for the same are presented in the table 8.2. For different ranges of inputs, it is observed that the resultant Scicos model provides the same output as that of Simulink model.

The various input parameters Constant, Constant1, Constant2 have been provided. The output of the product block and the sum block are checked. The final output at the output port D is also verified for different test values and found to be same as that for Simulink model.

Similarly, input parameter OffsetIn is varied and the output at the OffsetRnd is recorded. This input-output combination provided to the Scicos model is compared with the results of the Simulink model and is found to be same. Hence, validating the transformation.

8.4 Summary

The results of both the forward and the reverse transformations are discussed in this chapter. A simple Simulink model was first developed and its forward transformation was verified. Then a Simulink subsystem of a cross wind assistance system was subjected to forward transformation. The outputs of the model before transformation and that after transformation were compared. They were found to give the same output as desired. Thus, verifying and validating the transformation definition.

An attempt was made to achieve reverse transformation. A subsystem in Scicos was considered for reverse transformation. After subjecting it to transformation, an intermediate model conforming to Simulink metamodel was obtained. This verifies the reverse transformation. Further, intermediate transformation for the reverse transformation can be taken up for future work.

Chapter 9
Conclusion and Future Work

Model transformation has been carried out for the interoperability between Simulink model and Scicos model. To bring about model transformation, the following has been achieved in the present work:

1. Development of metamodel for Simulink
2. Development of metamodel for Scicos
3. Development of Forward Transformation Definition, including the intermediate transformations

The metamodels for Simulink and Scicos are developed based on OMGs MOF Standards, using the EMF Ecore's Graphical Modeling Tool. As a part of the solution for the interoperability, a forward Transformation Definition from Simulink to Scicos has been developed for a subset of the Simulink blocks. The development of the transformation definition is carried out using QVT Operational Mappings (QVTo). The forward transformation has been applied to a subsystem of cross wind assistance system.

Further, an attempt has been made to bring about a reverse transformation from Scicos to Simulink. The transformation definition developed transforms an intermediate model which conforms to the Scicos metamodel into another intermediate model which conforms to the Simulink metamodel. This can further be carried forward to develop an intermediate transformations for reverse transformation.

Further, enhancement include the following:

1. Extension of the transformation to more blocks
2. Implementation of intermediate transformation for Reverse transformation
3. Improvement of the intermediate transformation for forward transformation
4. Improvement of the Simulink metamodel for reverse transformation

References

[1] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic, "Translating discrete-time simulink to lustre". ACM Trans. Embed. Comput. Syst. 4, 4, November 2005, pp. 779-818

[2] http://nicolas.thiery.name/CalculFormelLibre/Scilab.pdf as on 16 December 2015

[3] Nah Soo Hoe, “Free/Open Source Software Open Standards”, Asia-Pacific Development Information Programme, 2006

[4] http://perens.com/OpenStandards/Definition.html as on 16 December 2015

[5] A. Sindico, M. Di Natale, and G. Panci, “Integrating SysML with Simulink using Open-source Model Transformations”. SIMULTECH, SciTePress, 2011, pp. 45-56

[6] http://www.omg.org/spec/QVT/ last accessed on 16 December 2015

[7] I. Arrassen, R. Esbai, A. Meziane and M. Erramdani, "QVT transformation by modeling: From UML model to MD model," 6th International Conference on, Sousse Sciences of Electronics, Technologies of Information and Telecommunications, 2012, pp. 86-91

[8] L. Zhang, M. Glab, N. Ballmann and J. Teich, "Bridging algorithm and ESL design: Matlab/Simulink model transformation and validation," Forum on Specification & Design Languages, Paris, France, 2013, pp. 1-8

[9] P. Guo, Y. Li, P. Li, S. Liu and D. Sun, "A UML Model to Simulink Model Transformation Method in the Design of Embedded Software," Tenth

International Conference on Computational Intelligence and Security, Kunming, 2014, pp. 583-587

[10] Dae-Kyoo Kim *et al.*, "QVT-Based Model Transformation to Support Unification of IEC 61850 and IEC 61970," in IEEE Transactions on Power Delivery, April 2014, vol. 29, no. 2, pp. 598-606

[11] D. Meedeniya, J. Bowles and I. Perera, "SD2CPN: A model transformation tool for software design models," International Computer Science and Engineering Conference, Khon Kaen, 2014, pp. 354-359

[12] A. B. Younes, Y. B. Hlaoui and L. J. B. Ayed, "A Meta-model Transformation from UML Activity Diagrams to Event-B Models," IEEE 38th International Computer Software and Applications Conference Workshops, Vasteras, 2014, pp. 740-745

[13] A. Achouri and L. J. Ben Ayed, "UML activity diagram to event-B: A model transformation approach based on the institution theory," IEEE 15th International Conference on Information Reuse and Integration, Redwood City, CA, 2014, pp. 823-829

[14] http://www.theenterprisearchitect.eu/blog/2009/02/18/model-driven-engineering-tools-compared-on-user-activities/ as on 20 October 2015

[15] http://www.omg.org/mda/ as on 24 December 2015

[16] M. Snoeck, "Enterprise Information Systems Engineering: The MERODE Approach", Springer, 2014

[17] A. G. Kleppe, W. Bast and J. B. Warmer, "MDA Explained: The Model Driven Architecture : Practice and Promise", Boston: Addison-Wesley, 2003

[18] T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation", Electronic Notes in Theoretical Computer Science, March 2006, 152 (1-2), pp.125-142

[19] S. L. Campbell, J. P. Chancelier and R. Nikoukhah, "Modeling and Simulation in Scilab/Scicos", Springer 2006

[20] http://in.mathworks.com/products/matlab/ as on 20 December 2015

[21] http://in.mathworks.com/products/simulink/ as on 20 December 2015

[22] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Feb 2015, ver1.2

[23] P. J. Barendrecht, "Modeling transformations using QVT Operational Mappings", April 2010

[24] R. C. Gronback, "ECLIPSE MODELING PROJECT", Addison-Wesley 2009

[25] http://www.esterel-technologies.com/products/scade-suite/ as on 20 December 2015

[26] http://www.softwaretestinghelp.com/what-is-stlc-v-model/ as on 20 December 2015

[27] OMG Specification Document, “Object Constraint Language”, Feb 2014, ver 2.4