REQUIREMENTS MODELS TRANSFORMATION:
FROM BUSINESS PROCESS MODELS TO
OBJECT STATECHARTS

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2014

By
Abdulrahman Abdallah Hussein
School of Computer Science
# Table of Contents

List of Figures .................................................................................................................. 5  
List of Tables .................................................................................................................... 6  
List of Abbreviations ........................................................................................................ 7  
Abstract ............................................................................................................................. 8  
Declaration .......................................................................................................................... 9  
Copyright ............................................................................................................................ 10  
Acknowledgement ............................................................................................................. 11  
1 Introduction ...................................................................................................................... 12  
  1.1 Motivation .................................................................................................................. 12  
  1.2 Aim and Objectives ................................................................................................... 14  
  1.3 Report Structure ....................................................................................................... 15  
2 Background ...................................................................................................................... 16  
  2.1 Requirements Engineering ....................................................................................... 16  
  2.2 PLANT .................................................................................................................... 18  
  2.3 Business Process Model and Notation ................................................................. 20  
  2.4 UML State Machine ............................................................................................... 22  
  2.5 Model-Driven Architecture .................................................................................... 25  
  2.6 Models Transformation ......................................................................................... 27  
  2.7 Related Work .......................................................................................................... 29  
3 Overall Design ................................................................................................................. 31  
  3.1 Assumptions ............................................................................................................. 31  
  3.2 System Architecture ............................................................................................... 32  
    3.2.1 Filtering Component ....................................................................................... 33  
    3.2.2 Synthesising Component ............................................................................... 33  
  3.3 System Modules ....................................................................................................... 33  
    3.3.1 Input validation ............................................................................................... 34  
    3.3.2 Preprocessing ................................................................................................. 34  
    3.3.3 Filtering ......................................................................................................... 34  
    3.3.4 Petri net mapping ........................................................................................... 35  
    3.3.5 Petri net to statechart hierarchy ..................................................................... 35  
    3.3.6 UML2 state machine mapping ....................................................................... 36  
3.4 Metamodels .................................................................................................................. 36
List of Figures

Figure 1-1: Report roadmap .......................................................................................................................... 15
Figure 2-1: Categories of requirement models [10] ....................................................................................... 17
Figure 2-2: PLANT pattern language and its multi-perspective requirement models [2] ......................... 19
Figure 2-3: BPMN2 Flow Elements, from Eclipse BPMN2.0 Ecore metamodel ........................................ 21
Figure 2-4: BPMN 2.0 data elements, from Eclipse BPMN2.0 Ecore metamodel .................................... 23
Figure 2-5: UML2 state machine, from Eclipse UML2.4 Ecore metamodel ................................................ 24
Figure 2-6: MDA levels of abstraction [15] .................................................................................................. 25
Figure 2-7: MOF metamodel levels [15] .................................................................................................... 26
Figure 2-8: Model transformations [15] ...................................................................................................... 27
Figure 3-1: System architecture and integration of modules ...................................................................... 32
Figure 3-2: System transformation workflow, represented as a BPMN process ...................................... 35
Figure 3-3: Normalised BPMN metamodel, where object nodes are also flow nodes ............................ 37
Figure 3-4: Petri Net and Statechart Ecore metamodels [27] ................................................................... 37
Figure 4-1: Preprocessing (normalisation) rules, based on [5] ................................................................. 41
Figure 4-2: Filtering rules [5] .................................................................................................................... 42
Figure 4-3: Filtered process model to Petri net mapping rules ............................................................... 44
Figure 4-4: AND/OR reduction rules defined in the PN2SC algorithm [7] ............................................ 47
Figure 4-5: UML state machine mapping rules .......................................................................................... 49
Figure 5-1: The Eclipse Modelling Project [32] ....................................................................................... 51
Figure 5-2: Epsilon platform architecture[35] .......................................................................................... 53
Figure 6-1: System flowchart .................................................................................................................... 58
Figure 6-2: Tax process model used as an example for the system input .............................................. 59
Figure 6-3: An example of two validation rules as part of the EVL validation module ...................... 60
Figure 6-4: Migration rule for DataInputAssociation as part of the preprocessing flock module ....... 61
Figure 6-5: Tax process after the preprocessing (normalisation) step ...................................................... 62
Figure 6-6: Filtering Rule R1 as part of the filtering EPL module .......................................................... 63
Figure 6-7: Tax process after applying the filtering rules ..................................................................... 64
Figure 6-8: Tax petri net equivalent to the tax filtered process model .................................................. 64
Figure 6-9: Function to store the (original filtered, migrated petri net) map for support of traceability .................................................. 65
Figure 6-10: Hierarchical statechart equivalent to the petri net in figure 6-8; generated after applying the reduction rules .................................................................................................................. 66
Figure 6-11: Part of M3 mapping rule implementation ............................................................................ 67
Figure 6-12: Final UML state machine output generated for the tax process model .......................... 67
Figure 7-1: EUnit test for R2 rule in the filtering module ..................................................................... 69
Figure 7-2: Test case T6 which contains cross-synchronisation that is removed by the filtering rules .................................................. 71
Figure 7-3: Test case T7, where cross-synchronisation is not removed by the filtering rules .... 72
Figure 7-4: Test case T7 output before applying the cross-synchronisation extension rules .......................... 72
Figure 7-5: Test case T7 output after applying the cross-synchronisation extension rules, without refactoring .................................................................................................................. 73
List of Tables

Table 2-1: Classification of visual requirement models [12]……………………………………… 18
Table 2-2: Model transformations classification, data taken from [20]………………………… 28
Table 2-3: Comparison of related work …………………………………………………………… 29
Table 3-1: Summary of transformation phases ………………………………………………….. 38
Table 4-1: Summary of preprocessing rules …………………………………………………… 42
Table 4-2: Summary of filtering rules [5] ……………………………………………………… 43
Table 4-3: Summary of petri net mapping rules ……………………………………………… 44
Table 4-4: Summary of the UML state machine mapping rules …………………………… 50
Table 5-1: List of Eclipse/Epsilon tools used in the project ………………………………… 57
Table 5-2: Epsilon languages used for each system module ……………………………… 57
Table 7-1: Test cases applied for the system and their matching with the expected result … 70
Table 7-2: Classification of petri nets and their translatability to statecharts, based on [7] … 74
### List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD2SC</td>
<td>UML Activity Diagram to UML Statechart</td>
</tr>
<tr>
<td>AOM</td>
<td>Agent-Oriented Model</td>
</tr>
<tr>
<td>ATL</td>
<td>Atlas Transformation Language</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Model and Notation</td>
</tr>
<tr>
<td>BP2SC</td>
<td>Business Process to UML Statechart</td>
</tr>
<tr>
<td>CIM</td>
<td>Computation Independent Model</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Framework</td>
</tr>
<tr>
<td>EOL</td>
<td>Epsilon Object Language</td>
</tr>
<tr>
<td>EPL</td>
<td>Epsilon Pattern Language</td>
</tr>
<tr>
<td>EVL</td>
<td>Epsilon Validation Language</td>
</tr>
<tr>
<td>GOM</td>
<td>Goal-Oriented Model</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-Driven Architecture</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-Driven Development</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-Driven Engineering</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta-Object Facility</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OTM</td>
<td>Object Transformation Model</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PN2SC</td>
<td>Petri Net to (Hierarchical) Statechart</td>
</tr>
<tr>
<td>POM</td>
<td>Process-Oriented Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>RML</td>
<td>Requirement Modelling Language</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
Abstract

The context of this project is set by a recent study that presented a pattern language, named PLANT, to create a set of multi-perspective requirement models from scenarios. A process model and an object life cycle model are two such viewpoints. Business Process Model and Notation (BPMN) is the de-facto process modelling standard used to represent business processes. UML state machine (statechart) is used to describe objects behaviour. As per PLANT, creating the set of requirements models manually seemed to take time and effort. This project looked into using model transformations to create a UML statechart automatically from a BPMN process model with explicit data objects.

The BPMN to UML statechart transformation approach used here is based on a proved petri net to statechart translation algorithm and its use in a transformation chain defined for UML activity diagrams. The algorithm is described as “structure and behaviour-preserving”; meaning that the generated statechart has similar structure (one-to-one mapping) and behaviour to the input petri net. The petri net represents the filtered input process model; including only object and relevant control nodes. Based on the used approach, the transformation is successful for a subclass of BPMN models; not all input process models could be translated into statecharts. In brief, this is due to the semantics and expressivity differences between the source and target models, while taking “structure-preserving” in consideration. The BPMN input model covered here has similar data behaviour to UML activity diagrams as it appeared that BPMN can have different data semantics which needs further investigation. The system was tested using a number of test cases. Epsilon platform was used for the implementation. Specifically, Epsilon Pattern Language (EPL) was used for in-place transformations and Epsilon flock for model migrations. Automatically creating the target statechart model can help minimize the effort needed to create the set of requirement models.
Declaration

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this dissertation (including any appendices and/or schedules to this dissertation) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this dissertation, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has entered into. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the dissertation, for example graphs and tables (“Reproductions”), which may be described in this dissertation, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this dissertation, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Police (see http://documents.manchester.ac.uk/display.aspx?DocID=487), in any relevant Dissertation restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s Guidance for the Presentation of Dissertations.
Acknowledgement

I would like to thank my project supervisor, Dr Liping Zhao, for giving me the opportunity to delve into the model-driven development world. Her continuous support and guidance throughout the project and her valuable and positive feedback helped me to overcome all the difficulties.

I am indebted to Professor Rik Eshuis and Professor Pieter Van Gorp, at The Eindhoven University of Technology, for sparing from their valuable time to answer my queries regarding their research work used here. Their invaluable explanations helped me to understand better the transformation approach used for the system.

I am also grateful to Dr Dimitris Kolovos and Dr Louis Rose, at The University of York, for their contributions towards my Epsilon implementation. I would like to thank Dr Louis for reusing his PN2SC Epsilon implementation. I would also like to thank Dr Dimitris for his prompt and helpful answers to all my posted questions on the Epsilon online forum.

Last but not least, I would like to thank my family and all my friends around me for their continual encouragement and support. Without all these people, this thesis would have never been a reality.
1 Introduction

1.1 Motivation

In the requirements acquisition and analysis phase of software development, system requirements are often captured as scenarios that are described within a use case. A scenario describes the interactions between the system and actors in order to achieve a goal for some stakeholders [1]. Scenarios are often written as simple textual narratives; hence, it is considered convenient to capture user’s knowledge. They are then transformed into different kinds of models toward the system design and implementation. Despite the fact that scenarios have an important role in requirements engineering, their informal textual nature can impose some drawbacks. Natural language is inherently flexible, expressive, and easy to use, yet it can be ambiguous and difficult to understand. Therefore, scenarios can be inaccurate and they may have implicitly hidden goals [2]. Nevertheless, it is believed that they will still be widely used to specify requirements [3].

In order to address their drawbacks, a recent study [2] proposed a pattern language, called PLANT, for transforming scenarios into different requirements models. The models provide a way to capture system requirements from different perspectives using a group of models each of which focuses on a specific view of the system. In the aforementioned paper, a scenario metamodel was created with four major aspects: process, object, agent, and goal. Accordingly, four different patterns were used to convert each aspect into a corresponding target model. The process aspect is transformed into a “Process-Oriented Model” (POM) which represents the sequence of actions undertaken in the path towards the goal of the scenario. The object aspect is transformed into an “Object Transformation Model” (OTM) which represents the state transitions of an object that is caused by the process actions. The agent aspect is transformed into an “Agent-Oriented Model” (AOM) that shows agents communication in the scenario. Finally, the goal aspect is transformed into a “Goal-Oriented Model” (GOM) that represents the scenario goal and its sub-goals.

Based on a usability study, the approach (PLANT) showed its usefulness and satisfaction. However, manually creating the different requirements models from
scenarios seemed to be a challenge and time-consuming effort. In order to increase the efficiency, the study concluded with a suggestion to seek the automation in some of the manual modelling processes [2]. The motivation of this project is to help in minimizing this effort by automatically generating models using model transformation technologies. This project looked into translating POM, specifically, Business Process Modelling Notation (BPMN) models, into OTM, specifically, UML state machine (statechart) models.

A POM model is a process flow model usually represented in BPMN while an OTM is a data flow model often represented as a UML statechart. BPMN process models represent enterprise business activities (‘As Is’ or ‘To Be’). As stated by OMG, the main aim of BPMN is to provide a simple and understandable representation which facilitates communication between business analysts and clients [4]. UML state machines are also used in the early stages of software development to help in understanding the behaviour and lifecycle of an object. While the object state transitions are often implicitly included in the process model, explicitly extracting it into an object state model provides a useful complementary data model.

To explain the idea behind the project, the tax collection process [5] [6] is used as a running example. The process represents communication between three parties: client, tax advisor, and municipality. The declaration document object is used instead of the messages exchanged between the parties. Briefly, the scenario is summarized as follows:

*The client submits an annual statement document to the tax advisor who checks it upon receiving and either accepts the document or rejects it and sends the rejected document back to the client. If the document is accepted, the tax advisor notifies the client and sends the processed document to the municipality that calculates the tax and sends back the assessed document. The client finally receives the assessed tax document.*

For example, a business analyst creates a BPMN process model from the above given scenario to show the actions and tasks performed. The process model also shows the data (declaration document) needs and results of the tasks. This data enriched process model can be used to automatically generate a UML statechart that shows the possible states and transitions for the data object. The BPMN and UML statechart models are shown later in chapter 6 as one of the test cases.
1.2 Aim and Objectives

The aim of this project is to generate automatically a UML statechart from a given BPMN process model. For this project, the initial process model is assumed to have been already created for a given scenario and represented using BPMN. This process model, conforming to BPMN (2.0) metamodel, will act as the input source model for the transformation system, and the output target model will be an object lifecycle model represented as a UML state machine model, which conforms to UML (2.4) metamodel.

The project attempted to answer the question: is it theoretically feasible to automatically translate a BPMN model with explicit data objects into a UML statechart, and how easily could it be implemented using the up-to-date modelling tools? In order to achieve this, the project involved the following objectives:

- Study the relative modelling standards and specifications, as well as the metamodels for the source and target models.
- Research the available literature for relative work to investigate the feasibility of the transformation.
- Examine current available model transformation technologies and their classification. Features available in modelling frameworks and tools such as the Eclipse Modelling Project (EMP) were considered.
- Design, implement, and test the transformation system.

The transformation can be straightforward if there is a direct one-to-one mapping between source and target model elements. However, the key challenge for this project is to identify transformation rules between two models that do not have this direct mapping. In fact, following the translation algorithm defined in [7], it turned out that not every process model is translatable into a UML statechart, although a subclass of BPMN models can be translated.

The deliverables of this project are as follows:

- The transformation program developed in Eclipse using the Epsilon platform.
- The test cases used for development and testing (input and output models in XMI format).
- The final dissertation report.
1.3 Report Structure

This report, in addition to this introductory chapter, has the following structure:

Chapter 2: Background
This chapter gives a review about related literature and background research. It includes requirement models, PLANT pattern language, BPMN and UML state machine models. Model-Driven Architecture (MDA) standards, and model transformations and classifications are also covered. The last section presents some related work.

Chapter 3: Overall Design
This chapter shows the system architecture, system modules, and the metamodels used in the system.

Chapter 4: Transformation and Mapping Rules
This chapter delves into the theory behind the transformation. The translation and mapping rules are explained in sufficient detail.

Chapter 5: Technology Used for Implementation
This chapter shows the selected technology for implementation based on Eclipse Modelling Project and Epsilon, and the rationale behind the selection.

Chapter 6: Implementation
This chapter demonstrates the implementation of the system with a running example and some code snippets.

Chapter 7: Testing and Evaluation
This chapter shows the unit tests and test cases used, followed by an evaluation of the obtained results.

Chapter 8: Conclusion and Future Work
This chapter gives a brief summary and presents some future work for the project.
2 Background

Modelling is considered an essential aspect in current software development methodologies. A model is an abstraction of the system that represents the system from a specific view or perspective. Providing different models helps in the communication between developers and stakeholders; hence reaching a better understanding of the business needs [3]. In Model-Driven Development (MDD) approach, models are considered the primary artefacts. Models are raised to an abstraction level above the programming language and implementation level, making it easier to use business domain concepts. The promises of MDD are in the flexibility of the implementation platform, and the improved productivity and quality of software; achieved by relying on automatic model transformations and generation of source code [8] [9].

2.1 Requirements Engineering

Requirements engineering refers to the software engineering field concerned with the processes and activities of deriving and managing software requirements. These activities vary depending on the environment and domain of the system, however, it commonly includes requirements elicitation, analysis, specification, validation, and management [10] [3].

Software requirements can be represented in different ways from informal narrative using natural languages to formal specification using formal mathematical languages. Formal specification is recommended for real-time critical systems. However, for many other systems, a well-structured textual document supplemented with graphical models and diagrams is commonly used [11].

Software requirements and models can be considered from different angles as shown in Figure 2-1. One effective way to capture user needs for a system is by taking a user-centric approach. A user-centric approach, in contrast to a product-centric way, focuses on what the users do to achieve their goals instead of what the system should do. Common user-centric approaches used in software requirements are use cases and user stories [11]. Use cases are based on identifying the system actors and their goals. An actor is a user role or an external system that communicates with the system. A single use case describes the interaction between the primary actor and the system to achieve
his goal [1]. Cockburn considers a use case as a collection of scenarios some of which are successful scenarios leading to the goal while others are failure scenarios. In his vision, a scenario may include a precondition, the steps taken in the normal flow and alternate flows, and a postcondition. Therefore, scenarios are usually represented in process flow models. In fact, scenarios are regarded as the first form of requirements developed; therefore, they derive the creation of other models [10].

Besides scenarios, different graphical models are also used to represent system requirements. In software models, there is a debate about to what extent the proverb “A picture is worth a thousand words” is true. However, many agree that there is “no single view of the requirements that gives a complete understanding” [11]. Therefore, the objective of requirements modelling is to create different viewpoints of the system at different levels of abstractions. This not only helps in understanding the system, but also in discovering incorrect, inconsistent, or missing requirements [11].

A set of models, called Requirement Modelling Languages (RML), has been developed to represent requirements visually in an easy way for both business and technical users [12]. Compared to UML, the latter is created mainly to model system design, and it is considered more complex for business users. The models in RML are classified into four categories as shown in Table 2-1: Objectives, People, Systems, and Data. As a best practice, all four categories are required to make a comprehensive picture of the solution [12].
Table 2-1: Classification of visual requirement models [12]

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Example of Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objectives</td>
<td>Describe the business value of the system and the priority of the requirements.</td>
<td>Business Objective Model. Feature Tree.</td>
</tr>
<tr>
<td>People</td>
<td>Describe the users of the system and their processes and goals.</td>
<td>Process Flow. Use Case.</td>
</tr>
<tr>
<td>System</td>
<td>Describe the existing systems, their interaction and behaviour, and the user interface.</td>
<td>Ecosystem Map. System Flow.</td>
</tr>
</tbody>
</table>

2.2 PLANT

Section 1.1 briefly mentioned PLANT [2] as a pattern language for transforming scenarios into multi-perspective models; where scenarios are considered as informal narratives. Therefore, they suffer from the limitations of natural language addressed earlier such as ambiguity and difficulty of understanding. The objective is that the generated requirement models, alongside with the scenario, can provide a more comprehensive and accurate view of the system requirements.

The paper defined a scenario metamodel based on the scenario concepts from “cognitive science” and requirement engineering viewpoints. A scenario is described to have an initial state, a final state, and a path that connects to the scenario’s goal. The path is a sequence of actions that involves one or many agents and can affect some object state. The scenario can be a normal scenario where the goal is achieved or an abnormal scenario where the goal is avoided.

There are four major aspects revealed by the scenario metamodel. These aspects are the action path, the object state transition, the agent interaction, and the goal. The PLANT pattern language consists of four patterns that produce a corresponding model for each of these aspects. The patterns and the produced models are shown in Figure 2-2 and explained as follows:
• “Establishing the story line”: transforms the scenario sequence of actions into a POM.
• “Elaborating things that change”: represents the transition of objects states as an OTM.
• “Identifying agents and their interactions”: shows the interaction between agents using an AOM that can be represented by a UML sequence diagram
• “Unravelling the goal and its sub-goals”: transforms the scenario’s goal into a GOM that can be represented as i* models.

A pilot usability study was conducted to test the approach. The test was based on rating the usefulness, learnability, efficiency, and satisfaction benchmarks. The results showed an encouraging reception of the approach and its usefulness. However, the study reported an efficiency limitation as building the four models for every scenario was considered time-consuming. The paper suggested further research to look into the possibility of generating some of the models automatically.
2.3 Business Process Model and Notation

BPMN is a standard graphical notation adopted by the Object Management Group (OMG) to represent business process activities. The BPMN specification [4] defines the metamodel, graphical notation, and serialization (interchange) format. BPMN aims to facilitate communication between business and technical stakeholders by providing an intuitive and easy-to-use standard model.

As per the specification, BPMN 2.0 supports three types of diagrams: Processes, Choreographies, and Collaborations. Process diagrams represent the business activities performed by an organization. There are five basic categories of BPMN graphical elements defined in the specification:

- **Flow Objects**: the main elements to define the behaviour of the process. Contain three elements: Events, Activities, and Gateways.

- **Connecting Objects**: There are four connecting objects that connect elements together: Sequence Flows, Message Flows, Associations, and Data Associations.

- **Swim Lanes**: There are two ways to group elements in swim lanes: Pools and Lanes.

- **Artefacts**: There are two artefacts that can be used for providing additional information: Group and Text Annotation.

- **Data**: There are four elements to represent data: Data Objects, Data Inputs, Data Outputs, and Data Stores.

Regarding the support of data handling, the BPMN specification states that data and messages flow can be shown as part of the diagrams; nevertheless, BPMN is not a data flow language. A business process can show the data items created or manipulated during the process life cycle and their associated activities. Data items can be physical or electronic resources, singular or collection, and persistent or volatile.

Figure 2-3 shows part of the BPMN2.0 metamodel (provided by Eclipse BPMN2 modeller project). The base BaseElement defines a string id attribute. The Definitions element is the top element in the BPMN2 model, which contains zero or moreRootElement, such as the Process element. The process element as a flow element container contains zero or more flow elements. A flow element can be a flow node (activities, events, and gateways), sequence flow, data object, or data object reference. Each sequence flow has one source and one target flow node.
Figure 2-3: BPMN2 Flow Elements, from Eclipse BPMN2.0 Ecore metamodel
Figure 2-4 shows part of the data elements in BPMN metamodel. Data in BPMN are represented as ItemAwareElement elements that can have an optional state defined by a DataState element. DataObject, DataInput, and DataOutput elements provide information about what activities require and/or produce.

Data needs and results for tasks and events are captured as InputSet and OutputSet, respectively. An input (output) set can reference many data inputs (outputs). A task can have many input or output set elements contained in an InputOutputSpecification element. Throw events (such as end events) can have only one input set while Catch events (such as start events) can have only one output set. Data objects can be referenced many times in the process using DataObjectReference elements. DataInputAssociation and DataOutputAssociation elements are used to associate data objects with activities or events. Data objects are contained in processes or subprocesses only.

It is worth mentioning that OMG defined two artefacts for the BPMN 2.0 metamodel. One format is based on Meta-Object Facility (MOF) and the other one on an XML Schema Document (XSD). Therefore, there are two serialization formats supported; an XMI-based and an XML (non-XMI) format.

### 2.4 UML State Machine

State machine diagrams are one of the UML behaviour models used to describe object’s states and transitions in a system. The UML 2.x specification [13] defines the metamodel and notation for the state machine diagram. State machine diagrams represent the life cycle of an object by showing the different states an object can exist and the events or triggers that cause the transition between the states. Transitions can be constrained by a Boolean condition called a guard. The diagram can also show the actions taken within a transition or state.
Figure 2-4: BPMN 2.0 data elements, from Eclipse BPMN2.0 Ecore metamodel
The state machine class diagram is shown in Figure 2-5. A StateMachine has at least one region and can have simple states or composite states. A Region is defined as an orthogonal part of either a state machine or a composite state that contains vertexes and transitions. Each transition has one source and one target state. A simple state does not have any sub-states or regions while a composite state contains one or more sub-states. State machine diagrams represent concurrent states by using composite states with multiple regions. The notion of pseudostates is defined to describe different types of abstract and complex vertexes. These include initial, fork, join, junction, and choice pseudostates. Fork and join pseudostates are used to represent splitting and merging from parallel states in a region. Choice and junction pseudostates show a decision branching point where only one condition is satisfied.

Figure 2-5: UML2 state machine, from Eclipse UML2.4 Ecore metamodel
2.5 Model-Driven Architecture

MDA architectural framework is OMG’s vision of the model-driven software development approach. MDA main idea is based on “separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform” [14]. Therefore, the MDA specification [14] defines three viewpoints on a system as shown in Figure 2-6 and described below:

- **Computation Independent Model (CIM):** focus on the domain under consideration and the system requirements while ignoring the technical details.
- **Platform Independent Model (PIM):** focus on the design of the system from a “technology-neutral” point of view without binding to a specific platform.
- **Platform Specific Model (PSM):** include additional details of a specific platform by adding the technical services provided by the platform to the PIM.

![Diagram of MDA levels of abstraction](image)

MDA is built on a foundation of standards defined by the OMG. The following standards are essential to the role of MDA:

- **UML [13]:** the de-facto standard used for modelling software solutions. UML is considered as a General Purpose Language (GPL) aimed to help software developers in the analysis, design and implementation of systems. The UML specification defines the abstract syntax using an MOF-based meta-model, the graphical notation, and the interchange format.
- MOF [16]: a universal standard used to describe different modelling languages. MOF defines an abstract syntax, called metamodel, which defines the syntactical constructs of the model and their semantics. A metamodel can be considered as a model of another model describing its elements and their relationships. MOF specification defines four meta-levels as shown in Figure 2-7. A model in each level is considered an instance of its above model (metamodel) if it conforms to it. The metamodel at level M3 is called a meta-metamodel because it describes metamodels at level M2. MOF M3 model is recognized as self-describing as it conforms to itself.

- OCL: a formal language which can be used to add more constraints to the model semantics that cannot be expressed in the metamodel. OCL expressions allow the definition of preconditions, postconditions, and guards for model elements where all instances of the model must conform to the metamodel and ensures that the constraints defined using OCL are also true. OCL is widely used in model transformation languages to query and access source and target models [15].

- XMI: an MOF-XML mapping standard used as the model exchange mechanism between different modelling tools. It specifies how to translate the model’s abstract syntax into an XML-based concrete syntax [15].
2.6 Models Transformation

Models transformation is at the heart of model-driven software development [17]. A transformation takes one or more source models as input and automatically generates one or more target models as output according to a transformation definition. A transformation definition is a set of transformation rules written in a transformation language. The transformation rules define how source model elements are mapped into target model elements and how the created target elements are initialized. A transformation tool is used to execute the transformation [9].

Figure 2-8 shows the principles behind models transformations. A transformation $Mt$ transforms a source model $Ma$, an instance of $MMa$, into a target model $Mb$, an instance of $MMb$. The transformation itself is considered as a model that conforms to its own metamodel. The transformation definition is written in a transformation language, and it handles source and target elements at the metamodel level. The transformation execution is a running instance of the transformation definition and handles model instances [15].

Figure 2-8: Model transformations [15]
Models can be transformed within the same abstraction level or between different abstraction levels such as from PIM to PSM. A detailed feature-based classification of model transformation approaches was implemented in [18]. Mens and Van Gorp [19] also proposed a similar taxonomy which includes aspects of tools and techniques available. A recent survey [20], based heavily on the previous two papers, unified and represented the characteristics of the classification. Table 2-2 shows a list of some of the main classification criteria used in the study.

Table 2-2: Model transformations classification, data taken from [20]

<table>
<thead>
<tr>
<th>Classification Criteria</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change of Abstraction</td>
<td>Vertical: change (increase or decrease) the level of abstraction.</td>
</tr>
<tr>
<td></td>
<td>Horizontal: change the representation but not the level of abstraction.</td>
</tr>
<tr>
<td>Change of Metamodel</td>
<td>Exogenous: source and target metamodels are the same.</td>
</tr>
<tr>
<td></td>
<td>Endogenous: source and target metamodels are different.</td>
</tr>
<tr>
<td>Number of Models</td>
<td>In-place: source and target models are the same (one model).</td>
</tr>
<tr>
<td></td>
<td>Out-place: source and target models are distinct.</td>
</tr>
<tr>
<td>Target Type</td>
<td>Model-to-Model: target model has a defined metamodel and elements.</td>
</tr>
<tr>
<td></td>
<td>Model-to-Text: target model is an arbitrary text.</td>
</tr>
<tr>
<td>Language Paradigm</td>
<td>Imperative/Operational: explicit control flow (like in Java/C languages).</td>
</tr>
<tr>
<td></td>
<td>Declarative/Relational: no explicit control flow.</td>
</tr>
<tr>
<td></td>
<td>Hybrid: support both imperative and declarative constructs.</td>
</tr>
<tr>
<td></td>
<td>Graph: models handled as graphs.</td>
</tr>
<tr>
<td></td>
<td>Template-based: used for Model-to-Text.</td>
</tr>
<tr>
<td></td>
<td>Direct Manipulation: use a general purpose programming language.</td>
</tr>
<tr>
<td>Rule Application Control</td>
<td>Implicit: no order control.</td>
</tr>
<tr>
<td></td>
<td>Explicit: order specified within rules.</td>
</tr>
<tr>
<td></td>
<td>External: order specified outside of rules.</td>
</tr>
<tr>
<td>Rule Scheduling</td>
<td>Rule Selection: control rule application (explicit)</td>
</tr>
<tr>
<td></td>
<td>Rule Iteration: use recursion, looping.</td>
</tr>
<tr>
<td></td>
<td>Phasing: specify the phases where rules can be executed.</td>
</tr>
</tbody>
</table>
2.7 Related Work

A number of studies have examined the area of model transformations in general. Few were found related to the transformation between process and statechart models. To my knowledge, none of them has addressed the transformation from BPMN 2.0 process models to UML (hierarchical) state machine diagrams. Table 2-3 gives a list of some related papers showing the source model, target model, and a description of the approach used in the study.

<table>
<thead>
<tr>
<th>Source Model</th>
<th>Target Model</th>
<th>Transformation Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPMN [21]</td>
<td>Object life cycle</td>
<td>Automatically generates a (flat) object life cycle model.</td>
</tr>
<tr>
<td>UML activity diagram [22]</td>
<td>Object life cycle</td>
<td>Generates a (flat) statechart for consistency checking.</td>
</tr>
<tr>
<td>BPMN process model [23]</td>
<td>BPMN process model</td>
<td>Explicit addition of data objects.</td>
</tr>
<tr>
<td>BPMN 1.0 process model [24]</td>
<td>UML activity diagram</td>
<td>Automatic transformation using ATL.</td>
</tr>
</tbody>
</table>

The most related work to this project is a study conducted by Eshuis and Van Gorp [5]. In this study, the authors presented an approach that takes a UML activity diagram and automatically constructs a hierarchical state machine diagram. GrGen¹, a graph transformation tool, was used for the implementation. The approach is divided into two main phases: the first phase filters the activity diagram by eliminating irrelevant nodes, and the second phase creates the statechart from the filtered activity diagram. The creation of statecharts is based on the algorithm defined in [7] for translating safe petri nets into hierarchical statecharts. The algorithm was extended to handle special situations, such as if the process model includes a “cross-synchronisation”, which is not supported in UML statechart diagrams. This approach is selected for this project and is explained in detail in the next chapters.

¹ http://www.info.uni-karlsruhe.de/software/grgen/
In [21], an approach was introduced to automatically generate an object life cycle from a business process. The process is assumed to have only XOR gateways, and data objects are created inside the process. The approach translates the business process (control and data) into Petri net notation and uses a “reachability graph” to construct the object life cycle. The output includes information about activities in the process that cause the transitions but does not consider composite states.

In [22], an approach was presented for consistency checking of UML activity diagram. The activity diagram, which has explicit object states, is mapped into an object life cycle using a set of generation rules. The state transitions are also labelled here with triggering activities names. The authors defined a notion of “compliance” and “coverage” to check the generated life cycle against a given one. Composite or concurrent states are not considered.

Meyer and Weske introduced an approach to extract data objects and their states from general process models, giving a usage example with BPMN [23]. The process models can have activities, gateways (xor, and), and control flow elements. The approach is based on using a language parser to parse activity’s labels based on a set of assumptions. The process model with extracted data objects can be used further for object model generation.

Cibran described an ATL\(^2\) transformation from BPMN 1.0 to UML activity diagrams [24]. The study defined a conceptual one-to-one mapping for some similar elements, one-to-many mapping for rich BPMN elements (such as events), and mapping for overloaded BPMN elements (such as gateways). However, data flow elements were not addressed in the transformation.

---

3 Overall Design

This chapter gives an overview of the system architecture, overall system design, and the different metamodels that are created or reused as part of the system design. The methodology used here for the BPMN to UML Statechart (BP2SC) implementation follows the approach used in the UML Activity Diagram to Statechart solution (AD2SC) [5], briefly mentioned in section 2.7. The latter is based on an algorithm introduced by Eshuis for translating petri nets to hierarchical statecharts (PN2SC) [7]. In the AD2SC solution, a number of “heuristic” filtering rules and an extension rule for this algorithm were presented. In order to complete the transformation chain, mapping rules from filtered models into petri nets and from hierarchical statecharts into UML state machines are derived. The algorithm and the associated rules are explained in the next chapter. The technology used for the implementation is described in chapter 5.

3.1 Assumptions

The BPMN2 specification is expressive and it defines a rich set of elements and notations (for ex. different task and event types). Covering the semantics of all these elements would require a substantial amount of time. To define the scope of the input model, only subset of the BPMN2 elements are considered. The following are assumed for the input:

- The input is a valid BPMN2 process model (in XMI format). The process model has only one Process element.
- The process model has one data object with multiple unique data states.
- Objects (states) have single incoming or outgoing associations.
- The process has one start event, at least one end event, (exclusive/parallel) gateways, and tasks.

As per [25], the elements specified in the last assumption are considered the most commonly used elements in the language. As a starting point, the input covered here has similar semantics to UML2 activity diagrams. While the two standards have some differences in their notations and expressivity, both can similarly represent many of the workflow patterns [26]. The semantic differences (such as optional and alternative data) and their effect in the used approach are left for future work.
3.2 System Architecture

Based on the AD2SC approach, the transformation system is considered as a chain of model transformation or modification tasks instead of a direct single transformation. The system is divided into two main high-level components: “filtering” component and “synthesising” component, as originally named in [5]. Each of the high-level components is divided into three related modules which contain the final program units. Each of the modules performs a specific model management task as part of the transformation chain. Figure 3-1 shows the system architecture consisting of the aforementioned components and their modules.

If we look to the system as a black box, it takes a valid business process model conforming to BPMN2 metamodel in XMI (or XML format) and generates a statechart model in XMI format according to the UML (2.4) metamodel, if the transformation succeeds. The transformation problem is divided into two sub-problems: filtering the input process model from any irrelevant nodes, and creating a statechart from the filtered model. The description of the components is given in the next sequel.
3.2.1 Filtering Component

The main aim of this component is to take a valid input business process model and produce a filtered model having only object nodes and relevant control nodes, by removing any activity nodes and irrelevant control nodes. The first validation step validates the input process model against a set of defined constraints which checks that the input can be handled by the system. The preprocessing stage takes a valid input process model and produces an output model in the normal form. The filtering step generates a filtered model out of this normalised process model. The filtered model has only object nodes and relevant control nodes; ones that affect the object’s state.

3.2.2 Synthesising Component

The role of this component is to generate the desired UML statechart model from the filtered process model. For this component, the system reuses a PN2SC solution implemented as part of the Transformation Tool Contest\(^1\) (TTC13) [27]. The reused module is the main module in this component as it implements the PN2SC translation algorithm. It generates a hierarchical statechart from a given petri net input. Therefore, the filtered process model needs to be mapped into Petri net. The petri net mapping module takes the filtered model output generated from the filtering component and creates a petri net model which acts as an input to the petri net to statechart hierarchy module. Finally, a mapping module is used to map the generated hierarchical statechart into a corresponding one with UML2 element types. The output statechart model is a valid UML2 state machine model which can be imported into different UML tools.

3.3 System Modules

The transformation chain consists of different model management modules that are connected together in a workflow. Figure 3-2 shows the orchestration of the model transformation tasks, represented as a process model. The different modules are shown as tasks while the input models, metamodels, and the generated output models are represented as data objects. As shown in the diagram, the system uses five different metamodels for the different models generated throughout the process. Two of them are the BPMN (source) metamodel and the UML (target) metamodel. The remaining metamodels are described later in section 3.4. The workflow diagram also shows the possible failure points in the chain: if the model does not satisfy the made assumptions,

---
if it cannot be normalised, or if the algorithm cannot generate a statechart. The individual modules are explained in the following subsections.

### 3.3.1 Input validation

This module is the entry point in the transformation chain. Model validation is not in fact a transformation task, but it provides a useful feature to check the input model before applying any processing. The module does not check for BPMN syntax or semantics correctness as this is not the main goal of the project; instead, the input is expected to be a valid BPMN2 model as a prerequisite. However, the constraints checked in this module are to make sure that the input model satisfies the assumptions specified in section 3.1. For example, multiple data objects are not considered for this project, so the input must have one stateful data object. If any of the specified constraints is violated, the system will exit execution displaying a message with the unsatisfied constraints.

### 3.3.2 Preprocessing

After the input model passes the validation step, the model is then transformed into normal form. A process in the normal form has all activity nodes with exactly one incoming and one outgoing edge, and all object nodes have at least one incoming and outgoing edges. Normalisation of the input model means that “each node is on a directed path from the initial to a final node.” [28]. Because this step may introduce object nodes that are connected to control nodes, the generated output is not a valid BPMN2 model. Therefore, an evolved BPMN metamodel (section 3.4.1) is created where object nodes are also considered as flow nodes. Before applying the preprocessing rules, the input model is migrated from the original BPMN metamodel into this evolved metamodel. At the end of this module, the generated output model is in normal form. This step might fail as will be explained later in section 4.2.1.

### 3.3.3 Filtering

This module takes the normalised process model and removes any irrelevant nodes. The module reduces the input model by repeatedly applying the filtering rules, shown later in section 4.2.2, until no more filtering rules are applicable. At the end of this module, the generated output model has only object nodes and control nodes that participate in the object life cycle.
3.3.4 Petri net mapping

This module maps the filtered process model into a petri net model in order to reuse the PN2SC solution. The mapping rules described in section 4.2.3 are applied to generate an equivalent petri net for the filtered model. At the end of the migration process, the (original, target) elements mapping is preserved in a map for “traceability” as will be explained in the next chapter.

3.3.5 Petri net to statechart hierarchy

This module is reused from the petri net to statechart case solution as part of TTC13 [27]. It implements the translation algorithm (AND/OR rules) described later in section 4.2.4. The module has been extended by implementing the “cross-synchronisation” rules. Another trace links are needed here at the end of this migration, similar to the ones after the petri net mapping. At the end of this step, the petri net model is reduced into a single place, and an equivalent statechart tree is created; otherwise, the system fails if the petri net cannot be translated into a hierarchical statechart.
3.3.6 UML2 state machine mapping

This module is the last step in the transformation chain. The previous step generates a hierarchy of states and transitions that conforms to the statechart metamodel, shown later in section 3.4.2. The migration in this module is an application of the mapping rules described in section 4.2.5, which maps from the statechart metamodel into the UML2 metamodel. This module makes use of the trace links (maps) created from the last two migration modules to retrieve the type of original elements in the filtered model that has been migrated into states.

3.4 Metamodels

This section introduces the metamodels that define the different models used in the transformation system. The BPMN2 and UML2 state machine metamodels were shown previously in section 2.3 and 2.4, respectively. The metamodels used for the system are Eclipse’s realization of the two OMG standards provided as part of the Eclipse Modelling Project. The Petri net and Statechart Ecore metamodels reused here were developed as part of the petri net to statechart transformation case used in the TTC13 [27]. The BPMN2 normalised Ecore metamodel is created here to define normalised and filtered process models, which have minor differences from the original BPMN metamodel as explained in the following section.

3.4.1 Normalised BPMN2 Ecore

This metamodel, shown in Figure 3-3, takes only parts of the BPMN2 metamodel relative to the transformation. The only difference from the original BPMN metamodel is that data object/data object reference elements are both represented as ObjectNode elements, and data input/output associations are represented as ObjectFlow elements in the new metamodel. Object nodes here are considered as flow nodes as this might happen after the normalisation and filtering steps. The Edge abstract class is introduced for both object and sequence flows.

There are two reasons for the design decision of this metamodel. First, the model generated after the filtering rules can have object nodes connected to control nodes, which is not valid in BPMN. Therefore, object nodes had to be considered as flow nodes as well. The second reason is that the definition of object flows similar to sequence flows rather than BPMN data input/output associations makes the model navigation written code simpler.
3.4.2 Petri net and statechart Ecore

The petri net metamodel defines a *Net* element that contains zero or more transitions and places. Each place has zero or more input and output transitions, or similarly, each transition has zero or more input and output places.
The statechart metamodel defines a *Statechart* root element which contains a single *AND* top state. Both *AND* and *OR* states are compound states which contain other states. *Basic* states are atomic/leaf states, and *HyperEdge* states represent transitions that connect states. Each state element has a name attribute and zero or more next and previous states.

The two metamodels are shown in Figure 3-4. For statecharts defined in the PN2SC algorithm, AND nodes contain only OR nodes, and OR nodes contain only Basic/AND nodes.

### 3.5 Summary of Transformation Phases

The processes involved in the transformation approach can be organised into four phases. First phase includes preprocessing and filtering of the input model. The second phase is a mapping toward the petri net formalism. The third phase consists of the actual building of the statechart hierarchy from petri net. The last one is a postprocessing step which maps the output to the desired UML state machine syntax. These phases are summarised in Table 3-1.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Validation</td>
<td>Validating input BPMN model satisfies the assumptions.</td>
</tr>
<tr>
<td></td>
<td>Preprocessing</td>
<td>Normalisation of input process model.</td>
</tr>
<tr>
<td></td>
<td>Filtering</td>
<td>Removing irrelevant activity and control nodes.</td>
</tr>
<tr>
<td>2</td>
<td>Petri net mapping</td>
<td>Integration step to map filtered process models into petri net.</td>
</tr>
<tr>
<td>3</td>
<td>Creating hierarchical statechart</td>
<td>Translation of petri net into hierarchical statechart.</td>
</tr>
<tr>
<td>4</td>
<td>UML state machine mapping</td>
<td>Mapping the statechart hierarchy into the corresponding UML state machine elements.</td>
</tr>
</tbody>
</table>
4 Transformation and Mapping Rules

This chapter illustrates the theory behind the transformation approach used for the system. The PN2SC translation algorithm introduced by Eshuis [7] for translating petri nets to hierarchical statecharts (PN2SC) is shown, as well as the preprocessing and filtering rules defined in the AD2SC solution [5]. In the latter, an extension rule for the algorithm was also introduced to handle “cross-synchronisation” (two parallel branches that cross-synchronize together). The reader is invited to refer to the referenced papers for a complete formal definition of the rules. In addition, the rules derived for mapping filtered models into petri nets, and mapping hierarchical statecharts into UML state machines are also presented. First, a definition for the concepts used is given, as can be found in [7] [5].

4.1 Concepts

Definition 1: Process Model

A process model is a tuple \((A, C, O, S, SF, DA, \text{type})\) where:

- \(A\) is a finite set of activity nodes (includes set of tasks and events).
- \(C\) is a finite set of gateways (also called control nodes).
- \(O\) is a finite set of data objects (also called object nodes).
- \(S\) is a unique data state referenced by each object node.
- \(SF \subseteq (A \cup C) \times (A \cup C)\) is the sequence/control flow between activity or gateway nodes.
- \(DA \subseteq (A \times O) \cup (O \times A)\) is the output or input data association (also called object flow) for activity nodes.
- \(\text{type}(C) \rightarrow \{\text{xor, and}\}\) assigns an exclusive or parallel type to each gateway.

The process model is assumed to have one start event and can have multiple end events. The gateways include exclusive gateways (decision/merges) and parallel gateways (forks/joins). Here we assume there is one data object which is referenced many times in different states. The data states are unique.

A filtered process model is a process model where an edge \(E \subseteq (A \cup C \cup O) \times (A \cup C \cup O)\) connects two flow nodes. This means that object nodes are also flow nodes which can connect to gateways.
Definition 2: Hierarchical Statechart

A statechart is a tuple \((N, H, \text{source}, \text{target}, \text{type}, \text{child})\) where:

- \(N\) is a finite set of nodes (also called states).
- \(H\) is a finite set of hyperedges (also called transitions).
- \(\text{source}(H)\) is a function which gives the set of input nodes for a hyperedge.
- \(\text{target}(H)\) is a function which gives the set of output nodes for a hyperedge.
- \(\text{type}(N) \rightarrow \{\text{BASIC, AND, OR}\}\) is a function which specifies the type of a node.
- \(\text{child}(N)\) is a function which returns the immediate children of a node.

In a statechart, there is exactly one root node and each node except the root has exactly one parent. BASIC nodes are leaf nodes, AND nodes represent parallelism, and OR nodes represent exclusiveness. As a constraint here, AND nodes contain only OR nodes, and OR nodes contain BASIC/AND nodes.

Definition 3: Petri Net

A petri net is a tuple \((P, T, F, \pi)\) where:

- \(P\) is a finite set of places.
- \(T\) is a finite set of transitions.
- \(F \subseteq (P \times T) \cup (T \times P)\) is a flow relation between places and transitions.
- \(\pi\) is the initial place.
- \(\text{pre}(p)\) and \(\text{post}(p)\) is a function which returns input and output transitions for a place \(p\).
- \(\text{prep}(t)\) and \(\text{postp}(t)\) is a function which returns input and output places for a transition \(t\).

A petri net marking assigns for each place a number of tokens (visualised as a black dot). In a safe petri net, each place has a maximum of one token in any possible marking.

4.2 Transformation Rules

The following subsections give a definition for the different transformation and mapping rules that are applied in the corresponding system modules shown in the previous chapter.

4.2.1 Preprocessing rules

In contrast to activity nodes in UML activity diagrams, BPMN specification permits activities with multiple incoming or outgoing sequence flows (splitting and merging). Hence, a new simple initialization rule \(PI\) is added in which the multiple (outgoing/incoming) sequence flows are replaced with an equivalent gateway (fork/merge). This rule, shown in Figure 4-1, is specified in the BPMN specification [4].
Notice that, as a matter of consistency, the UML fork/join notation is used in the following figures instead of the BPMN notation.

The objects processing step ensures that each object node has at least one incoming and outgoing data flow. This applies for object nodes with no outgoing (incoming) data flow. The conditions for $P2$ rule shown in Figure 4-1 must be met or otherwise this step fails. The negative conditions are shown with a cross symbol. For example, $P2(a)$ rule is for object nodes with no outgoing edges. The object node must be connected to a unique activity node (must not be output of another activity), and at the same time this activity node has a single outgoing sequence flow and no other output object node. Then, the source of the sequence flow from the unique activity node is changed to the object node as shown on the right-hand side of $P2(a)$. Similarly, rule $P2(b)$, is applied for data objects with no incoming object flows. If this step fails, the user has to correct the model manually.

The activities processing step ensures that each activity node has exactly one incoming and outgoing edge. After applying the initialization step, each activity will have one incoming and outgoing sequence flow. By applying $P3$ rules shown in Figure 4-1, a fork (join) gateway will be added for each activity node with output (input) object nodes.
The output after this step is a process model in the normal form. Table 4-1 gives a summary of the preprocessing rules.

<table>
<thead>
<tr>
<th>Table 4-1: Summary of preprocessing rules</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input preprocessing rules:</strong></td>
</tr>
<tr>
<td><strong>P1:</strong> Add a (fork/merge) for activity nodes with multiple (outgoing/incoming) sequence flows.</td>
</tr>
<tr>
<td><strong>P2:</strong> Object nodes with no (outgoing/incoming) edge are set as the (source/target) of the (outgoing/incoming) sequence flow of the associated activity node. The activity node should be unique and must not have another (outgoing/incoming) edge or sequence flow.</td>
</tr>
<tr>
<td><strong>P3:</strong> Add a (fork/join) for activity nodes with multiple (outgoing/incoming) edges.</td>
</tr>
</tbody>
</table>

### 4.2.2 Filtering rules

The filtering step applies a set of “heuristic” filtering rules [5] to the normalised process model in an iterative and arbitrary order. The filtering rules remove irrelevant activity and control nodes that do not affect the object’s states. The filtering rules in Figure 4-2 are summarized in Table 4-2. Another straightforward rule not explicitly mentioned in AD2SC is to remove redundant edges between two flow nodes.

![Filtering rules](image)

*Figure 4-2: Filtering rules [5]*
An important feature of the filtering rules is “confluence”; meaning that no two different rules can be applied to the same node at any given time [5]. This ensures the outcome of the filtering rules is deterministic: the output will always be the same when the rules are applied to a given model in any order. After these rules are applied, a filtered process model with only data objects and control nodes (which controls the object’s behaviour) is produced.

### Table 4-2: Summary of filtering rules [5]

<table>
<thead>
<tr>
<th>Filtering rules:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R1:</strong> removes all nodes (except data objects) with exactly one incoming and</td>
<td><strong>R2:</strong> merges pairs of exclusive gateways (decisions) where the</td>
</tr>
<tr>
<td>one outgoing edge.</td>
<td>successor has only one incoming edge from the predecessor</td>
</tr>
<tr>
<td><strong>R3:</strong> merges pairs of exclusive gateways (merges) where the predecessor has</td>
<td><strong>R4:</strong> removes a self-loop from an exclusive gateway.</td>
</tr>
<tr>
<td>only one outgoing edge to the successor.</td>
<td></td>
</tr>
<tr>
<td><strong>R5:</strong> removes a redundant edge connecting a parallel gateway (fork) to</td>
<td><strong>R6:</strong> removes an end event following a parallel gateway (fork).</td>
</tr>
<tr>
<td>another parallel gateway (join), if there is another path between the two</td>
<td></td>
</tr>
<tr>
<td>gateways.</td>
<td></td>
</tr>
<tr>
<td><strong>R7:</strong> merges pairs of parallel (forks) gateways where the successor has only</td>
<td><strong>R8:</strong> merges pairs of parallel (joins) gateways where the</td>
</tr>
<tr>
<td>one incoming edge from the predecessor.</td>
<td>predecessor has only one outgoing edge to the successor.</td>
</tr>
<tr>
<td><strong>R9:</strong> removes a redundant exclusive gateway (decision) which is in complete</td>
<td><strong>R10:</strong> removes a redundant exclusive gateway (merge) which is</td>
</tr>
<tr>
<td>parallel to a data object (guard conditions are moved to the data object</td>
<td>in complete parallel to a data object.</td>
</tr>
<tr>
<td>edges).</td>
<td></td>
</tr>
</tbody>
</table>
4.2.3 Petri net mapping rules

This migration is the integration point between the filtering phase and the statechart building phase. A mapping of BPMN to Petri nets is described in [29]. The mapping here concerns only with object nodes and gateways (beside start and end events). The mapping rules are shown in Figure 4-3 and summarised in Table 4-3. The mapped petri net represents the behaviour of the objects nodes in the process model.

**Figure 4-3: Filtered process model to Petri net mapping rules**

**Table 4-3: Summary of petri net mapping rules**

<table>
<thead>
<tr>
<th>Petri net mapping rules:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>N1:</strong> Maps each start event, end event, exclusive gateway, and object node into a place. Start events will be mapped into places with no pre-transitions, and end events will be mapped into places with no post-transitions.</td>
</tr>
<tr>
<td><strong>N2:</strong> Maps each parallel gateway into a petri net transition.</td>
</tr>
<tr>
<td><strong>N3:</strong> Each edge is mapped into a petri net arc connecting the equivalent places and transitions. If both the source and target of the edge are places, a dummy transition is added between them. Similarly, a dummy place is added if both the source and target are transitions. This is to conform to the petri net definition.</td>
</tr>
</tbody>
</table>
4.2.4 Petri net to hierarchical statechart translation rules

Eshuis presented an algorithm for translating petri nets into hierarchical statecharts [7]. The algorithm is described as “structured and behaviour preserving”. “Structure-preserving” means that the input and output models have a similar structure, and “behaviour-preserving” means that the execution semantics for the two models are the same. The algorithm achieves structure-preserving feature by having a one-to-one correspondence between petri net and statechart elements. Each petri net place is mapped into a BASIC state, and each transition is mapped into a hyperedge. The algorithm then adds the proper OR/AND nesting into the structure.

The algorithm is proved correct and complete for a subclass of petri nets. This subclass of safe petri nets is formally defined in [7]. Briefly speaking, with “structure/behaviour-preserving” in mind, unsafe petri nets are not translatable into statecharts. On the other hand, not every safe petri net has an equivalent statechart. Moreover, some petri nets have an equivalent statechart where the algorithm fails to translate. A summary of the input models classification is given in the evaluation section 7.4.1.

The petri net here represents the filtered process model (object and control nodes) which has been mapped in the previous phase. In petri nets, concurrency is represented by a fork/join transition with multiple input or output places. Similarly, in statecharts a transition with multiple input or output states indicates the states are in parallel. This is represented in UML statecharts as composite (AND) states with multiple regions (OR). If a petri net has “balanced” fork/join transitions (each fork has all its paths merged in a subsequent join), an AND state is created for each fork/join pair. Moreover, if the fork/joins are unbalanced, still there can be an equivalent statechart.

The algorithm first maps the petri net places and transitions into statechart BASIC nodes and hyperedges, respectively (initialization). Then, it builds the statechart tree by processing the petri net transitions and reducing both the petri net and the statechart by applying the (AND/OR) rules. The rules are explained as follows:

**Initialization step**

- Each place $p$ in the petri net is mapped into a BASIC node $p$ which is a child of a new OR node $o_p$.
- Each transition $t$ in petri net is mapped into a hyperedge $t$ in the statechart tree.
**AND reduction rule (T1)**

For each processable transition \( t \), the input places of \( t \) are reduced provided that the set of \( \text{prep}(t) > 1 \) and each place \( p \) in the set has same \( \text{pret}(p) \) and same \( \text{postt}(p) \). The following changes are applied:

- The petri net is reduced by replacing the set of places with a new place \( p_a \). Each other transition \( t' \) in \( T \) which has any place in the set \( \text{prep}(t) \) in its \( \text{prep}(t') \) or \( \text{postp}(t') \) will be changed with \( p_a \) (once if it has many places in the set).
- The statechart trees are reduced by adding a new AND state \( a \) as parent of all the OR nodes equivalent to the set of places in \( \text{prep}(t) \). A new OR node \( o \) is added as the parent of \( a \).

A symmetric rule is applied for reducing the output places of \( t \) provided that the set of \( \text{postp}(t) > 1 \) and each place \( p \) in the set has same \( \text{pret}(p) \) and same \( \text{postt}(p) \). This rule is shown in Figure 4-4 (a).

The conditions in the rule are necessary or otherwise the generated statechart will not be valid, or will have different behaviour. A transition \( t \) must have multiple input or output places to represent concurrency. If not all the input places or output places have the same transitions as input and output, this means there is a transition \( t' \) which partially enters or partially leaves an AND state, which is inconsistent with statechart syntax.

**OR reduction rule (T2)**

For each processable transition \( t \) with single input place \( q \) and single output place \( r \), such that \( q \neq r \), and there is no transition \( t' \) which has both \( q \) and \( r \) in its \( \text{prep}(t') \) or \( \text{postp}(t') \), then:

- The petri net is reduced by replacing the transition \( t, q, \) and \( r \) with a new place \( p_o \). Each transition \( t' \) in \( T \) which has \( q \) or \( r \) in its \( \text{prep}(t') \) or \( \text{postp}(t') \) will be changed to \( p_o \).
- The statechart trees are reduced by merging the two OR nodes equivalent to \( q \) and \( r \) into a single OR node \( o \) which has the children of both OR nodes as its children.

These rule conditions are again necessary as if otherwise there is a transition which has both places as input or output, this means that the states should be parallel and not in one OR node. This rule is shown in Figure 4-4 (b).
If the set of transitions $T$ is empty, this means the petri net has been reduced into a single place and the statechart tree into a single tree with this single place as the OR root of the tree. Otherwise, there are some transitions that cannot be processed, and there is no single root node; hence, the reduction rules fail to construct a structure/behaviour-equivalent statechart.

![Diagram](image)

**Figure 4-4: AND/OR reduction rules defined in the PN2SC algorithm [7]**

**Cross-synchronisation rule (T3)**

If the petri net is not reduced into a single place (some transitions cannot be processed) then this extension rule can be applied. This rule is applicable only when both the AND and OR rules cannot be applied.

The rule is applied if there is a place $p$ between a fork $f$ and a join $j$ transitions; then:

- All the edges connected to the place are removed. This means that the pre($p$) and post($p$) = $\emptyset$.

After removing the edges from the place, the AND rule or OR rule may become applicable again. If this is the case, and the set of transitions $T$ became empty afterwards, then the result has no single place as the previously processed place $p$ is in parallel with other places. The following rule is applied:
- The petri net is reduced by merging the remaining places into a single place.
- The statechart tree is reduced by applying the AND rule; merging the OR nodes into one AND node.

A final post-processing step can be applied to adjust the initial state of the statechart. The generated statechart have a similar structure to the petri net, but inconsistent behaviour as will be explained later in section 7.4.1; there exists a transition connecting two parallel states. The following refactoring rules are applied.

**Refactoring rules (T4)**

The refactoring rules can be applied if the cross-synchronisation removing rule is applied. The statechart generated does not have same behaviour as the petri net, so a refactoring step is added which uses transition events, triggers, and guards. The following refactoring steps are applied for each OR state equivalent to a place \( p \) that was processed with T3 rule:

- A new initial state, \( init \), is added as the default state for the OR state.
- For every fork transition \( t \) that enters a BASIC state \( b \) in the OR state from its outside, the edge from the transition \( t \) to \( b \) is removed, and the edge to \( t \) is updated with an event \( e \). A new transition from \( init \) to \( b \) is added with trigger condition \( e \). This implies that the state \( b \) is reachable only if transition \( t \) is taken.
- For every join transition \( t \) that leaves a BASIC state \( b \) in the OR state to its outside, the edge from state \( b \) to the transition \( t \) is removed, and the edge \( t \) is updated with a guard condition \( in[b] \) and a generated event \( e \). A new transition from \( b \) to \( init \) is added with trigger condition \( e \). This implies that the transition \( t \) is taken only if state \( b \) is active, and once \( t \) is fired, state \( b \) is also left.

If the cross-synchronisation rule cannot be applied, then the algorithm fails to generate a statechart for the petri net.

**4.2.5 UML2 state machine mapping rules**

This final step is added to map the hierarchical statechart model elements into UML2 state machine elements. The mapping is straightforward as the hierarchical statechart tree already includes the desired nesting of states and edges.

The only additional step required is to create “traceability links” between the statechart elements (target) and the filtered process model elements (source) [30]. After the previous phase, BASIC states in the statechart model represent both events, exclusive
Chapter 4: Transformation and Mapping Rules

gateways, and object nodes. Each of these nodes has a different model element in the UML2 model as shown in section 2.4. Therefore, the traceability links enable to return their original type in the filtered model. This can be implemented in different ways. In this project, a map $M_1$ [filtered element, petri net element] is created between the source filtered model elements and the migrated petri net elements at the end of the migration. A second map $M_2$ [petri net element, hierarchical statechart element] is created between the source petri net elements and the migrated hierarchical statechart elements after executing the initialization step of this migration. From the two maps, the original element type (from the filtered model) for each statechart node can be obtained. Another way would have been to add a type attribute for the petri net or statechart definition (metamodel), but this seemed unnecessary changes for the metamodels. The mapping rules are shown in Figure 4-5 and summarised in Table 4-4. For all the rules, the nesting is preserved by setting the container for the UML elements equal to the tree node’s parent equivalent element.

- **Basic**: AND, OR
- **Composite state**: Region, Composite state
- **Region**: Initial state, if Basic source type is Start Event
- **Composite state**: Final state, if Basic source type is End Event
- **Decision**: Decision, if Basic source type is XOR Gateway
- **State**: State, if Basic source type is Data object
- **HyperEdge**: Transition
- **HyperEdge fork**: Transition fork
- **HyperEdge join**: Transition join

![Diagram of UML state machine mapping rules](image)

*Figure 4-5: UML state machine mapping rules*
### Table 4-4: Summary of the UML state machine mapping rules

**UML state machine mapping rules:**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>M1:</strong></td>
<td>Maps each OR state into a UML2 region. If the OR node has the root AND as parent, the region’s container will be the UML state machine element.</td>
</tr>
<tr>
<td><strong>M2:</strong></td>
<td>Maps each AND state (except the top AND state) into a UML composite state.</td>
</tr>
<tr>
<td><strong>M3:</strong></td>
<td>Maps a BASIC state into a UML pseudostate, final state, or state. If the BASIC state type returned from the traceability function is a start event (exclusive gateway), the migrated element will be an initial (decision) pseudostate. If the type is an end event, the corresponding UML element will be a final state. Otherwise (the type is object node), the equivalent will be a UML (simple) state.</td>
</tr>
<tr>
<td><strong>M4:</strong></td>
<td>Each hyperedge with single incoming and single outgoing edge is mapped into a UML transition. The source of the transition is set to the equivalent state of the hyperedge source, and the target is set to the equivalent state of the hyperedge target.</td>
</tr>
<tr>
<td><strong>M5:</strong></td>
<td>Each hyperedge with fork-like behaviour is mapped into a UML fork pseudostate, and each hyperedge with join-like behaviour is mapped into a UML join pseudostate. The source and target of the fork (join) is set similarly to rule M4.</td>
</tr>
</tbody>
</table>
5 Technology Used for Implementation

This chapter presents the technology selected for implementing the system. Eclipse Modelling Project (EMP) provides a comprehensive modelling platform by implementing many of the MDA standards. The Epsilon platform was chosen for implementation over some other options. In the following sections, the rationale for the selection is explained by elaborating the main features of the languages and tools used. In addition, the base eclipse modelling tools used for the system are shown.

5.1 Eclipse MDA Support

The MDA specification defined by OMG has been implemented by different commercial and open source groups. The Eclipse Modelling Project [31] is an example of a widely used open source MDA implementation. This section gives a brief overview of the Eclipse Modelling Project and its main subprojects.

5.1.1 Eclipse Modelling Project

The Eclipse Modelling Project\(^1\) is one of the top-level projects in Eclipse that provides a set of modelling frameworks, tools, and standard implementations. The project contains a number of specialized subprojects such as Abstract Syntax Development, Concrete Syntax Development, Model Development Tools, and Model Transformations. Figure 5-1 shows an overview of the different features and key standards available from the Eclipse Modelling Project.

\(^1\) http://www.eclipse.org/modeling/
5.1.2 Eclipse Modelling Framework

Eclipse Modelling Framework (EMF) is central to the Eclipse Modelling project. EMF includes a core framework that provides abstract model definition and XMI serialization features. It also includes components for building editors and Java classes for EMF models. Ecore is the abstract syntax component in EMF that is “closely aligned” to OMG’s Essential MOF.

5.1.3 Model Development Tools

The focus in Model Development Tools (MDT) is to provide an implementation of industry standard metamodels. BPMN 2.0 and UML 2.0 specifications are implemented as part of this subproject.

The BPMN MDT\(^2\) component provides a simple graphical interface that allows designing and modelling of BPMN process diagrams. It supports the two OMG metamodel formats, MOF and XSD. Therefore, the serialization of the models generated can be either XMI-based or XML-based. The default format used is the XML-based [33]. UML2 component provides an implementation of OMG UML 2.x specifications with an XMI schema for models interchange. Papyrus\(^3\), which superseded Eclipse UML2 Tools\(^4\), provides an environment for building EMF models, particularly UML models.

5.2 Why Epsilon?

It can be noticed from the model transformation classifications outlined in Table 2-2 that a number of suitable model transformation tools and languages exist for different transformation problems. The tools and language choice is based on the transformation problem and features of the language. Initially, Atlas Transformation Language (ATL) was considered for the system implementation because of its popularity and simplicity. ATL is a hybrid language widely used for out-place, exogenous Model-to-Model (M2M) transformations. However, it appeared that ATL might not be suitable because the system transformation chain includes copying, removing as well as creating new model elements. Epsilon provides a set of task-specific languages for different model management tasks such as model migration, transformation, and validation which makes it a suitable choice. Epsilon was chosen over GrGEN, used in the AD2SC

\(^2\) http://wiki.eclipse.org/MDT-BPMN2
\(^3\) http://www.eclipse.org/papyrus/
\(^4\) http://wiki.eclipse.org/MDT-UML2Tools
solution, mainly because its language syntax seemed more familiar for the author, compared to the other graph transformation tool. Although Epsilon languages have their specific declarative syntax, being based on Java made it easier to learn. It turned out also that it has a very active and helpful user community through its online forum.

Epsilon\(^5\) [34] is a model management platform which provides a set of languages and tools for different model management tasks. Epsilon supports uniformly many modelling technologies, such as EMF, through a model connectivity layer. It can be distributed as an Eclipse plug-in or as a small footprint Eclipse modelling-IDE, enabling the use of Eclipse’s development tools. The base language in Epsilon is the Epsilon Object Language (EOL) which is considered as a general-purpose imperative language. EOL provides built-in primitive types and model element types. It also supports OCL-similar syntax and can be used to navigate and modify EMF models.

Other task-specific model management languages are built on top of EOL. The list of languages includes Epsilon Validation Language (EVL), Epsilon Flock, and Epsilon Pattern Language (EPL), to name a few. EVL is a validation language that supports checking models constraints. Epsilon Flock is a rule-based language used to update models with regard to changes in their metamodels (model migration). EPL is a pattern matching language that can be used for in-place model transformation [34]. The hybrid (imperative and rule-based declarative) syntax provided by Epsilon makes the transformation definition concise and understandable, with control flow flexibility at the same time. Figure 5-2 shows an overview of Epsilon platform architecture.

---

\(^5\) http://www.eclipse.org/epsilon/
The following sections describe the model languages and tools selected for the BP2SC system implementation.

5.2.1 Epsilon Validation Language

Epsilon Validation Language (EVL) [34] provides model validation capabilities by defining constraints and conditions on models. The constraints are specified within a context, which represents the set of model instances that will be evaluated against the constraint. The constraint can have a guard that specifies the condition to be met. If this is not satisfied, an appropriate error message can be displayed to the user. EVL supports two types of constraints: errors and warnings. Error constraints are critical and cause the execution to abort. EVL also provides a feature for fixing those validation errors programmatically.

Therefore, EVL was used for implementing the BPMN input validation module for checking the input model constraints shown previously in section 3.3.1.

5.2.2 Epsilon Flock Migration Language

Epsilon Flock [36] provides model migration capabilities. Model migration can be considered as a Model-to-Model transformation where the source (called original) metamodel is similar to the target (called migrated) metamodel. Typically, in model migration only some model elements need to be updated “in response to metamodel evolution”.

Compared to general Model-to-Model transformation languages, such as ATL, flock minimizes the need to write a large portion of the transformation code. The reason is that the flock engine’s execution semantics involves a “conservative copying algorithm” which automatically copies all elements which conforms to the migrated metamodel and not affected by the migration. This generally needs to be specified in other Model-to-Model languages. Therefore, flock modules can usually be smaller in size in comparison to other modules written in Model-to-Model transformation languages [36]. Flock provides two variables in each migration block: original and migrated; which refers as the names imply to the original and migrated elements for the migration rule under application. Flock also provides a built-in operation, named equivalent(), which when called in any original element returns the corresponding migrated element.
For these mentioned features, flock is well-suited for the mapping rules defined in the previous chapter (filtered process model to petri net, petri net to hierarchical statecharts, and hierarchical statecharts to UML statecharts) because the source and target metamodels are very similar. It is also suitable for the migration from the original BPMN metamodel to the evolved one where there are only minor changes.

5.2.3 Epsilon Pattern Language

Epsilon Pattern Language (EPL) provides pattern-matching features against instances of model elements. EPL patterns specify what elements in the input model have to be matched using roles (element types) and guards. EPL engine checks the input model against all the patterns defined in an EPL module and an onmatch block can be specified for execution when a match occurs. When the phase of all successful matches finishes, the do block for those matched patterns is executed. EPL supports an iterative mode where model matching process will run until there are no more matches, or until the specified maximum number is reached.

EPL is typically used for in-place transformations or models reduction where a pattern rule has a changing side-effect on the input model. Therefore, it is a perfect choice for implementing the filtering reduction rules specified in section 4.2.2.

5.2.4 ANT Orchestration Workflow

Epsilon provides a set of development tools that complement the model management languages described in the previous sections. In complex transformation systems, model management tasks are usually combined to form a complete workflow. Therefore, there is a need for a tool that facilitates this requirement and automates composing different model management tasks [34], which is the case for the BP2SC transformation system. Epsilon provides an ANT orchestration workflow framework which extends the commonly used ANT framework. Epsilon ANT provides a set of core tasks and model management tasks for each of its languages.

The core tasks include a LoadModelTask to load a model from a location (file location) into the project context, and StoreModelTask to save the model back to a target location. The project context provides a shared repository accessible by all Epsilon tasks which can be used for tasks integration. Tasks can export variables to the shared context for other tasks to reuse using the <exports> and <uses> nested task tags. This feature came into use for implementing the traceability links described in section 4.2.5. The petri net
and statechart migration modules export the trace links (maps) which are used later by the final UML state machine mapping flock module.

The Epsilon management tasks for the task-specific languages use the src attribute to specify the source file for the module and the ref attribute to reference loaded models. The EVL task executes an Epsilon validation module, and it supports an additional failOnError attribute which if set to true will cause a build exception when there are any unsatisfied critical constraints in the validation module. The Epsilon flock task provides two attributes: OriginalModel and TargetModel for specifying the original and target models. Epsilon EPL tasks have an optional attribute repeatWhileMatches which can be set to true in order to enable the iterative mode. The maxLoop attribute can be used to set the maximum number of time the iteration can run.

The ANT tool is used to build the system workflow and the integration between the system modules as presented previously in Figure 3-2.

5.2.5 Epsilon Unit Testing

Epsilon provides a unit testing framework (EUnit), similar to JUnit, for testing model management tasks. EUnit test suites are organized as EOL scripts with special annotations (for ex. @test for tests) and Epsilon ANT tasks. EUnit allows tests to be reused with different models. In addition to the default testing assertions (assertEqual, assertTrue, etc.), EUnit provides a task-specific assertion (assertEqualModels) for comparing models. This is helpful when there is a need to compare an expected model with the actual model created in the model management task, such as checking the output of the filtering rules with the expected output. EUnit is used for writing the unit tests for some of the system modules.

5.2.6 EuGENia Graphical Editor

Epsilon EuGENia is a tool that provides the capability to create Graphical Modelling Framework (GMF) editors automatically. The tool automatically generates the graphical configuration models which otherwise need to be created manually. This is based on defining special-purpose annotations (@gmf.node, @gmf.link) on the Emfatic metamodel (textual syntax for Ecore). EuGENia was used to automatically create a simple graphical editor for the BPMN normalised metamodel which enables to visualise the normalised and filtered output models. This reduced some testing efforts by graphically inspecting the intermediate models for correctness rather than checking their textual syntax.
5.3 Summary of Used Technology

Table 5-1 shows a list of Eclipse/Epsilon tools used as part of the system setup and implementation. Table 5-2 summarizes the system modules and the modelling task they perform, with the different Epsilon languages selected for implementation.

Table 5-1: List of Eclipse/Epsilon tools used in the project

<table>
<thead>
<tr>
<th>Eclipse/Epsilon Tool</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPMN modeller</td>
<td>Used to draw input business process models</td>
</tr>
<tr>
<td>UML2 Tools</td>
<td>Used to generate a UML state machine diagram from the output model</td>
</tr>
<tr>
<td>Epsilon EuGENia</td>
<td>Used to create a simple graphical editor for normalised and filtered models</td>
</tr>
<tr>
<td>Epsilon ANT workflow</td>
<td>Used to orchestrate the system modules together into the desired transformation chain.</td>
</tr>
<tr>
<td>Epsilon EUnit</td>
<td>Used to apply unit testing for individual system modules.</td>
</tr>
</tbody>
</table>

Table 5-2: Epsilon languages used for each system module

<table>
<thead>
<tr>
<th>System Module</th>
<th>Transformation/Mapping rules</th>
<th>Model management task</th>
<th>Epsilon language used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation</td>
<td>Input constraints rules; Sec 3.1</td>
<td>Model validation</td>
<td>EVL</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>P1-P3 rules; Sec 4.2.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Petri Net mapping</td>
<td>N1-N4 rules; Sec 4.2.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statechart mapping*</td>
<td>PN2SC initialization rule; Sec 4.2.4</td>
<td>Model migration</td>
<td>Flock</td>
</tr>
<tr>
<td>UML state machine mapping</td>
<td>M1-M5 rules; Sec 4.2.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filtering</td>
<td>R1-R10 rules; Sec 4.2.2</td>
<td>In-place modification</td>
<td>EPL</td>
</tr>
<tr>
<td>Statechart reduction*</td>
<td>AND/OR and T3 rules; Sec 4.2.4</td>
<td>In-place modification</td>
<td>EOL</td>
</tr>
<tr>
<td>Preprocessing helper</td>
<td>-</td>
<td>Model navigation</td>
<td></td>
</tr>
</tbody>
</table>

* Modules reused from PN2SC Epsilon implementation (TTC13).
6 Implementation

This chapter shows the implementation of the system. As the system is expected to execute its modules sequentially following the transformation chain, the modules were developed accordingly from the start towards the end of the workflow. The following sections present an overview of the system flow and give a running example to explain the system implementation.

6.1 System Flowchart

The overall system flow with the modules/rules applied is shown in Figure 6-1.

First, the BPMN validation module checks the input BPMN model is consistent for the transformation. If there are any unsatisfied conditions, the system shows an error message and aborts, else the model is migrated into the evolved metamodel and the preprocessing rules are applied. If any of the object nodes (ON) cannot be normalised, the system fails. After the model is normalised, the filtering rules are applied until no more rules can be applied. The filtered model is then mapped into petri net and then into statechart trees by applying the PN2SC initialization rule. The petri net and statecharts
are reduced by applying the AND/OR rules until the petri net is reduced into a single place, or the reduction rules cannot be applied anymore. In case the latter is true, if the cross-synchronisation rule $T3$ is applicable, the rule is applied and the petri net reduction continues. Otherwise, the system fails. If the petri net is reduced, then the final step maps the statechart into UML state machine. The refactoring rules are applied before the final step if $T3$ rule was applied.

### 6.2 A Running Example

In order to explain the system implementation and functionality, a running test case example is demonstrated in a stepwise fashion. The input and output models at each phase is presented. This is accompanied with some code snippets in order to highlight some of the main features of Epsilon languages. The input business process model used here as an example is shown in Figure 6-2 and it refers to the tax collection scenario described earlier in the first chapter 1.1. The example was used as one of the test cases in the AD2SC solution. A minor change is made here by adding the declaration [not submitted] state in order to explain the preprocessing step. This example and all the other test cases were modelled using Eclipse BPMN2 modeller tool.*

![Diagram of Tax Process Model](image)

* The input for the system was modelled using Eclipse BPMN tool, but all the images in this report are created using Visual Paradigm for image quality and clarity.
6.2.1 Validating the input BPMN model

This Epsilon validation module defines the set of constraints that need to be fulfilled in the input process model. This module does not affect or change the input model; instead, it only checks it against the set conditions.

The code snippet in Figure 6-3 gives an example for two constraints defined using the guard and check keywords: The first one checks that the process contains only the allowed elements types that can be handled by the transformation. The second constraint checks that there exists exactly one stateful data object. This is true for the input model given where there is one data object (declaration) with seven different states.

If any of the constraints is violated, the program will abort showing an informative message to the user such as the ones defined in the message block shown in the code snippet. Since the tax process model passes all the validation conditions, it is passed into the next stage.

```javascript
constraint MustHaveOnlyAllowedNodes {
    guard: self.satisfies("MustExistOneProcess")
    check {
        var nonAllowedElements = self.flowElements.reject(n => n.isKindOf(Event) or n.isTypeOf(ExclusiveGateway)
            or n.isTypeOf(ParallelGateway) or n.isTypeOf(Task) or n.isTypeOf(SequenceFlow)
            or n.isTypeOf(DataObject) or n.isTypeOf(DataObjectReference));
        return nonAllowedElements.size = 0;
    }
    message : "There must be only elements of type [Event, Task, Exclusive Gateway, Parallel Gateway, " +
        "SequenceFlow, Data Object, Data Object Reference] "
        + "but found " + nonAllowedElements
}

constraint MustHaveOneStatefulDataObject {
    guard: self.satisfies("MustExistOneProcess")
    check {
        var numberOfStatefulData = self.getStatefulDataObjects().size();
        return numberOfStatefulData =1;
    }
    message : "There must be one stateful data object but there are " + numberOfStatefulData
}
```

Figure 6-3: An example of two validation rules as part of the EVL validation module

6.2.2 Preprocessing the input process model

This flock migration module produces a process model in the normal form. The input process model conforming to the original BPMN is migrated to a simple model conforming to the BPMN normalised metamodel. Following this migration, the data object and all its data object references are migrated into object nodes with same name
Data output and input associations are also mapped into object flows. All other elements (tasks, events, gateways, sequence flows) are copied into the evolved metamodel. The code fragment in Figure 6-4 shows how data input associations are migrated into object flows. The corresponding type in the migrated metamodel is specified using the `retype` keyword and the `migrate` block specifies the logic behind the migration. The source and target of the object flow are obtained from the equivalent (using the built-in `equivalent` function) of the source and target of the data association. The `original` and `migrated` two keywords refer to the two instances under migration. The last line adds the migrated element to the containing process.

For example, the `declaration [not submitted]` data object is migrated into an object node with same name and state, and the start event node is copied into the migrated model with the same type. The data output association from the start event to the `declaration [not submitted]` data object will be migrated into an object flow. The source node of this object flow is the migrated (copied) start event, and the target is the migrated `declaration [not submitted]` object node.

```plaintext
retype DataInputAssociation to ObjectFlow
migrate DataInputAssociation
{
    original.println("Orig: DIO ");
    migrated.println("Mig: ObjectFlow ");

    migrated.targetRef = original.getAssociatedFlowNode().equivalent();
    migrated.sourceRef = original.sourceRef.first().equivalent();
    original.sourceRef.first().getContainingProcess().equivalent().flowElements.add(migrated);
}
```

Figure 6-4: Migration rule for DataInputAssociation as part of the preprocessing flock module

After the model is migrated, the preprocessing rules are applied for any matching elements. The input model is kept intact while the normalised output model file is created. Figure 6-5 shows the output at the end of the preprocessing phase. In this case, \( P2(a) \) preprocessing rule is applied, for example, for object `declaration [not submitted]` because it has no outgoing flow, and the target of the sequence flow going from the start event is changed to the object node. As a result of \( P3 \) rule application, `send annual statement` task gets a fork afterwards and `receive annual statement` task gets a join before it.
Figure 6-5: Tax process after the preprocessing (normalisation) step
6.2.3 Filtering the normalised process model

This epsilon pattern module applies the filtering rules to produce a filtered process model out of the normalised one. The filtering rules are expressed as patterns that are matched against the input model elements. Whenever there is a match, the actions specified in the do part are executed. Since the repeatWhileMatchesFound feature is selected, and maxLoops is set to (-1) in the workflow, this will run until no more pattern matches can be found for the model.

```
/ *
/ * R1: If there is an activity or control Node (Not Object Node) with one incoming and outgoing edge, it can be deleted.
/ */

pattern SingleInputOutputNonObjectNode
  flowNode: FlowNode
  guard: not flowNode.isTypeOf(ObjectNode)
       and flowNode.incoming.size() == 1 and flowNode.outgoing.size() == 1
{
  onmatch {
    "": println("R1 Matched.");
  }
  do {
    "": println("R1 do.");
    flowNode.incoming.first().targetRef = flowNode.outgoing.first().targetRef;
    delete flowNode.outgoing;
    delete flowNode;
    "": println("R1 applied.");
  }
}
```

Figure 6-6: Filtering Rule R1 as part of the filtering EPL module

Figure 6-6 shows the implementation of R1 filtering rule: The guard for the FlowNode role specifies that the matched node must not be of type object node, and must have exactly one incoming and outgoing edge. The onmatch block can be used for any actions to be executed immediately upon matching, which is used for nothing significant here. Model changes in the do block are applied after the matching phase finishes. Here, the single incoming edge of the matched flow node will point to its single following node. The matched node and its outgoing edge will be deleted.

When applied to the normalised model in Figure 6-5, we can see that in the first matching run, R1 rule will apply 13 times for all the tasks and R5 rule will apply 6 times for the sequence flows adjacent to all the object nodes (except declaration [not submitted]). After the model is changed, R1 rule will again be applicable for other nodes. The final filtered model is shown in Figure 6-7. To keep track of the output results, the normalised input model file is copied before the filtering changes are applied.
6.2.4 Mapping the filtered model to petri net

This flock migration module acts as a bridge to the PN2SC solution. Figure 6-8 shows the petri net equivalent to the filtered tax process model. N1 mapping rule is applied to all the object nodes, the two events, and the exclusive gateway. N2 rule is applied for the two parallel gateways. The transition added between the two places (not submitted and submitted), for instance, is the result of applying N3 rule.

Figure 6-8: Tax petri net equivalent to the tax filtered process model

Figure 6-9 shows the implementation for the map between the filtered model elements and the petri net elements, to support the traceability link functionality needed. The map will contain all the (original, migrated) element pairs at the end of the migration. For example, a record with the filtered model start event as a key with startEvent_1 petri net place as its value will be stored in the map.
6.2.5 Creating the hierarchical statechart

This part of the transformation has been reused from the TTC13 petri net to statechart Epsilon implementation [27]. The reused Epsilon implementation consists of an initialization flock migration module which maps petri net models into statechart hierarchy models, and a number of other EOL modules which implements the (AND/OR) reduction rules. The implementation is extended by adding T3 rule for handling cross-synchronisation situations, and the refactoring rules needed after the cross-synchronisation rule is applied. However, a final postprocessing step is still needed which unfortunately was not finished in time. The trace map for the (original) petri net and (migrated) statechart elements is created at the end of the migration script; similar to the one created when mapping the filtered model to petri net. The use of these two maps as a traceability function is shown in the next step.

Figure 6-10 shows the hierarchical statechart the algorithm generates from the petri net in Figure 6-8. Initially, all the places are mapped into BASIC nodes (shown as inner small boxes with no inner nodes), and all transitions are mapped into hyperedges (shown as petri net transitions). Looking at the petri net diagram, we can see that the OR rule can be applied at the beginning to any of the transitions except the fork and join transitions. The AND rule can be applied only after the three petri net places processed, assessed, and returned are reduced into one place by applying the OR rule twice. The equivalence of this reduction in the statechart tree is the OR node (shown as a dotted box) which contains the three BASIC nodes; named as the nodes names concatenation. After the reduction process finishes, the petri net has only one place and no transitions, and the statechart in Figure 6-10 is generated. The output statechart hierarchy contains three OR nodes and two AND nodes (shown as boxes that contain the OR nodes). The top AND node (named AND) is always produced as the root node. The inner AND node which contains the two OR nodes is a composite state which represents concurrency.
6.2.6 Mapping to UML state machine

This flock migration module is the final step in the transformation chain. The statechart hierarchy model produced in the previous stage is mapped into UML2 state machine. Figure 6-11 shows part of $M3$ migration rule implementation. Here, the migrated type of BASIC elements can be either *FinalState* (first retype statement) or *Pseudostate* (second retype statement). This is specified using the *pm_equivalent()* traceability function implemented. For example, all object nodes, events and exclusive gateways in the filtered model in Figure 6-7 were mapped into BASIC nodes in the statechart in Figure 6-10. The trace function, for instance, will return the exclusive gateway in the tax filtered model as the source for the BASIC state named *Exclusive Gateway 1*, therefore applying the second migration rule shown in the code. The rule sets the type of the UML element as *choice* and the containing element as the region equivalent to the parent of the OR node in the statechart. This way, all the elements types and nesting is preserved in the final UML output.
The final output model produced is shown in Figure 6-12. We can see that the behaviour of the UML state machine output is consistent with the data represented in the tax business process in Figure 6-2. For example, state `accepted` is concurrent with the three states: `processed`, `assessed`, and `returned`. The transformation successfully creates a structure and behaviour equivalent statechart. The output is a valid UML2 model serialized in XMI format. The diagram shown here is automatically generated from the output UML semantics model using Eclipse UML2 Tools. The regions (Region 1, Region 2) and composite states (declaration, Composite State) are manually named.
7 Testing and Evaluation

This chapter shows how the system was tested and evaluated. Unit tests were implemented for the individual modules, and a number of test cases were applied to the system to check its correctness. First, the testing approach applied is briefly explained.

7.1 Testing Approach

Tests can be applied at the unit, integration, or system level. In this project, unit testing was applied to test that individual modules work independently as expected. Each system module is tested with respect to its functionality specified by the algorithm or mapping rules. Integration testing was applied as the system was developed incrementally. A module is tested if it works individually, and when integrated with the adjacent modules in the chain, and so on. For example, both the normalisation and filtering modules are first tested separately and then when combined in a workflow. For each input test case the output is inspected. The inputs used in this stage are not real test cases like the ones used for the final system testing. The system testing is used to test the end-to-end system where the individual modules are integrated into the complete workflow. This checks that each intermediate output model at one stage is passed and handled correctly by the next module in the chain, and the desired output model is generated at the end. The system testing is applied by using a number of different test cases.

7.2 Unit Tests

Epsilon EUnit testing framework was used to write the unit tests for the system. The unit tests suites test the individual system modules independently. The basis of the written unit tests is to validate the implementation of the different rules (normalisation, filtering, mapping) by comparing the test output result generated by the system with the expected result from manually applying the rules. This is achieved with the assistance of the assertEqualModels(result, expected) task-specific assertion for comparing models, which compares two different models for equality based on their elements and attributes. EPL provides a particular model, PatternMatchModel, at the end of running EPL modules which have the matched elements as instances. This was used for the
filtering module EUnit to check when a specific pattern (filtering rule) has been matched. Similarly, EVL provides access to `EVLUnsatisfiedConstraint` instances from a validation module result, which was used to check for the unsatisfied constraints in the BPMN validation module. No unit tests were written for the PN2SC reused implementation. Figure 7-1 code snippet shows an EUnit test which checks that after two runs of the filtering module, R2 filtering rule was matched once (first assertion) and the output result matches the expected result given (second assertion).

```java
@test
operation R2() {
    applyFilteringRules("R2");
    EPL.getMatches().size().println();
    applyFilteringRules("R2");
    EPL.getMatches().size().println();
    var R2matched = EPL!MergeDivergingExclusiveGateway.all;
    assertEquals(1, R2matched.size);
    assertEqualModels("R2","R2_expected");
}
```

**Figure 7-1: EUnit test for R2 rule in the filtering module**

7.3 Test Cases

In order to test the complete system workflow, a set of test cases, shown in Table 7-1, were applied. All the test cases were selected from the set of input test cases used in the AD2SC solution, except test case T1 which was chosen from PLANT paper. The results obtained from the AD2SC solution are used as a benchmark for validating the BP2SC system results. All the input cases were modelled using the Eclipse BPMN Modeller tool. The output is valid a UML model file which can be imported in Eclipse-based tools (tested as well in MagicDraw tool). In order to visualise the output diagram, such as the one in Figure 6-12, Eclipse UML2 Tools was used. The tests were executed on Eclipse IDE running on a Windows 8, Intel core 2 Duo machine. Although the system performance is not the main issue in this project, the test results showed that it takes only few seconds (1-3) for the complete transformation (as reported by Eclipse ant tool).

The input and output for test case T2 (tax process) were shown in the previous chapter. Test cases T1, T3, T4, T5 input and output are shown in Appendix A: Test Cases. T1 contains no parallel gateways, and the output has no parallel states. Both T3 and T4 inputs contain unbalanced forks/joins, and the output results have nested parallel states. T4 input case also had two object nodes with same state name `[bike ready]` which does not satisfy the BPMN validation constraints as the state names are required to be
unique. Therefore, one of the two states was manually renamed before applying the test case. Other test cases are shown and discussed in the next evaluation section.

### Table 7-1: Test cases applied for the system and their matching with the expected result

<table>
<thead>
<tr>
<th>Test case</th>
<th># of flow nodes</th>
<th># of object states</th>
<th>Results match expected output?</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 Materials order</td>
<td>15</td>
<td>6</td>
<td>Yes</td>
<td>No parallelism in input and output</td>
</tr>
<tr>
<td>T2 Tax collection</td>
<td>19</td>
<td>7</td>
<td>Yes</td>
<td>Balanced forks/joins</td>
</tr>
<tr>
<td>T3 Claim settlement</td>
<td>19</td>
<td>11</td>
<td>Yes</td>
<td>Unbalanced forks/joins</td>
</tr>
<tr>
<td>T4 Bike shop</td>
<td>28</td>
<td>12</td>
<td>Yes</td>
<td>Object state names had to be made unique</td>
</tr>
<tr>
<td>T5 Money withdraw</td>
<td>11</td>
<td>3</td>
<td>Yes</td>
<td>Cross-synchronisation removed by filtering rules</td>
</tr>
<tr>
<td>T6 Media store</td>
<td>16</td>
<td>3</td>
<td>Yes</td>
<td>Cross-synchronisation removed by filtering rules</td>
</tr>
<tr>
<td>T7 Production</td>
<td>12</td>
<td>10</td>
<td>No</td>
<td>Cross-synchronisation not removed by filtering rules. Unmatched due to refactoring rules not being finalized</td>
</tr>
</tbody>
</table>

### 7.4 Results Evaluation

The system was evaluated based on the results of the used test cases. The output results were compared against the expected results for the test cases. This section discusses the three system results categories as follows: **correct** output, **inconsistent** output due to implementation incompleteness, or **failed** situations due to algorithm incompleteness. A classification of the input based on its equivalent (filtered) petri net is shown. In addition to that, some practical issues are also discussed.

#### 7.4.1 Theoretical Evaluation

Based on the fact that the PN2SC algorithm is proved to be correct and complete for a subclass of petri nets [7], the same output pattern can be noticed from the test cases results. The system successfully generates a state machine model for test cases T1 to T6 where their equivalent filtered petri nets are **safe** and have **no cross-synchronisation**. Test cases T5 and T6 originally contain cross-synchronisation that the filtering rules
managed to remove. Therefore, the system generates an output without the need to apply the cross-synchronisation extension rules. This shows the importance of the transformation processes order; having the filtering phase before creating the hierarchy statechart ensures irrelevant nodes are removed [5]. The UML state machine generated is consistent with the data behaviour in the corresponding input BPMN process model, and preserves its structure. Figure 7-2 shows test case T6 where the input process model has cross-synchronisation (confirm delivery task needs to wait for both deliver and confirm checkout tasks to finish, while pay task does not need to wait). The output is the three states in a sequence.

Test case T7, shown in Figure 7-3, contains a cross-synchronisation which the filtering rules do not remove. Therefore, the cross-synchronisation rules (T3) are applied. The system output for T7, shown in Figure 7-5, is invalid and has inconsistent behaviour with the input model. To explain this, if we examine the output in Figure 7-4 before applying the cross-synchronisation rules, we can see that T3 rules can be applied to either of the three OR nodes [unchecked checked][costs unknown costs calculated][planned produced], which are both between a fork and a join.
Ideally, to have a structure-similar output, the second OR node must be handled by the T3 rules. However, the first OR node is selected by the system. In any case, the output has invalid/inconsistent statechart behaviour; there exist a transition which partially enters or partially leaves a region, similar to the transition from *checked* state in Figure 7-5. The system can be extended by finalizing the refactoring rules specified in section 4.2.4 in order to get a behaviour-equivalent output. End users intervention might be needed during execution to choose which node to be processed by T3 rules [5].
As mentioned in chapter 4, there are situations where the algorithm fails and the system cannot generate a statechart output. Figure 7-6 shows three examples for such cases taken from [7]. Examples (a) and (b) show two petri nets where there is no equivalent statechart with same behaviour and structure. The first one (a) is an example of an unsafe petri net; place $p4$ can be reached through transition $t2$ and $t3$. This means that, for example, places $p3$ and $p4$ can both be active in a possible marking. For statecharts, if one of the parallel states $p2, p3$ is left, the other will also be left; therefore, the same behaviour is not valid. The second example (b) is a safe petri net where transition $t3$ partially enters the concurrent places $p2, p3$; hence in one case they can be parallel (if $t1$ fires) while in the other case they are not (if $t3$ fires). Again, in statecharts if a transition enters a concurrent state partially, the other parallel states are also entered (their default states).

Figure (c) is an example of a safe petri net where the algorithm fails to generate the structure/behaviour equivalent statechart shown on the right-hand side of the figure. Here, either states $p1,p2$ or states $p3,p2$ can be in parallel. The OR node $O1$ contains two unconnected nodes $p1,p3$. Such class of statecharts where there are unconnected states in one OR state (region) is not translatable by the algorithm. Eshuis argues that this class of petri nets is not often found in practice [7].
Figure 7-6: (a) unsafe petri net and (b) safe petri net with no structure/behaviour equivalent statechart; (c) safe petri net with its equivalent statechart which the algorithm cannot generate [7].

Table 7-2 summarises the classification of input petri nets and its translatability using the transformation approach. The petri nets represent the filtered input BPMN process model (its data states behaviour). Formal definition of the class of input where the algorithm is complete and correct is provided in [5, 7].

<table>
<thead>
<tr>
<th>Petri net class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsafe petri nets</td>
<td>Unsafe petri nets, such as Figure 7-6(a), do not have structure/behaviour equivalent statechart.</td>
</tr>
<tr>
<td>safe petri nets with no equivalent structure-</td>
<td>Includes petri nets with cross-synchronisation such as Figure 7-3. The extension rules can generate a behaviour non-structure preserving statechart which uses events, actions/guards. Includes also petri nets where a transition partially enters a parallel AND state, such as Figure 7-6 (b).</td>
</tr>
<tr>
<td>preserving statechart</td>
<td></td>
</tr>
<tr>
<td>safe petri nets with an equivalent structure-</td>
<td>Class of statecharts which have unconnected BASIC states within an OR state, such as Figure 7-6 (c).</td>
</tr>
<tr>
<td>preserving statechart, where the algorithm fails</td>
<td></td>
</tr>
<tr>
<td>safe petri nets where the algorithm is complete</td>
<td>Safe petri nets where the algorithm is proved to generate a structure and behaviour preserving statechart. This is formally defined in [7].</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finally, the preprocessing rules shown in section 4.2.1 are from a “UML perspective” and the BPMN input is expected to have similar behaviour for the preprocessing rules to be valid [37]. The example in Figure 7-7 is given to show some of the BPMN data semantics differences. The example contains some data “anomalies”, as described in [38], but is valid to explain these differences.

Object node _claim[accepted]_ has two outgoing flows to the two parallel tasks. If we consider the “UML perspective”, this indicates a choice; which means only one of the tasks can access the object and the other task will wait indefinitely. If, for example, we consider both tasks can access the same copy, and the _file and update history_ task executes and writes to the data before the _pay_ task starts, this again could cause a deadlock [38]. If we assume each of the tasks gets a copy of the data, then, in this case, the preprocessing rules used do not reflect this behaviour. In a personal communication (email) with Eshuis, he mentioned for a similar example “… If the tasks are really in parallel, I would expect from the UML perspective that they have different input object nodes (as in all examples in the paper). With a BPMN perspective, you can argue that your view is right, but then you no longer have a token-semantics but a copy semantics for objects; … To create a statechart in that case, you would have to change the mapping rules.” [37]. This can show a possible difference for objects with multiple flows.

*Figure 7-7: Claim example with some BPMN semantics differences, from [38]*
In addition, from the interpretation of the example given in [38], the close task requires the claim object to be either in state claim[filed] or claim[rejected], but not both. BPMN supports alternative data semantics for tasks by using input/output sets, without providing a different graphical notation. The system fails for the given example. The preprocessing rules used here add a join before the close task meaning that both data are required, which is not the desired behaviour.

Unfortunately, these issues are not covered here and needs further study. A recent study [39] defined a formalization of BPMN data semantics by mapping into petri net. The mapping rules are complicated as it takes into account optional and alternative data. This project can be extended by applying these rules in combination with the AD2SC approach.

### 7.4.2 Practical Evaluation

Regarding the practical side of the transformation, the implementation of the transformation approach was feasible using the current Eclipse modelling tools. The transformation under implementation here is considered complex enough as it consists of a chain of different model management tasks.

The Epsilon platform with its provided set of languages and tools supports handling these modelling tasks, as was explained in chapter 5. Although I have never involved before with hybrid languages, Epsilon languages syntax can be considered as both understandable and expressive. Though, I had experienced that it can sometimes be tricky to specify the pattern rules in EPL\(^1\) for someone who is not familiar with its execution semantics. The roles used to define the pattern can help writing a concise rule, however, it can sometimes lead to runtime errors if, for instance, one matched rule deletes an element which is a role instance in another matched rule.

One difficulty realized at the end of the system implementation was creating an executable file to run the system as a standalone (outside Eclipse IDE). Out of the discussion in the Epsilon online forum, it appeared that there are some issues related to loading the Eclipse UML metamodel from the ANT tool, which causes a runtime error\(^2\). From my little experience, Epsilon is continuously evolving and it has an active and friendly user community.

---

\(^1\) EPL is considered relatively a new language in the set of Epsilon languages

\(^2\) As per its contributor, this is planned to be fixed in the coming Epsilon release.
The input and output models were both created and manipulated using Eclipse-based tools. Since ever, it has been an aim in model-driven technology to achieve compatibility between different modelling tools. Standards such as XMI were mainly introduced for such reason. In order to test the interoperability of the system with other tools, some BPMN input models were created in Visual Paradigm (evaluation copy). Although the tool provides an *Export to BPMN2.0* capability, some differences were realized regarding the data association and data objects states syntax compared to the syntax supported by Eclipse. Few manual changes have to be applied before the input can be passed to the system, which can be considered as a limitation.

On the other hand, the output UML *semantic* model can be imported into Eclipse-based UML tools as well as both MagicDraw and Visual Paradigm tools. In order to inspect the output easily, there was a necessity to visualise the UML statechart model by automatically generating a UML statechart *diagram* out of the semantic model file. This proved to be difficult with many of the available UML modelling tools. In fact, the only tool found capable of this was Eclipse *UML2 Tools*\(^3\). Although UML2 Tools is considered obsolete, the newer Eclipse tool, Papyrus, surprisingly seems not to provide the same diagram generation feature for UML statechart models.

Regarding the effectiveness of the system, we can argue that it obviously saves time that would have been needed to manually model the statechart diagram. The transformation is fully automated and the only manual intervention needed is to label the output UML composite states, regions, and guards, if needed. It would be interesting if the system usability and usefulness can be assessed in real situations by running a pilot study, similar to the one used in PLANT.

\(^3\) The UML version of the output (2.4) need to be changed to (2.3) in order to be imported into tools which does not supports the last XMI version.
8 Conclusion and Future Work

This research has been set to investigate the question posed at the beginning of the study; whether it is feasible to automatically generate a UML state machine model from a BPMN model that implicitly embodies an object life cycle. The findings showed that a proven systematic approach could be used for a subset of the input process models. It also showed that this might fail for another input class, generally due to semantic constraints of UML state machines. This chapter summarises the findings and contributions of the project. It also addresses some areas for future work.

8.1 Contributions

Business process models and object statecharts have their wide use in software development lifecycles. PLANT pattern language identified that scenarios used in software requirements can be represented into four different perspectives, two of which is the process model and object model views. This project has investigated the possibility of incorporating models transformation in requirements modelling to automatically generate a UML statechart from a given BPMN process model. There are two main advantages that can arise from this: first, it can reduce time and effort required to manually generate the object statechart model. Second, the generated output model can help checking the consistency of the process model behaviour early on the development stage. Some tools can even further use the statechart model to generate software code.

To help answer the question posed at the beginning of the project, a detailed background research was initially conducted. This started with an overview about MDE in order to become familiar with the topic, as it was new for me, followed by a detailed study about MDA base standards and the BPMN and UML state machine specifications. It also involved examining the model transformation classifications and technologies available, and considering the tools and languages supported.

The next step was to research the literature for related work to understand what approaches taken in similar projects and to make an informative decision about a suitable methodology that can be applied for this project. It should be highlighted that this was far from easy as, to my knowledge, not many papers addressed similar
transformation from process models to object statecharts. A recent study [5] was found to have achieved so for UML activity diagrams, based on a proved algorithm [7] for translating petri nets into statecharts. Another study [22] defined an approach to create flat statecharts (no parallel states) from UML activity diagrams. In [21], petri nets and “reachability graph” models are used to create the object life cycle, but the input BPMN model can have only XOR gateways, and the statechart output is also not hierarchical. This project followed the AD2SC approach for a subset of BPMN models.

At the same time, the available modelling languages and tools were investigated. Epsilon with its comprehensive toolset was a suitable choice for the transformation. Epsilon Flock was used for tasks which require model update according to changes in its metamodel (normalisation, petri net mapping, UML statechart mapping). Epsilon EPL was used for in-place modification tasks (filtering rules).

Based on the transformation approach and selected technology, the system was designed as shown in chapter 3. A normalised BPMN metamodel was created to simplify model navigation, as well as to define normalised and filtered process models. The system workflow was built to resemble the transformation chain. In the implementation phase, the system modules were implemented incrementally and the system was built around an existing PN2SC Epsilon implementation. Unit testing and a number of test cases were used to evaluate the system and the correctness of the output results. The special situations where the transformation might fail were highlighted in the evaluation section.

We can conclude that the project has achieved its main objectives. The results obtained show that the same approach and rules used in AD2SC solution can be applied to some BPMN process models that have similar semantics as UML activity diagrams, by mapping UML activity diagram elements into their counterpart BPMN elements (data object for object node, data input/output association for object flow). The project also highlighted that BPMN models can have different behaviour with optional and alternative data, as well as data objects with multiple flows. Situations where there might be semantic differences between the two standards can be investigated in the future.
8.2 Challenges and Reflection

This transformation from process models to hierarchical statecharts inherits certain challenges [5]. First, as mentioned in the BPMN OMG specification [4], BPMN is not intended as a data model. Instead, the BPMN model implicitly shows the object life cycle inside the business workflow. The goal of the transformation is to make this explicit by generating a hierarchical statechart (composite/nested states) from the process model. Concurrency of data states is represented using such composite states and regions in a hierarchy fashion, which does not exist in the flat process model. In the BPMN model, parallel tasks with data input/output or single tasks with multiple data input/output can imply the data are concurrent. The algorithm defined in [7] achieves this by translating a subclass of safe petri nets (which can be a target for many workflow models) into hierarchical statecharts.

Another challenge is the semantics expressivity for the two models. The UML statechart metamodel is considered limited compared to the large set of elements in BPMN. Some elements do not have a counterpart or are not expressible in statecharts, such as inclusive gateways [37]. In addition, there are irrelevant nodes in the input process model that does not affect the object life cycle. These nodes need to be removed before constructing the statechart. This is addressed through the filtering rules defined in [5].

Finally, some BPMN models may have “cross-synchronisation”. This behaviour has no equivalent “structure-preserving” statechart. The extension rules in [5] address this challenge by constructing a “behaviour-preserving” statechart.

Having a “structure-preserving” transformation is desirable and interesting because it allows easily to relate and reason between the source and target models. However, we can see when loosening this feature, a “behaviour-preserving” transformation could be achieved for some source models where there is no structure equivalent target model; such as with the “cross-synchronisation” case.

Yet, the transformation cannot generate statecharts for every input process model as it may fail in some cases as was explained in the previous chapter.

From a personal point of view, the challenge of the project was mostly in the theory part. The system here is different from the kind of application systems where the key is on the design and implementation. The rules and algorithm used in this project are
complex, and it would need a substantial background knowledge about the semantics of
the models under study. I think what adds to the difficulty the fact that the BPMN data
semantics in the specification document is not formally or clearly defined, and it was
difficult to find many definitions for this in the available literature, especially for the
latest standard BPMN 2.0.

In model transformations, there could be different ways to achieve the transformation
from one source model to another target model. For some, a direct mapping could be
feasible while, for others, mapping to another intermediate model could be a logical
way. Petri nets are a common option for this because they have a formal definition and
many studies have been applied on them. If the mapping from BPMN data into petri net
is clearly defined, it can be seamless to use the different available and proved petri net
algorithms, such as the PN2SC algorithm.

It is worth mentioning that another direction for the project solution would have been to
transform BPMN to UML activity diagrams (BPMN2AD) and reuse the AD2SC
implementation. BPMN2AD is a common transformation because the two standards
have many similarities. Studies as [24] defined some transformation rules for BPMN
(1.0). Someone might argue this would have been a better option. However, I decided to
implement the BP2SC as my personal project goal was to learn as much as possible and
do as much implementation as I can. I have learned different theoretical and practical
aspects out of this project as well as gained some experience with different Eclipse
tools. I hope that the system implementation in Epsilon would be a useful contribution
for others who want to learn about the new Epsilon languages such as EPL and use it in
similar modelling tasks.

8.3 Future Work

The project has successfully managed to implement the algorithm and main rules used
to generate UML statecharts for a large set of BPMN process models. The cross-
synchronisation extension rules were also implemented to cover input models which
exhibit this behaviour. The refactoring steps required to repair such class of models are
implemented, but a final postprocessing step need to be added in this case. This can be
an extension to the system, with the addition of user intervention capability to allow
selecting the node to be processed with cross-synchronisation rule. In addition, actions,
events, and guards which trigger and control state transitions can be added into the
generated state machine output.
The scope of BPMN models covered in this project resembles UML activity diagrams. It would be useful to investigate situations where the BPMN semantic can be different. Data semantics in BPMN support optional and alternative data where different mapping seemed to be required, such as in [39]. The set of BPMN elements covered by the system transformation was also limited to normal tasks, parallel and exclusive gateways, and start and end events. Although those are considered the most commonly used elements based on many surveys, it would still be useful to investigate how other BPMN elements such as sub-processes or loop activities can affect the approach. It would also be interesting to see how other approaches could be introduced for cases not handled by the PN2SC algorithm.

To make the application more usable and user-friendly, the system can be extended, for instance, as a plug-in feature for the Eclipse BPMN modeller tool. This can enable the analyst/end-user to draw the BPMN model and have an option within the same modelling tool to generate the statechart model. The main aim of the project was to create statecharts from BPMN process models assuming the process model is correct, which is not always the case. It would be useful if the system can detect and try to fix or highlight any data “anomalies” or inconsistencies in the process. Features as models comparison and code generation from statechart models can also be incorporated into the tool to have a comprehensive modelling tool. A systematic usability study can also be conducted in order to check the usefulness of the tool in real world scenarios.

In the wider context of PLANT, this project addressed generating OTM models from POM. Further work can similarly look into the other two perspectives, AOM and GOM, and whether any of the two models can be automatically generated from the other perspectives. Another goal of PLANT is trying to generate the POM model automatically from textual scenarios using approaches such as natural language processing (NLP).
9 References


Appendix A: Test Cases

Figure A-1: Test case (T1): material order
Appendix A: Test Cases

Figure A-2: Test Case (T3): claim settlement
Figure A-3: Test Case (T4): bike shop
Figure A-4: Test Case (T5): money withdraw
Appendix B: Sample Code

```java
post edge {
    for (original in PM!Edge.all) {
        var srcNode = original.sourceRef.equivalent();
        var tgtNode = original.targetRef.equivalent();
        if (srcNode.isTypeOf(PM!Place)) {
            if (tgtNode.isTypeOf(PM!Place)) {
                var postt = new PM!Transition;
                postt.name = srcNode.name + tgtNode.name;
                original.getContainingProcess().equivalent().transitions.add(postt);
                srcNode.postt.add(postt);
            } else {
                srcNode.postt.add(tgtNode);
            }
        } else {
            if (tgtNode.isTypeOf(PM!Transition)) {
                -- add Dummy Place (This should never happen!!)
                var postp = new PM!Place;
                postp.name = srcNode.name + tgtNode.name;
                original.getContainingProcess().equivalent().places.add(postp);
                srcNode.postp.add(postp);
            } else {
                srcNode.postp.add(tgtNode);
            }
        }
        original.println("Org Seq: ");
        --migrated.println("Mig Seq: ");
    }
}
```

Figure B-1: Rules M3, M4 for mapping filtered process model to Petri net

```java
@cached
operation SC!State pm_equivalent() : PM!FlowNode {
    return self.get_src_map();
    --return PM!FlowNode.all.selectOne(s | s.name == self.name);
}

@cached
operation SC!State get_src_map() : PM!FlowNode {
    self.println();
    --var pn_sc_map = pn_link_sc();
    --var pm pn_map = pm_link pn();
    var pn;
    for (key in pn_sc_map.keySet())
        if (pn_sc_map.get(key) == self)
            pn = key;
    for (key in pm pn_map.keySet())
        if (pm pn_map.get(key) == pn)
            return key;
    return null;
}
```

Figure B-2: Traceability function for mapping filtered model elements to statechart elements
<!-- Invoke Flock to migrate petri net into state chart hierarchy, and then apply reduction rules -->
<target name="pn2sc_initialization" depends="mapping2petrinet">
  <epsilon.emf.loadModel name="sc_initialization" read="false" store="true"
    metamodelfile="netamodels/StateCharts.ecore" modelfile="${outputfile_sc_after_reduction}"
  />
</epsilon.emf.loadModel>

<epsilon.flock src="src/pn2sc/Initialisation/Petrinet2StateChart.mig" originalModel="PN" migratedModel="SC">
  <model ref="petrinet" as="PN"></model>
  <model ref="sc_initialization" as="SC"></model>
  <exports ref="pn_sc_map" optional="false"></exports>
</epsilon.flock>

<!-- Store the migrated SC file, just for checking -->
<epsilon.storeModel model="sc_initialization" target="${outputfile_sc}"></epsilon.storeModel>
</target>

<target name="statechart_reduction" depends="pn2sc_initialization">
  <epsilon.ecl src="src/pn2sc/reduction/reduce_state_chart.ecl">
    <model ref="petrinet" as="PN"></model>
    <model ref="sc_initialization" as="SC"></model>
  </epsilon.ecl>
</target>

<target name="uml_statechart_mapping" depends="statechart_reduction">
  <epsilon.emf.loadModel name="uml_statechart" read="false" store="true"
    metamodelfile="netamodels/UML.ecore" modelfile="${outputfile_umlSc}"
  />
</epsilon.emf.loadModel>

<epsilon.flock src="src/postprocessing/sc2umlStatechart.mig" originalModel="SC" migratedModel="UML">
  <model ref="sc_initialization" as="SC"></model>
  <model ref="uml_statechart" as="UML"></model>
  <model ref="petrinet" as="PN"></model>
  <model ref="preprocessed" as="PN"></model>
  <uses ref="pm_uml_map" optional="false"></uses>
  <uses ref="pm_sc_map" optional="false"></uses>
</epsilon.flock>
</target>

<!-- END of Synthesizing Phase -->

Figure B-3: Part of the system workflow (ANT build file)