Exploring Model Driven Engineering from Behavioural Models

Muideen Adesola AJAGBE

MSc by Research

UNIVERSITY OF YORK
Computer Science
April, 2017

Abstract

Model Driven Engineering (MDE) is an approach in software engineering that promotes the use of models as first class artefacts. It enables improved productivity and consistency through the reuse of models to generate other necessary artefacts such as working code and textual documents. In MDE, modelling tools and model management operations are deployed to explore a system under development. Through model management operations (e.g. model transformation, validation and comparison), many system can be automated. For example, model-to-model transformation (M2M) are used to develop other model artefacts from the source models.

EAE system is used as our case study and code is normally hand-crafted for its simulation. However, this hand-crafted code might contain bugs as modellers are prone to error. The need to avoid this error gives rise to using MDE practices on the system and it provides an avenue to generating OO code. This code is meant to be fitness-for-purpose thereby leading to support for EAE simulation. As MDE is applied to this system, the transformation of EAE model mostly represented in behaviour diagrams to structure model is conceived.

This thesis explores EAE's models by transforming its behaviour diagrams to structural diagram where OO code can be generated. Our techniques deploy two approaches towards the EAE's domain model. The structural diagram produces by these different approaches leads to OO code generation which will be used as a guide to EAE's simulation. This process is supported by the results of transformation rule used in the thesis thereby reducing loss of information during the transformation process.

For my parents and siblings.

Contents

List of Figures

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Dr. Fiona Polack and Professor Richard Paige for their incredible support and tutelage throughout the duration of this project. I am also very grateful to my assessor, Dr Simon OKeefe, for providing me with valuable feedbacks. I am highly indebted to Dr. Dimitris Kolovos and Athanasios Zolotas for their technical support.

A big thanks and appreciation goes to my amazing parents, Adetunji and Remi, my brother, Saheed, and darling sisters, Kudirat and Medinat for the endless love and moral support every day. Special thanks to Isobel Atkinson, Jamal Hallam and Alfa Ryano for making my York experience worthwhile.

Finally, All praise be to Almighty Allah (SWT), Lord of the Universe, the Most Gracious and the Most Merciful for the successful completion of my degree.

Author's Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Chapter 1

Introduction

This report concerns the application of model driven engineering (MDE) techniques and tools to support creation of simulator code. The work is part of an initiative to provide software engineering support for the modeling and development approach used in immune systems simulation by the York Computational Immunology Lab (YCIL)[1].

Model-Driven Engineering (MDE) is a software development approach that seeks to use models as first-class engineering artefacts in the software development lifecycle. It has been stated that MDE improves systems productivity, maintainability and reusability by using models to manipulate systems [4]. MDE techniques and tools such as EMF and Epsilon are used to enable large model manipulations such as validation and transformation.

More so, if detailed UML models exist and are valid with respect to the domain, software engineering support can be used to generate code that is valid with respect to the models. This support is a step towards developing model simulations.

Read's thesis [8] is a typical example of the sort of simulation design that we are interested in. Rather than working from a class model, it uses UML behaviour models in design. This report uses the design models and text information from [8] to propose approaches using MDE to support simulation development.

1.1 Overview of Model and Model Driven Engineering

Model is an abstraction used in understanding or representing a concept. In MDE, models are represented in well-defined modeling languages. The definition of the modeling language is specified by a MDE metamodel. A good MDE model is said to conform to its metamodel. A metamodel is itself a model, and is usually defined in a metamodelling language such as MOF or Ecore.

[1] York Computation Immunology Group: https://www.york.ac.uk/computationalimmunology/publications/

By defining and instantiating metamodels, software engineers are able to develop well-defined domain specific languages (DSL), whilst model management tools that operate on well-defined languages provide support for construction, manipulation and validation of models.

MDE-supported development typically uses a structure diagram (a UML class diagram or similar) as a basis for generating Object Oriented (OO) code. However, MDE has not generally been used to support systems modelled primarily through their behaviour - a major aspect of our work.

1.2 Research simulations: YCIL immune systems studies

In this project, work from existing case studies created by the YCIL group are explored. The YCIL approach uses behavioural models such as activity diagrams and state diagrams to create a domain model which biologists can understand and validate. These diagrams are typically created using UML with slight variations. The models are currently used as a guide to implementation, but the code is not directly derived from the models.

The work in this report is based on the study of a murine autoimmune disease called experimental autoimmune encephalomyelitis (EAE) [8]. Our work is focused on the EAE domain and how the domain models can be simulated and modelled in order to study the emergent behaviours observed in them. More so, [8] uses behavioural diagrams such as activity and state diagram to represent biological components of EAE.

Read [8] develops his simulation using the CoSMoS [2] approach, where the domain, domain model, platform model, simulation platform and result model are developed to give a system that is fit for the specified simulation purpose [12]. The benefits of using a CoSMoS-like approach on case studies such as this one include:

- The capturing and merging of data from different sources which can be used to developing a system-level synopsis of the system behaviour.
- Simulation provides a platform for the formulation and evaluation of hypotheses concerning the complex systems' operation.

1.3 Motivation and Research Hypothesis

The YCIL case studies [8, 13, 14] use a common set of behavioural modelling approaches (based on UML activity diagram and state diagrams) but then handcraft code for simulation. The act of hand-crafting code is error prone, since it is difficult to provide validation that the models have been correctly interpreted in the code.

[2] York Centre for Complex Systems Analysis :https://www.york.ac.uk/yccsa/research/cosmos/

MDE approaches may provide a way to rigorously transform models to code which is more fit-for-purpose.

This project investigates the hypothesis that metamodelling and model transformation can be used to provide an automatable approach to creating simulation code from the behavioural models used by YCIL projects.

To investigate this hypothesis, the report presents two approaches to automation. The first (naive) approach builds on published work that shows how a class model can be derived from an activity diagram alone. The second approach, which is more relevant to the large, complex systems addressed by YCIL, uses all the domain models to create a class model suitable for OO code generation.

Finally, the work presents a detailed review of behavioural transformation of UML behaviour diagrams to structure diagrams. The effective current-state-of-the-art tools and techniques used in MDE are discussed, showing their benefits and limitations.

1.4 Thesis Structure

Chapter 2 discusses the detailed overview of Model Driven Engineering, including the concepts and terminologies of MDE. Models, metamodels and modelling language are discussed in relation to our work. Also, different model management operations are reviewed. MDE tools are discussed with emphasis on EMF, Papyrus and Epsilon as the primary tools used in this work. The challenges and benefits of MDE are also briefly reviewed.

Chapter 3 presents Read's EAE domain analysis by providing an overview of EAE immunology disease as a complex systems with the sections presenting the low level, top level of the systems model. In this chapter, EAE diagrams from Read's thesis [8] are discussed. Also, the domain model of EAE is presented as the system-level overview, system and modelling perspective and system-level dynamics where diagram representing each level are discussed.

Chapter 4 presents an analysis and hypothesis of our research. The research background, hypothesis and scope is discussed in this chapter. Also, the research methodology in form of analysis, design and implementation is reviewed. A brief overview of the research modelling approach is also motivated.

Chapter 5 details the naive (first) modelling approach. The input model (activity diagram) used and the transformation strategy applied towards a structure class diagram are discussed. The automation of the modelling process and the tool used are discussed. The code generated is presented. Also, an evaluation and critique of this approach is given.

Chapter 6 discusses the second approach. It presents the input model (state machine diagram with other information) and its transformation strategies. Also, the output model and a target output model are reviewed. The MDE practices explored using EMF are presented. Also, model management operations are discussed

under EOL, EVL and ETL. Finally, the evaluation and critique of this approach is discussed.

Chapter 7 concludes by summarizing the approach and the exploration of MDE through EAE model diagram in this thesis. It provides direction for further work to support simulation with the generated object-oriented structure model.

Chapter 2

Background: Model Driven Engineering

This chapter introduces concepts of Model Driven Engineering (MDE). A detailed context of MDE is needed in order to understand model validation and transformation. A model is an abstraction of a system; models are the necessary artefacts needed to understand the detail of a system under study.

The use of MDE practices for modelling Read's [8] EAE system is justified by Greenfield [15]: *"The software industry remains reliant on the craftsmanship of skilled individuals engaged in labour intensive manual tasks. However, growing pressure to reduce cost and time to market and to improve software quality may catalyse a transition to more automated methods. We look at how the software industry may be industrialized, and we describe technologies that might be used to support this vision. We suggest that the current software development paradigm, based on object orientation, may have reached the point of exhaustion, and we propose a model for its successor"* [15].

In MDE, a principled process to software engineering where models are used throughout the engineering process is advocated. The concept and terminology of MDE are discussed to buttress this point. In this chapter, we review the MDE guidelines and the MDA standard to guide through effective modelling approach. Model management is also discussed as a step to exploring MDE. Also MDE tools available to support the development of our work are discussed. In conclusion, MDE benefits and its challenges are presented so as to fully utilize MDE to our advantage in this thesis.

2.1 MDE Concepts and Terminologies

MDE is used for the construction and manipulation of artefacts such as code and documentation [16]. It involves the use of different artefacts like model, metamodels and model management operations. MDE deploys model management operations on models, metamodels and other artefacts so as to manipulate models effectively.

This section details the activities involved in MDE.

2.1.1 Models and Metamodels

MDE as a contemporary approach to software development allows model to be used as a first-class artefacts during development processes. "Model provides the representation of simplified system in a well-defined language" [17]. According to [18], these models are "the descriptions of phenomena of interest" which can be expressed in general-purpose or specific languages. Also, a "model allows concepts to be shown, wherein, an abstraction of irrelevant details from reality is made possible" [19]. Thus, a model is an abstract representation of a system in a domain and it is created by software engineers so as to capture important details of such system.

There are many definitions used in expressing the meaning of model; it is essentially the abstract representation of artefacts from the real-world [20]. The real world view that a model represents may be called the model's domain or the system of interest. The exploration of this domain may use automatable model tools for analysis, validation, transformation and other intended purposes.

In MDE, description of interests can be easily manipulated by powerful automated model management tools [21]. Model management processes allow model manipulation to be automated if the models are defined in modeling languages i.e. the models conform to metamodels [22].

Metamodels describe the structure, concepts and well-formedness rules pertinent to a group of models [21]. The metamodels are models themselves and they conform to a metametamodel [21]. Figure 2.1 illustrastes model which is an instance of a metamodel and these models is expressed using defined languages. The instantiation or development of a metamodel enables model operations such as refactoring, transformation and comparisons to be managed and applied to models [23]. Metamodel creates a type-system for models which allow their properties to be substantiated and validated. Also, metamodel aids model interchange and, therefore, interoperability between modelling artefacts and tools is made possible. This section presents important areas of MDE, model management approaches, and how they relate to our work.

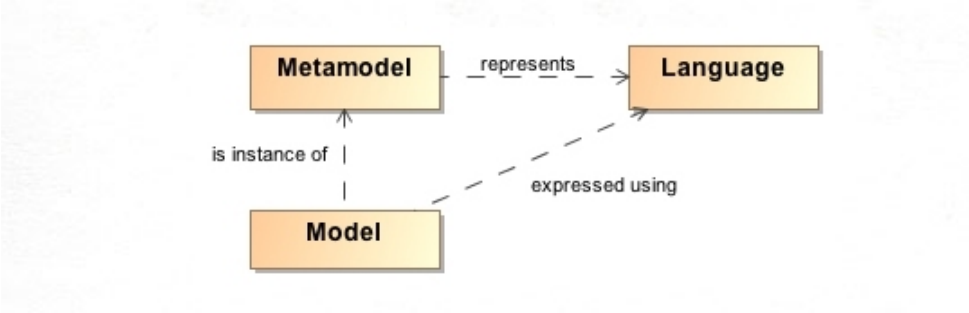


Figure 2.1: An illustration of the relationship between model, metamodel and language from [1].

2.1.2 Modelling Languages

Models can be grouped into structured or unstructured, depending on their conformance to specified rules. Structured models comply to established and well defined rules (e.g. notations) while unstructured models do not conform to any rules. In MDE, with the aid of a modelling language, models are structured [24]. A modelling language contains "syntactic and semantic constraints" deployed to define a group of related model structures [25]. These groups of models adhere to the rigid sets of rules and syntax established in their metamodels [26]. Also, these sets of rules are encoded within a modelling language.

MDE advocates the development of abstract syntax, concrete syntax and semantics in order to establish conformance - a relationship between model and metamodel [27]. Conformance is viewed as a sets of constraints between models and metamodels [28]. A satisfaction of these constraints determines that a model conforms to a metamodel.

A metamodel comprises three categories of constraints:

- **The concrete syntax** represents the modelling concepts of a model (e.g. graphically). For example, a group of boxes connected by lines can represent a model. Concrete syntax give notations for developing models that conform to a metamodelling language thereby facilitating communication. Also, a strict concrete syntax is optimal for machine comsumption (e.g XML Metadata Interchange (XMI) or Unified Modelling Language (UML) [25].)

- **The abstract syntax** describes modelling objects, defined in a metamodelling language, such as classes, packages, interfaces and datatypes. Representing these objects is independent of any concrete syntax. For example, an abstract syntax tree to encodes an abstract syntax of a program when trying to implement a compiler.

- **The semantics** is the meaning of the modelling concepts used in a particular domain. Semantic differs from one another as a modelling language has different semantics for different domain. Also, semantics can be defined using formal language like Z [29] or in a semi-formal instance such as a natural language [30].

In MDE, concrete and abstract syntaxes plus semantics together describe a modelling language [31]. Many approaches to defining languages have been specified but using these three constraints are common in MDE. A metamodel is used to define abstract syntax; concrete syntax for model transformation; and behaviour models to evolve semantics [25]. For example, Figure 2.2 shows a metamodel of a language which defines models as a collection of types where types have attributes, which in turn have a type.

Figure 2.3 shows a model in an instance diagram conforming to the metamodel in 2.2 where Type String is represented after the Attribute Name of Type User,

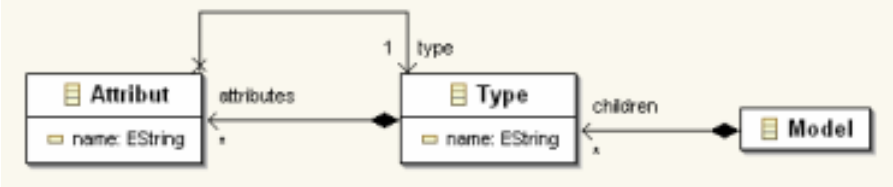


Figure 2.2: Metamodel of abstract syntax for a simple language from [2].

whose type is String. Also, a typical concrete syntax from [2], is shown in Listing 2.1.

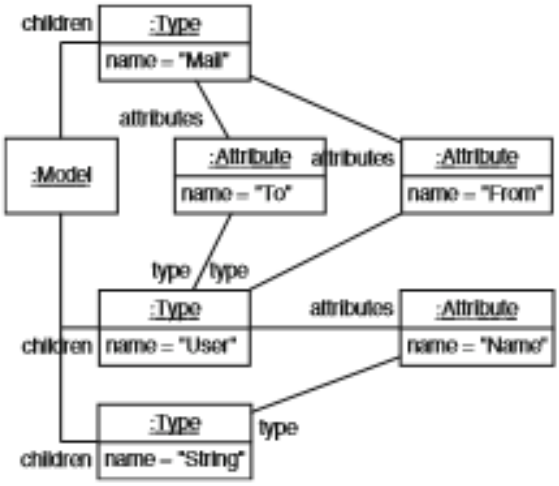


Figure 2.3: A model represented as an instance diagram from [2].

Listing 2.1: A textual concrete syntax [2]

```
1  Type Mail
2   {
3  From : User
4  To : User
5  }
6  Type String;
7  Type User
8   {
9  Name : String
10 }
```

A metamodel is used in either a domain- specific language (DSL) or general purpose language (GPL) to archetype a model, depending on the level of abstractions involved in the domain of interest [25]. The two languages are further defined below.

General Purpose Language - general purpose modelling languages include UML, widely used by modellers to model the abstraction of a system, from different levels and views. UML allows the development of variety of models that describe the

behaviour and structure of a system. Specifically, our domain of interest comprises models in UML, with slight variations to explain aspects of biology. GPLs provide general concepts (abstract syntax and semantics) and a generic concrete syntax.

Domain-Specific Language - In MDE, DSLs are tailored to a particular domain. For our work, languages such as EOL[1], EVL[2], ETL[3] are MDE DSLs used to query, validate and transform models, respectively. While these languages' scope is limited to one domain, they provide a concise solution at same level of abstraction as the problem domain, by being less suceptible to portability [32]. DSLs can capture concepts of a modelling domain(abstract syntax, semantics) and use notations familiar in the domain for the concrete syntax.

2.1.3 MOF: A Metamodelling Language

One of the standard languages for specifying metamodels is called the Meta-Object facility (MOF); it is defined by Object Management Group (OMG)[4]. UML is the origin of MOF [3]. MOF allows the expression of abstract syntax for modelling language by developers. MOF is supplemented by the Object Constraint Language (OCL) which is a formal language used in defining constraints [33]. MOF was developed because of the need to have a standard form of defining metamodels for MDE.

MOF is a modelling language used in defining modelling languages hence it is called a metamodelling language. Figure 2.4 is defined in MOF using a concrete syntax similar to a UML class diagram. According to [34], "The objective of the MOF standard is to enhance consistency in the way in which modelling languages are specified. Without a standardised metamodelling language modelling tools can have diverse modelling languages, which makes interoperability challenging. With a common metamodelling language in place, tools can create modelling languages with the metamodelling language and exchange such modelling languages with no compatibility issues. Thus, a standardised metamodelling language promotes modelling tool interoperability [34].

2.1.4 MDE Guidelines

Effective approaches and tools are needed for MDE efficient model engineering practices. This section discusses the level of abstraction involved in MDE, the method and tools used in this research.

[1]Epsilon Object Language : http://www.eclipse.org/epsilon/doc/eol/
[2]Epsilon Validation Language : http://www.eclipse.org/epsilon/doc/evl/
[3]Epsilon Transformation Language : http://www.eclipse.org/epsilon/doc/eol/
[4]Object Management group : http://www.omg.org/spec/

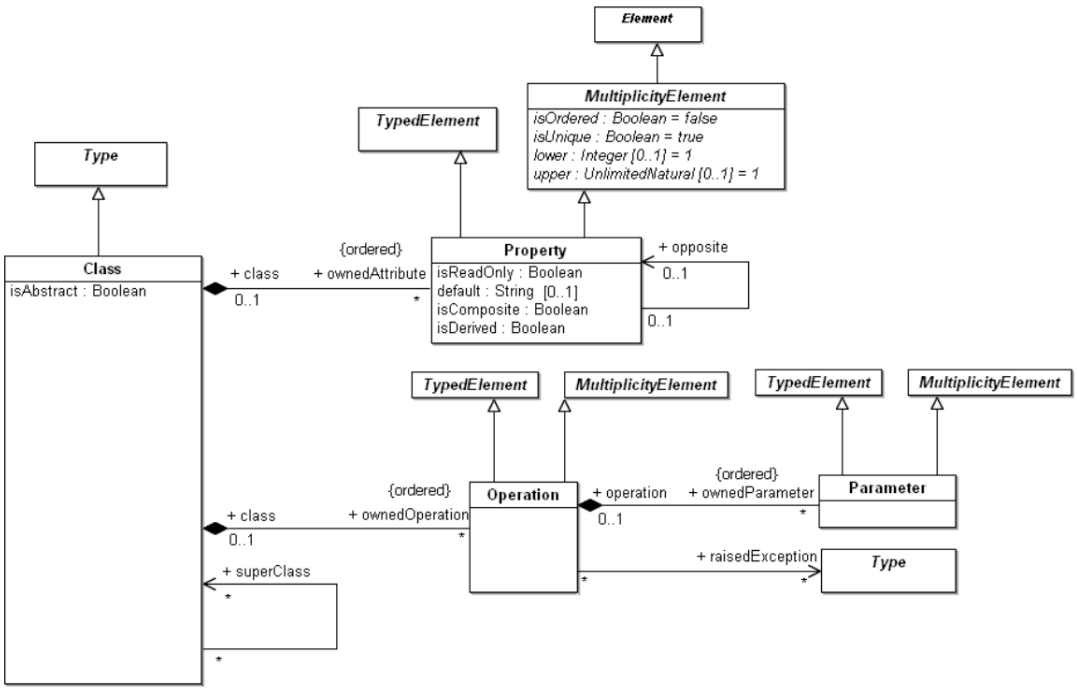


Figure 2.4: An excerpt of the UML metamodel defined in MOF, from [3].

The Model Driven Architecture

Model Driven Architecture (MDA) is an approach to developing technological complex systems. It proffers a standard to models and modeling languages. This standard is reflected in representing and exchanging models (XMI), constraints specification (OCL), and transformation specification on models [35]. Model architecture allows models to be created at various level of abstractions mostly in standards formats like XMI. MDA is used as an approach to specifying ways in which MDE is instantiated in software engineering. MDA lays down guidelines and approaches for a MDE so as to enable the development of system from raw data and business logic, leaving behind the crucial implementation technologies.

Standards for the MDA

OMG defined set of standards for the MDA as a part of guidelines for MDE. These standards are the basis for defining metamodels and imposing constraints on models to make them conform to their metamodels. Each standard is allocated to one of four tiers and each tier shows model abstraction in different level as seen in Figure 2.5.

The base of the pyramid, M0, depicts the domain of interest (real world). M1 is the representation of the M0 - a model of the M0 concepts. M2 defines the modelling

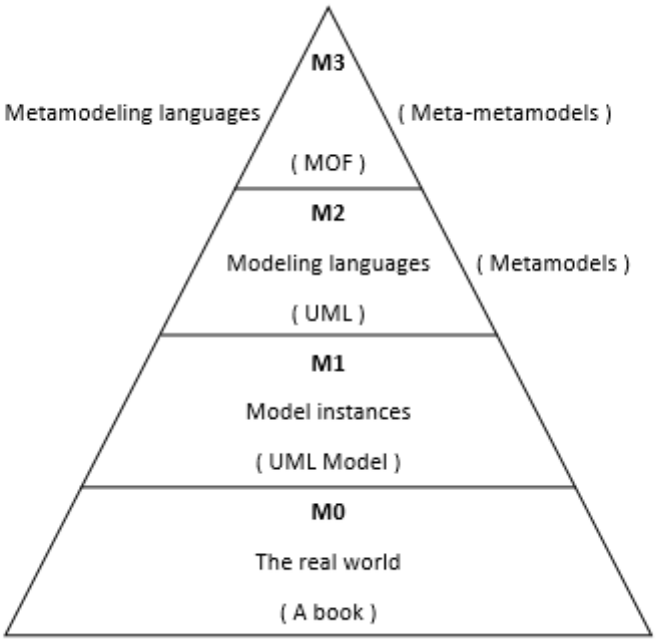


Figure 2.5: The stages of standards used as part of MDA from [4].

language (metamodels) used to define M1. M2 contains UML metamodels used in making M1 conforms to them. Lastly, M3 identifies a metamodelling language, used in defining M2. For example, if a real world (M0) is a book, the model of the book is M1, a language suitable to model books is M2 and a language suitable for defining a language (that can model a book) is M3.

2.1.5 Model Management

In MDE, models are managed to construct software. Model management refers to operators deployed in manipulating models [36]. Models and transformations are regarded as the core operations of MDE [37]. This section details these operations and their tools.

Model Transformation

Model transformation is a development operation where a modelling artefact is derived from another by systematic application of rules that map concepts of one model to the concept of another. The transformation of models is often specified to enhance quality, recognise emergent patterns, and automate software evolutions, among many other attributes [38]. Model transformations have three types which are the model-to-model, model-to-text and text-to-model transformation. The remaining part of this section discusses two relevant parts of these transformation. [5] presents Figure 2.6 which shows a model transformation pattern adapted from model driven architecture [39].

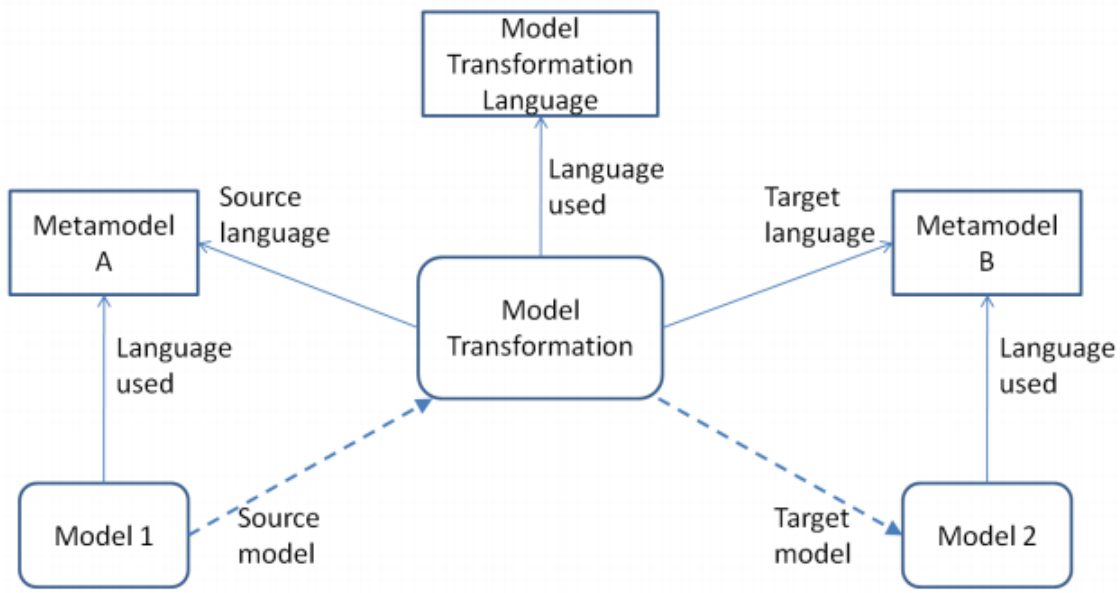


Figure 2.6: Model transformation pattern in Model Driven Development from [5].

Model to Model (M2M)

M2M is an approach to model transformation where diagrammatic models are derived from other diagrammatic model. M2M rules are written at metamodel level and when automated helps reduce engineering cost of complex systems instead of transformation between pairs of interdependent models [37]. Using M2M approach, the input model (also called the source model) conforming to a metamodel, is transformed by following a set of transformation rules, in a transformation language, to an output model (called the target model) conforming to another metamodel. Listing 2.2 is an example of a M2M transformation in the ETL transformation language, where a class is transformed to a database table. The details of the table's primary key is detailed, and if the class extends some other classes, a foreign key pointing towards the primary key of the parent class is created.

Listing 2.2: A sample ETL language of a class to a table from Epsilon ETL [40].

```
1  rule Class2Table
2    transform c : OO!Class
3    to t : DB!Table, pk : DB!Column
4     {
5    t.name = c.name;
6    t.database = db;
7
8    pk.name = t.primaryKeyName();
9    pk.type = "INT";
10   t.columns.add(pk);
11   t.primaryKeys.add(pk);
12
13   if (c.`extends`.isDefined())
```

```
14    {
15      var fk : new DB!ForeignKey;
16      var childFkCol : new DB!Column;
17      var parentFkCol : DB!Column;
18      var parentTable : DB!Table;
19
20      parentTable ::= c.`extends`;
21      parentFkCol = parentTable.primaryKeys.first();
22
23      childFkCol.name = parentFkCol.name;
24      childFkCol.type = "INT";
25      childFkCol.table = t;
26
27      fk.database = db;
28      fk.parent = parentFkCol;
29      fk.child = childFkCol;
30      fk.name = c.name + "Extends" + c.`extends`.name;
31    }
32  }
```

- **Declarative M2M** transformation languages define a relationship between source and target model using declarative constructs for mappings between models. The limitation of the declarative approach is its inability to produce fine grained rule scheduling for executable transformations [41]. Examples of declarative M2M includes QVT-relations from OMG.

- **Imperative M2M** transformation languages define a series of steps needed for transformation of models from the source to the target model, enabling exclusive control of transformation rules. The limitation to this approach is the difficulty in writing and maintaining the language [42]. Example of Imperative M2M includes QVT-operational from OMG.

- **Hybrid M2M** transformation seeks to combine declarative and imperative M2M by providing both implicit and explicit rule scheduling. Our work made use of ETL, a hybrid M2M languages in order to handle complex transformation scenarios observed in our model. ETL is further discussed in Chapter 4.

Model to Text (M2T)

M2T is an approach to model transformation where diagrammatic models can be serialized, or transformed into code or other textual artefacts. In M2T, generation is not limited to code as any type of textual artefacts such as documentations, manuals and requirements can be generated.

M2T transformation is used to produce unstructured textual artefact as in contrast to M2M. According to [25] "*M2T allow the use of mechanisms for specifying*

sections of text that will be completed manually and must not be overwritten by transformation engine" [25].

M2T languages allow the use of templates which have static and dynamic sections. A verbatim response is deduced when transformation is done in static sections. For dynamic sections, it contains executable logic. EGL is an example of language used in M2T. According to [25], Listing 2.3 contains two static sections ('package' and ';') and a dynamic output section ([%=class.package.name%]), and will generate a package declaration when executed. Similarly, line 3 will generate a class declaration. Lines 4 to 6 iterate over every attribute of the class, outputting a field declaration for each attribute.

Listing 2.3: M2T transformation in the Epsilon Generation Language from [25].

```
1 package [%=class.package.name%];
2
3  public class [%=class.name%] {
4   [% for(attribute in class.attributes) { %]
5    private [%=attribute.type%] [%=attribute.name%];
6   [% } %]
7  }
```

Model Validation

A fit-for-purpose model that captures a system's domain of interest needs to be verified and validated. Model validation provides integrity to a software system under development using MDE. A model is "incomplete, contradictory and inconsistent when it leaves out information" [42]. Also a model becomes redundant when it is incomplete and shows differences in its concepts [43]. According to [43], incompleteness and redundancy are example of inconsistency. By using model validation, some limitations of models can be detected, analyzed and corrected using MDE approaches.

To have a validated and consistent model, constraints can be specified on UML and MOF models using validation and constraint languages such as the Object Constraint Language (OCL), an OMG standard. Pertaining to our work, a validation language called Epsilon Validation Language (EVL) is used to define and evaluate constraints within and between models as OCL is limited to expresing inter-model constraints [24].

By using EVL, model dependecy can be supported among constraints imposed on model thereby decomposing complicated constraints to simpler forms. An example of constraint written in EVL from Epsilon EVL is shown in Listing 2.4. Here, the validation checks if the class in the model starts with an upper case letter. When validation is executed, the EVL model is invoked for every specified class of the model.

Listing 2.4: A sample EVL language.

```
1 context OO!Class {
```

```
2   critique NameShouldStartWithUpperCase
3    {
4     guard : self.satisfies("HasName")
5
6     check : self.name.substring(0,1) =
7       self.name.substring(0,1).toUpperCase()
8
9     message : "The name of class " + self.name +
10      " should start with an upper-case letter"
11
12    fix
13    {
14      title : "Rename class " + self.name + " to " +
15        self.name.firstToUpperCase()
16
17      do
18      {
19        self.name = self.name.firstToUpperCase();}
20    }
21  }
```

Other model management operations

With a model transformation and validation, other examples of model management operations include model merging which combines two or more models (e.g. Reuseware) [44] and model comparison where traces of same or different artefacts are constructed from two or more models (e.g. EMF compare) [45].

2.2 MDE Tools

MDE is supported by powerful tools that support model interoperability. This section reviews the MDE tools that are used in our research. Section 2.2.1 gives an overview of the Eclipse Modelling Framework (EMF) which uses MOF and supports several MDE tools and languages thereby enabling their interoperability. Section 2.2.2 discusses the papyrus tool which allows UML model and other language models to be drawn. Section 2.2.3 discusses Epsilon which is an extensible platform customizable for model management language.

In order to use behaviour models to support simulation development, our approach to modelling in MDE involves the use of diagramming tool to draw diagrams that conform to the tool's internal GPL metamodel. The purpose of this section is to give an overview of relevant MDE tools. This section does not discuss other MDE tools and environment such as ATL for M2M transformation and AMMA platform used for large scale modelling as we are more comfortable with the tools we used.

2.2.1 Eclipse Modelling Framework (EMF)

EMF is a framework that helps with the development and instantiation of metamodels. EMF provides support for MDE via a metamodelling language, Ecore, a partial implementation of OMG's MOF metamodelling language [6]. It is widely believed that EMF is the most widely used MDE modelling framework. EMF Ecore is used to define the metamodel used in our work. EMF provides both a tree-based and graphical metamodel editors and it is a contemporary MDE framework that is widely used.

EMF enables users to define their own metamodels in Ecore. Figure 2.7 presents a high level overview of Ecore. In Ecore, the name of every element of a class starts with 'E', as every Ecore object is an EObject. The root element of an Ecore metamodel is an EPackage. Other metamodel elements include EDataTypes, EClasses, EAttributes, EReferences.

In our work, with the definition of a metamodel, EMF enables the generation of a model editor that allows us to create models that conforms to our metamodel. Likewise, a graphical modelling framework (GMF) can be created from metamodels instantiated with EMF. A model driven approach where several models are specified, merged and transformed to enable code generation is made possible from a graphical editor. Also, several MDE tools are interoperable with EMF.

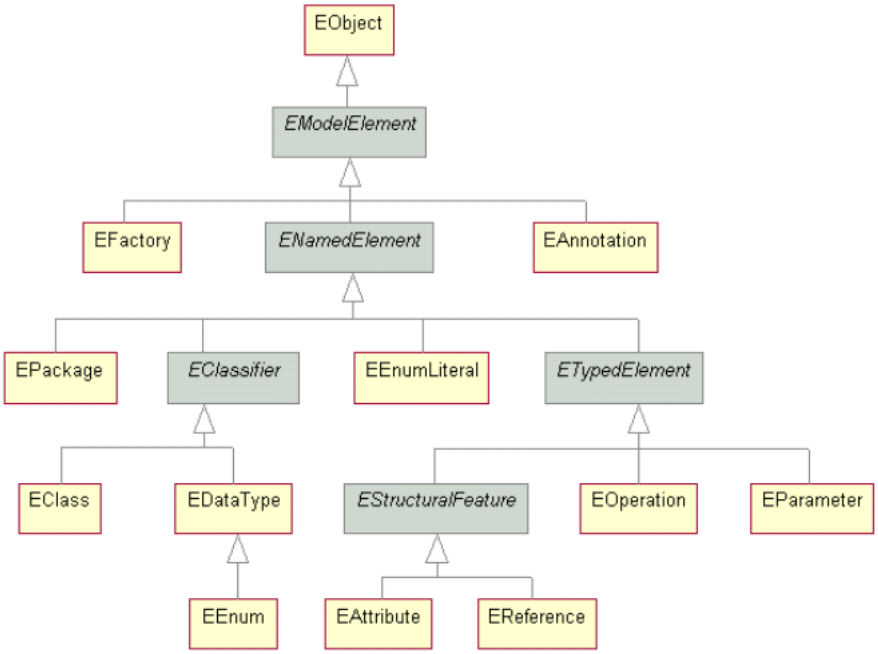


Figure 2.7: The Ecore Metamodeling Language taken from [6].

2.2.2 Papyrus

Papyrus provides an environment where any kind of EMF model can be edited. It also supports UML and other related modelling language like SysML and Marte

[46]. A model (diagram) editor and support is provided for EMF thereby enabling the drawing of UML diagram where their serialization is made interoperable. In Papyrus, a UML metamodel and graphical models are used to define and modify models. A behaviour or structure model can be easily created. We deploy Papyrus to create our behaviour diagrams to structure diagrams. Thus enables us to automate different UML models representing different domain of our system, model and metamodels. Figure 2.8 shows a papyrus modelling environment where different UML diagram can be created from papyrus website [47].

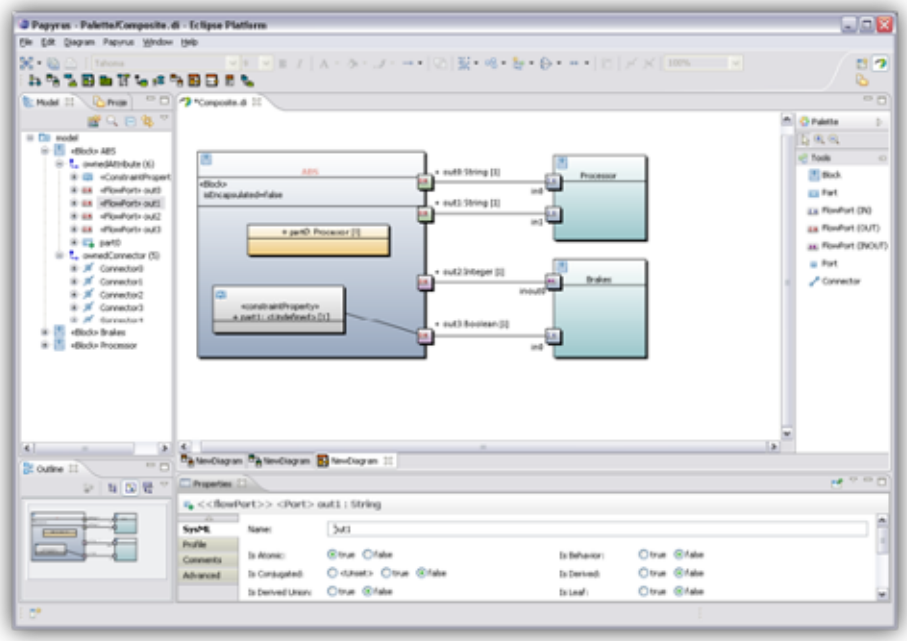


Figure 2.8: A Papyrus model environment.

2.2.3 Epsilon

The Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon) [42] is a powerful tool suite for MDE. It incorporates many model management languages and it is used for performing management tasks such as transformation, validation and merging [42]. Figure 2.9 shows the architecture of Epsilon and it comprises of two main components which are the Epsilon language family and the layer of Epsilon Model Connectivity (EMC).

According to [26], "Epsilon is modelling technology agnostic". Whilst many model management languages are bound to a particular subset of modelling technologies thereby inhibiting their utilization, Epsilon is able to manipulate models expressed in various modelling languages [48]. Currently, Epsilon supports models implemented with EMF, MOF, XML, or Community Z Tools (CZT) and they are supported by technology-specific drivers [26].

Furthermore, Epsilon allows model reusability when constructing independent model management languages. EOL which is the core language of Epsilon platform

provides functionality closely related to OCL but with more features provides functionality such as model updates, access to various models, imperative (conditional and loop) statements, standard output, user feedback and error reporting [25].

Epsilon supports lightweight means for defining new developmental languages for MDE. As shown in Figure 2.9, EOL is used to build task-specific languages such as Epsilon Generation language(EGL) for model-to-text transformation, Epsilon Wizard Language (EWL) for model-to-model transformations, Epsilon Comparison Language (ECL) for model comparison. Other languages includes Epsilon Merging Language (EML) for model merging, Epsilon Transformation Language (ETL) for model-to-model transformations, Epsilon Validation Language (EVL) for model validation, Epsilon Flock for model migration and Epsilon Pattern Language (EPL) for pattern-based querying. Epsilon is suitable as a platform for the research of this thesis as it support modelling technologies and task-specific model management operations.

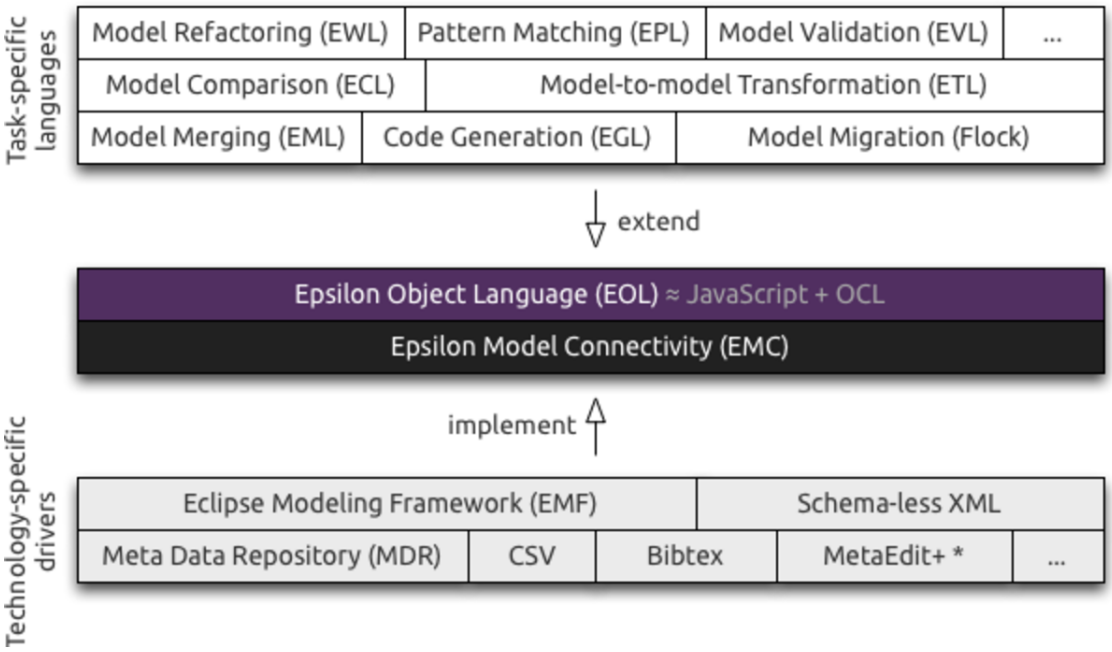


Figure 2.9: The architecture of Epsilon, taken from Eclipse Epsilon [7].

2.2.4 Summary

This section introduced MDE tools used in this thesis. The Eclipse Modelling Framework (EMF)enables the implementation of MOF or Ecore for defining metamodels. Using metamodels defined in Ecore, EMF aid persisting models to disk enable code generation for distinct metamodel editors. Tools such as Epsilon and Graphical Modelling Framework (GMF) is used to enhance EMF's functionality as it is the most used contemporary MDE framework. GMF aid metamodel specification of

graphical concrete syntax and therefore enable generation of graphical model editors. Epsilon as an extensible platform enables re-use so as to aid the expression of new model management language.

2.3 MDE Benefits and Recent Challenges

In comparison to traditional software engineering approaches such as language-oriented programming and domain-specific modelling, MDE has offered tremendous benefits which comes with limitations. This section discusses MDE benefits and its challenges.

2.3.1 Benefits

As discussed in this chapter, benefits of MDE are expressed to help describe the advantages of MDE practices, approaches and why MDE is crucial to complex system modelling.

Interoperability

MOF as a standard metamodelling language enables modelling tools interoperability through model interchange. EMF utilizes Ecore to provide a holistic reference implementation of MOF thereby enabling the development of several contemporary MDE tool. Model management operations performed between different modelling tools is achieved through interoperability among modelling tools thereby modellers are not tied to just one specific modelling tool. Also, single environment can be used by models represented in wide range of modelling languages. Before the advent of MOF, developers used several tools for each modelling language but with MDE, interoperability of models is made possible.

2.3.2 Challenges

MDE has helped software engineers a lot whether through using domain specific modelling to automate traditional software engineering practices or model management operations. Also, tool interoperability allowed compability of various tools used on different models.

Whilst MDE benefits have been highlighted above, some challenges it faces is discussed here. The challenges to MDE has been identified and they are used as a motivation towards exploring potential research areas to improve MDE. This section highlights these challenges and they are reviewed below.

Learnability

It is difficult for a new user to utilize MDE properly. This is due to its increasing developmental activities and conceptualized principle to software engineering. The learnability of MDE comes from its perceived adoption in a mainstream spectrum. According to [49], "GMF is difficult for new users to understand and mechanisms for its simplification have been proposed recently" [49].

Scalability

Scalabilty is the need to make MDE more powerful. According to [50], "MDE is increasingly getting utilized in modelling complex systems therefore its modelling languages and model management tools are being stretched to accommodate more model activities such as collaborative development, model persistence when dealing with models with large megabytes" [50].

Furthermore, the relevance of MDE in software engineering depends on the need to bring MDE languages and tools to scale. This scalability will help accommodate modelling of large and complex models.

Scalability in MDE is considered as the "Holy Grail of MDE as its a major concern for modellers" [50]. The achievement of MDE scalability comes in the need to enable the development of large models and domain specific languages. Also, the enhance of model management tool to efficiently accomodate large models is a huge step towards scalability. MDE needs to enable large modeller teams to develop and refine large models.

2.4 Chapter Summary

This chapter provides a background review of Model-Driven Engineering (MDE) - a crucial modelling approach in software engineering. Modelling artefacts such as model, metamodels and model architecture are discussed. Also, several model management operations such as model transformation (focus on model-to-model and model-to-text transformation) and model validation are discussed.

The Eclipse Modelling Framework (EMF) and Epsilon platform which are widely utilized in this thesis were discussed. Finally, benefits and challenges identified in MDE that are closely related to this thesis were highlighted. This thesis uses Read [8] behaviour diagram to explore MDE and subsequent chapters discuss it further while the next chapter, Chapter 3 presents the domain analysis and observation from [8].

Chapter 3

EAE: Domain Analysis and Observation

This chapter presents the domain model of an EAE immune system from Read's thesis [8] *using the Unified Modeling Language (UML) and other diagrams provided in the thesis.*

3.1 Introduction

Experimental autoimmune encephalomyelitis (EAE) is a murine autoimmune disease which has many parallels with multiple sclerosis [8]. According to [8], the motivation for the simulation and modeling of this immune system stems from the need to understand the emergent behaviour and patterns observed in the EAE system.

Read's wet-lab experimentation detailed his observation on EAE system for modelling and simulation. According to [8], "EAE as resulting from sub-cutaneous immunisation with myelin basic protein (MBP), complete Freunds adjuvant (CFA), and pertussis toxin (PTx), leads to damage of the central nervous system (CNS). This leads to paralysis in the subject. Following the induction of EAE, the majority of experimental animals experience physiological recovery from paralysis; no experimental intervention is administered in facilitating recovery. Lastly, mice having undergone recovery from autoimmunity are resistant to subsequent attempts to induce paralysis with similar immunisation".

In this project, Read's design models [8] are used. By deploying MDE techniques, we explore the feasibility of automatically generating simulator code from the designs. Model simulation helps capture and integrate different information thereby providing a system-level synopsis of what the wet-lab data represents.

In Read's work [8], simulation helps to guide wet-lab experimentation, as an *in-silico* investigation can show crucial part of a system highlighting areas where information on EAE system is insufficient. The targeted collection of information on the system can help bridge the gap between competing theories identified in the domain of the system.

Read [8] shows that the modelling and simulation of the EAE system presents a fit-for-purpose representation of the target domain through the generation of *in-silico* hypotheses which can be verified further in wet-lab experimentation. It is important to note that the need for confidence in simulation results comes from the demonstration of fitness-for-purpose models representing the immune system.

Crucially, explaining hypotheses, as well as tuning the simulator, requires many revision to the code, making it hard to keep models and code in synch manually. We would like to ultimately to "tweak" models not code, as biologists understand the models and changes to models.

3.2 Domain Model Overview

Read's domain model [8] is tailored using the framework of the CoSMoS process where the first stage details the development of the domain model that capture the detailed understanding of a system's domain [51]. The detailed model captures the exploration of the domain of interest before simulation and provides an avenue for exploring MDE from UML behaviour models in our work.

The domain of a system encompasses the system's scientific scope as seen from the scientist or domain expert perspective. The domain expert and developers use domain models to determine the purpose of simulation thereby guiding the developers to a fit-for-purpose simulation.

For detailed analysis, [52] describes domain as "a development contract: a developer depends on the scientist to provide appropriate information about the domain, and guarantees a desirable simulation; meanwhile, the scientist also depends on the developer to use the domain information suitably, and guarantees to work with the developer to ensure that the right outcome is achieved" [52].

Domain model reveals the limitation of knowledge about the system, as well as inconsistency and underspecification, which can be resolved by making assumptions of the domain. According to [8], a domain model is validated by domain experts who help to make sense off different specific domain understandings. In MDE terms, a model is verified by its conformance to a metamodel and semantics checking.

3.3 EAE Model Diagram

Analytically, EAE domain model in [8], like many other biological illustration uses bespoke, formal and informal illustrations to show the distinct interactions between EAE system's complexity. Modelling techniques used to describe EAE uses defined syntax and semantics as well as varied notations which are captured in the diagrams shown in Section 3.4.

3.3.1 Behavioural diagram

The UML behavioural diagram enables the specification and visualization of dynamic aspects of EAE system simulation. The diagram represents a series of actions. Activity diagram and state machines are widely used to capture the different levels of abstraction in [8]. Activity diagrams represent EAE sequences and conditions for EAE lower-level behaviours.

On the other hand, state diagrams show potential evolutions of elements of the EAE system; they define EAE object existence in different states. State diagrams also show how these objects transition between states. According to [12], "A transition is a response to an event, and an event is, typically, an input received by the object(or system). Transitions are protected by guards - a set of conditions, concerning the wider system state (and perhaps the environmental context of the system) that must be true if the state is to change" [12].

Benefits of using state diagrams for modelling EAE include the expression of dynamic structures and simulation of objects collection. They allow the use of concurrent state. This makes it possible to build sophisticated models used in EAE cell interactions [8]. The EAE model [8] uses both activity diagrams and state diagrams to model EAE's behaviour.

As discussed earlier, EAE models [8] are mostly represented using UML diagrams and they are discussed extensively in Section 3.4.

3.4 EAE Domain Model

Read's thesis [8] presents the EAE domain model using different UML models. The domain model is detailed using a top-down methodology and each layer presents the domain of interest according to the level of abstraction observed in them. The layers are described in the following subsections.

3.4.1 The System-level Overview

This section introduces the most abstract view of Read's EAE model [8]. The top layer of Read's design [8] is shown in Figure 3.1. The expected behaviours diagram format was devised by [8] and is used in all subsequent work by YCIL [13, 14]. The model reproduced in Figure 3.1 captures relevant observable phenomena, the potential foci of the simulation and, for each potential phenomena, the informal notation captures known biological components (cells, neurons) and the domain expert view of how these interact.

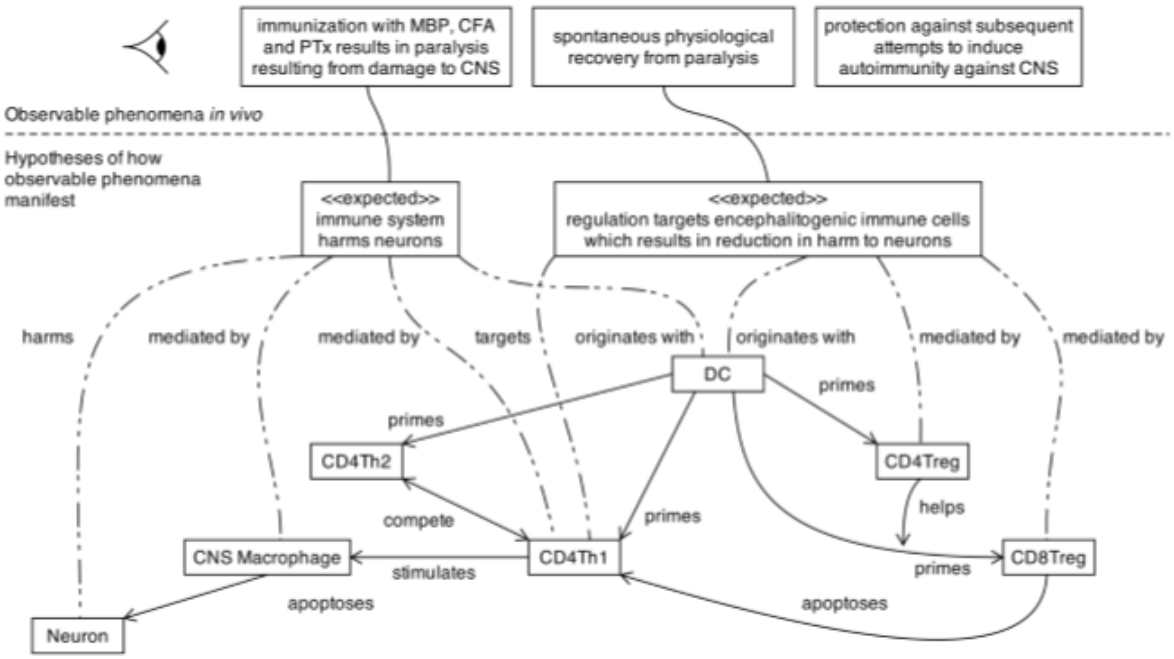


Figure 3.1: Read's expected behaviours diagram depicting the phenomena observed in a real domain, and the behaviours manifesting from cellular interactions believed to be responsible for them [8].

3.4.2 The System's and Modelling Perspectives

From the expected behaviour diagram and the domain exploration that it captures, [8] uses an informal box and arrow diagram to show 6 bodily compartments, Figure 3.2. Read's simulation [8] ultimately simulates each of these compartments, linked as shown in Figure 3.2.

Having summarized the domain in the expected behaviours and compartment diagrams, the UML base modelling expresses the activity of the cells and their interactions that make up the system's behaviours in each relevant stage of the disease and recovery. Different UML notations such as activity diagram are used in expressing these models. Also, some minor variant notations are introduced by Read [8].

EAE perspectives expressed in [8] detail how EAE causes paralysis and how mice recover, which is considered to be inherently complex. These perspectives describe collective consequences of cellular interactions, and they are grouped into four stages:

1. The initial establishment of autoimmunity in the CNS following immunisation.
2. The self-perpetuation of autoimmunity.

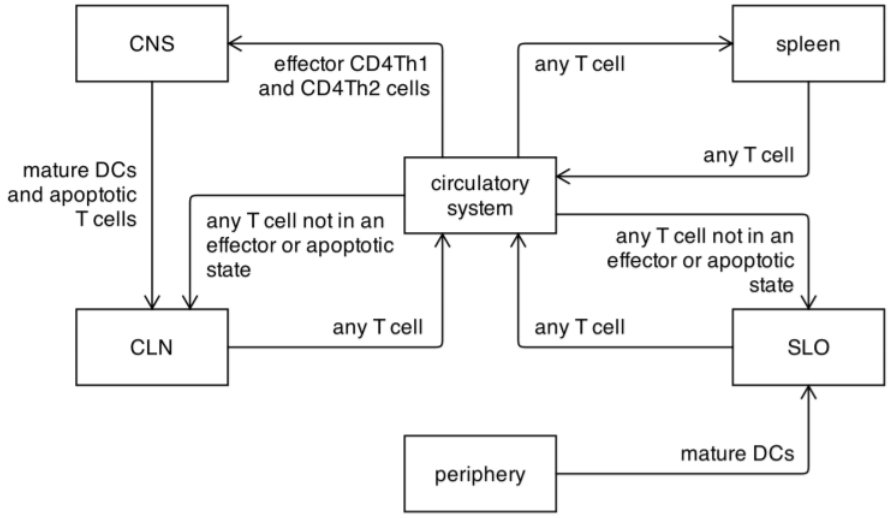


Figure 3.2: The spatial components of the domain model, and the manner in which the cells of the domain model may migrate between them from [8].

3. The establishment of regulation that results in the apoptosis of CD4Th1 cells.

4. A deviation of autoimmune response that results from regulatory activity and ultimately leads to the termination of both autoimmune and regulatory immune responses.

Furthermore, two diagrams are used to show how immunity is established and the apoptosis state. The first section (Figure 3.3.) focuses on the DC or CD4 cell activities and the compartment-level migration, whilst Figure 3.6. focuses on the biochemistry of the CD4 cells, stimulated by the apoptosis of an infected CD4 cell, phagocytosed by DC.

Initial establishment of autoimmunity

The initial establishment of EAE autoimmunity shows the events that lead from immunization to the apoptosis of neurons in the CNS. The various activities of the composing cells, how they effect changes and move around in the EAE system are shown in Figure 3.3. UML variation is introduced in the SLO compartment of Figure 3.3.

Figure 3.4 is a notation borrowed from biological modelling of feedback loops which represents an inhibitory response which leads to producing naive daughter cell through an action called *spawning*. According to [8], "the majority of these naive daughter cells will immediately bind the MHC-II:MBP complexes expressed by the priming DC, and follow a similar sequence of events as the parent cell. Those that do not begin priming on the same DC as their parents assume migratory behaviour" [8].

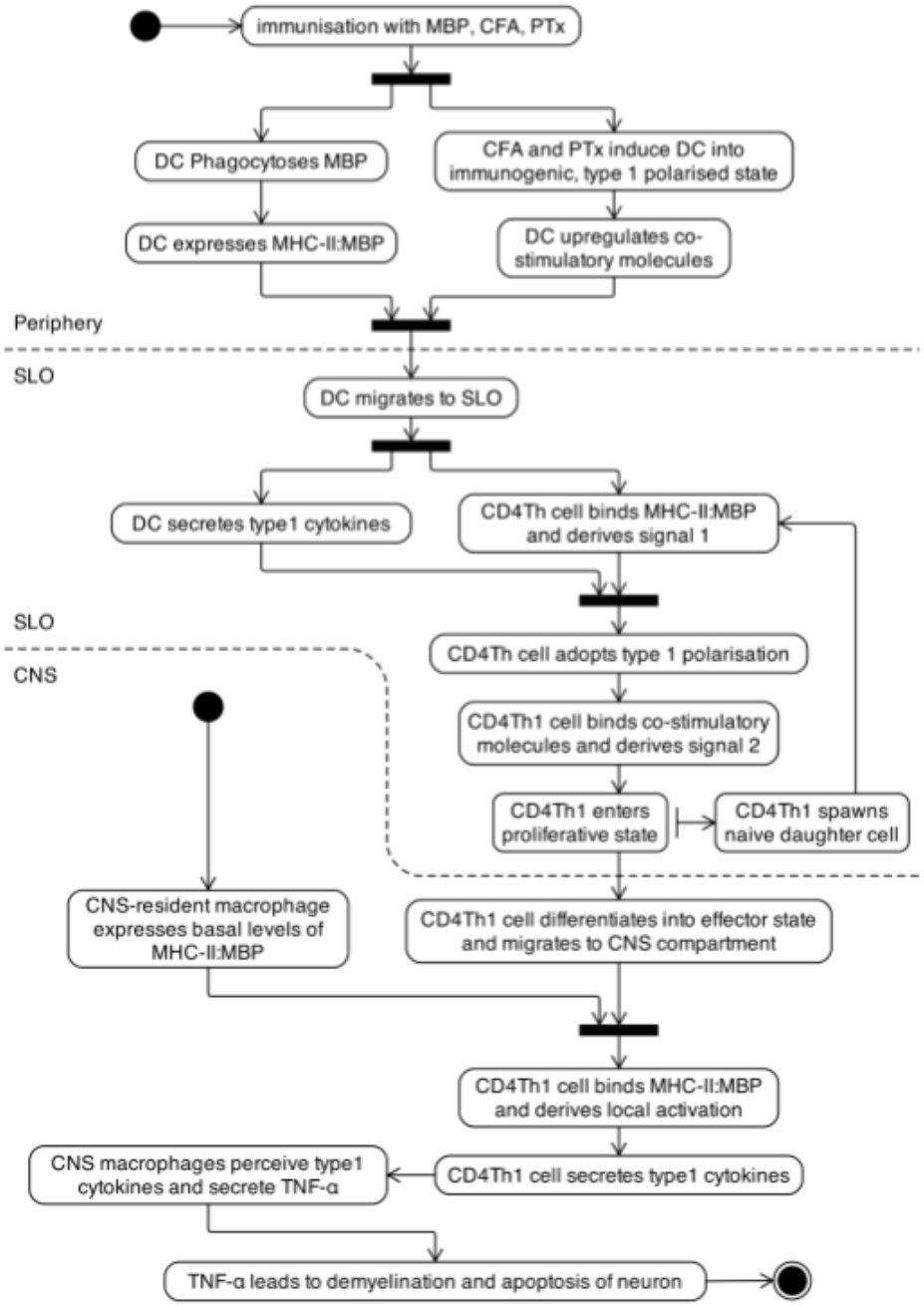


Figure 3.3: UML activity diagram depicting the cellular interactions and events that lead to neuronal apoptosis in the CNS following immunisation for EAE from [8].

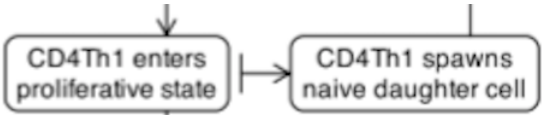


Figure 3.4: The introduced notation variation in Figure 3.3 from [8].

Self-perpetuation of autoimmunity

According to [8], after the initial establishment of automimmunity, some series of events enables its self-perpetuation. These series of events and the interactions

between cells are important to help understand EAE emergent patterns. The behaviour is represented using an activity diagram in Figure 3.5. As the nature of autoimmunity is self-perpetuating, the diagram has no termination.

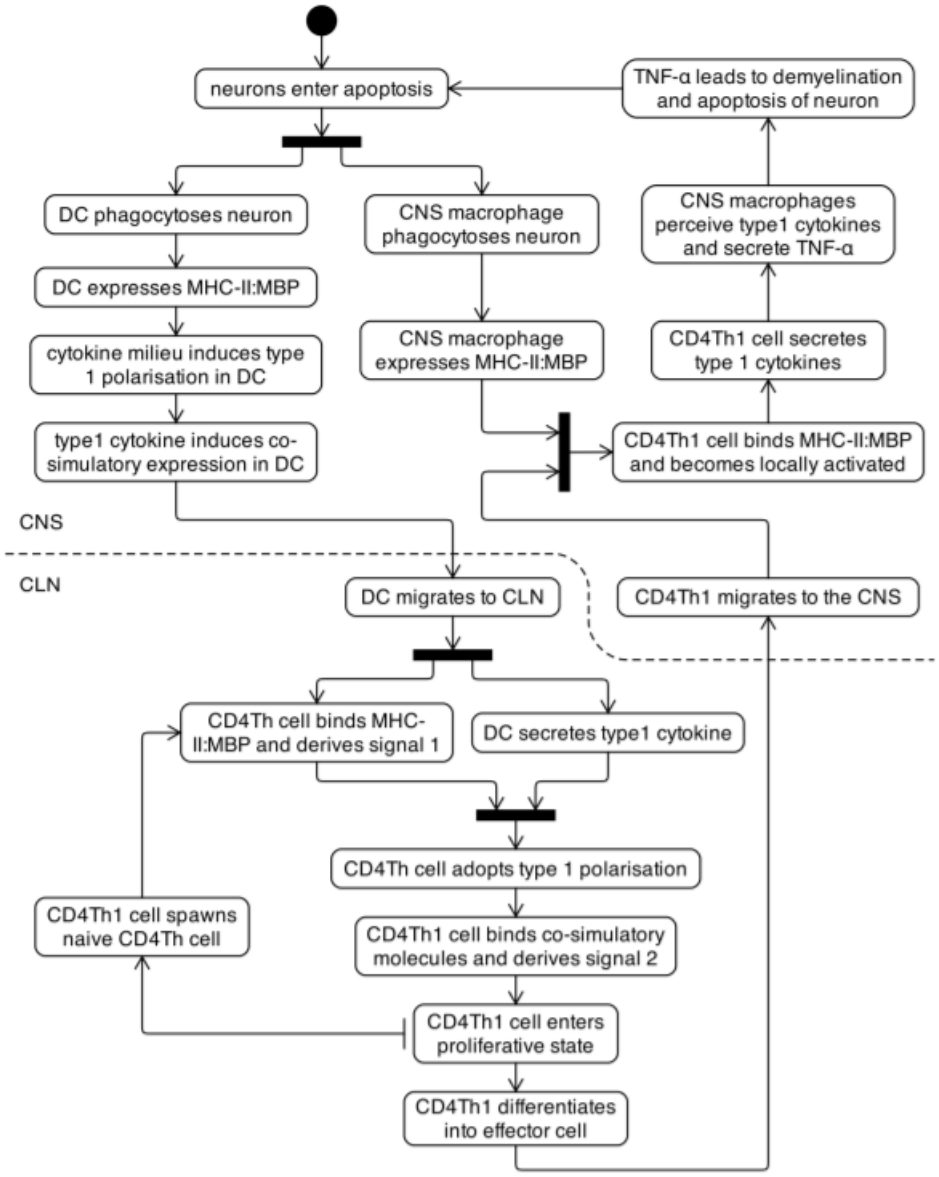


Figure 3.5: UML activity diagram depicting the cellular interactions and events that lead to the self-perpetuation of autoimmunity following neuronal apoptosis resulting from immunisation for EAE from [8].

Establishment of regulation

The establishment of regulation creates a physiological lifecycle of cells leading them to enter apoptosis. This regulation enables the recognition of actions between the cellular interactions of the system. Most importantly, the regulatory immune re-

sponse perpetuates thereby it has no terminating state, and it is shown in Figure 3.6. Furthermore, the first section i.e. Figure 3.3 focuses on the DC and CD4 cell activities and the compartment-level migration, whilst Figure 3.6 focuses on the biochemistry of the CD4 cells, stimulated by the apoptosis of an infected CD4 cell, phagocytosed by DC.

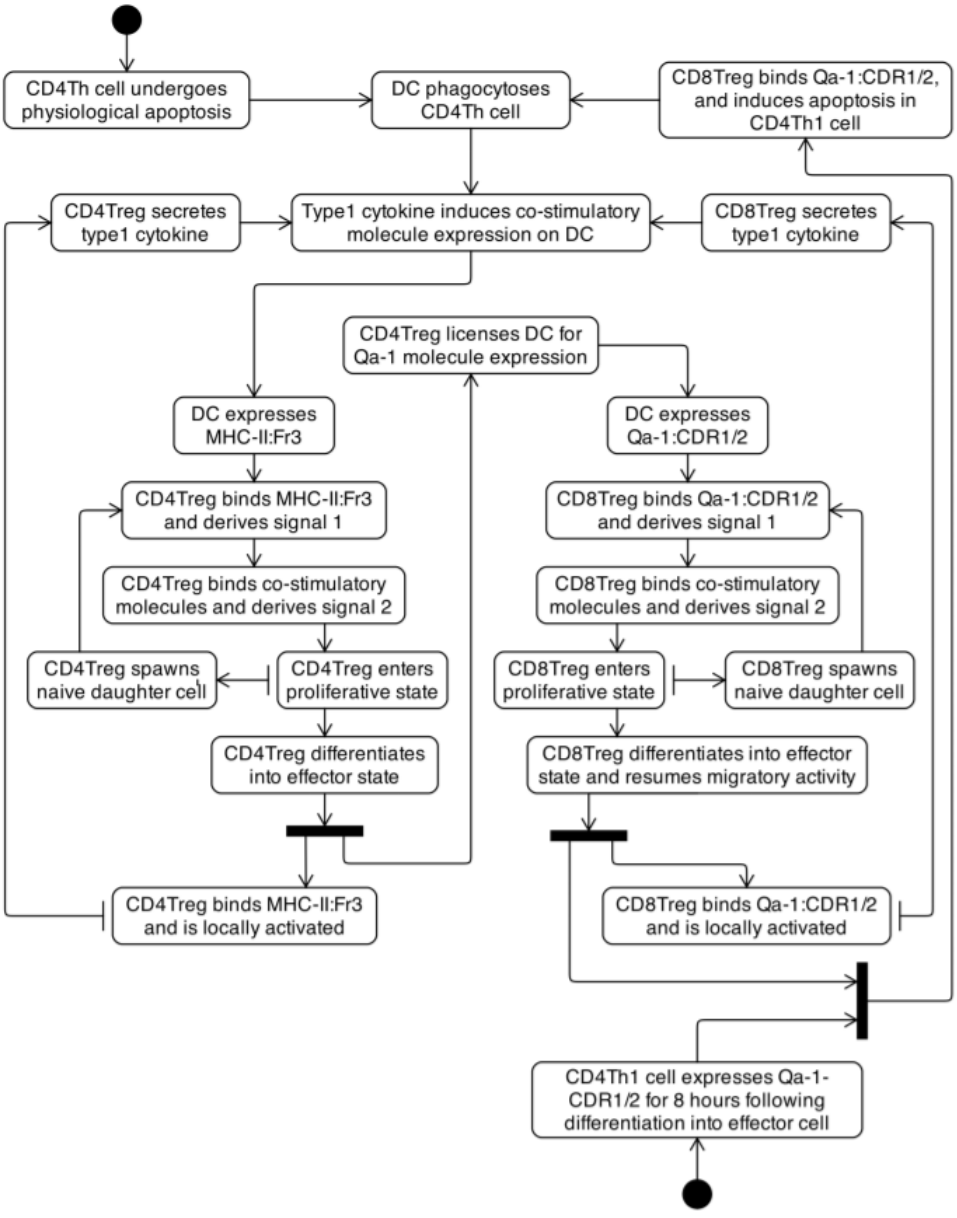


Figure 3.6: UML activity diagram depicting the cellular interactions and events that lead to the self-perpetuation of autoimmunity following neuronal apoptosis resulting from immunisation for EAE from [8].

Deviation of the autoimmune response

For an EAE infection and its recovery, Read [8] considers a specific deviation of the autoimmune response. According to [8], "the actions depicted here are cyclic, the deviation does not occur as a single atomic action within the system, but emerges as a gradual shift in behaviours spanning multiple populations of cells. However, the autoimmune response does eventually terminate, and hence an end state is expressed" [8]. This is modelled in Figure 3.7. The diagram is a non-standard UML model as notations are introduced to accommodate the biological entities.

Figure 3.8 is a notation borrowed from biological modelling of cyclic types showing CNS cytokine milieu. According to [8], "the CNS cytokine milieu is composed primarily of type 1 cytokine. In sufficient concentration, type 1 cytokine leads to neuronal apoptosis" [8].

3.4.3 The System Single-Entity Dynamics

The activity diagrams in Section 3.4.2 are used to model the structure of the behaviour of EAE. An alternative view is provided using UML state diagrams, which shows the permitted sequence of behaviours for each type of cell (in UML, for objects of each class).

Read's state diagrams [8] include invariant conditions that control the state transitions observed in the EAE system.

T-cell Dynamics

Read [8] presents relevant dynamics of T-cells shown as part of the domain model. The state machine diagrams (Figures 3.9 and 3.10) show the different behaviour of T-cells. A concurrent state diagram shows how T-cells migrate around the body compartments (Figure 3.9). For the T-cell model, Read[8] affirms that several T-cell types represented in the domain have similar characteristics as they commence their life cycle in a naive state, circulatory system and gears toward migratory behaviour. Subsequentrly Read's model and simulation [8] only considers generic T-cell behaviour.

DC and CNS macrophage Dynamics

According to [8], the relevant involvement of a dendritic cell (DC) begins in an immature state and further matures later. DCs are also responsible for priming T-cell. As for the T-cell, Read [8] uses a concurrent state model to illustrate the relationship of DC behaviour and the body compartments (Figure 3.11). The DC cells have a range of relevant concurrent behaviour options which are not specific to compartments.

The CNS macrophage is found on the central nervous system (CNS) and exhibits a subset of the behaviour of the DC. These cells exist in immature and mature

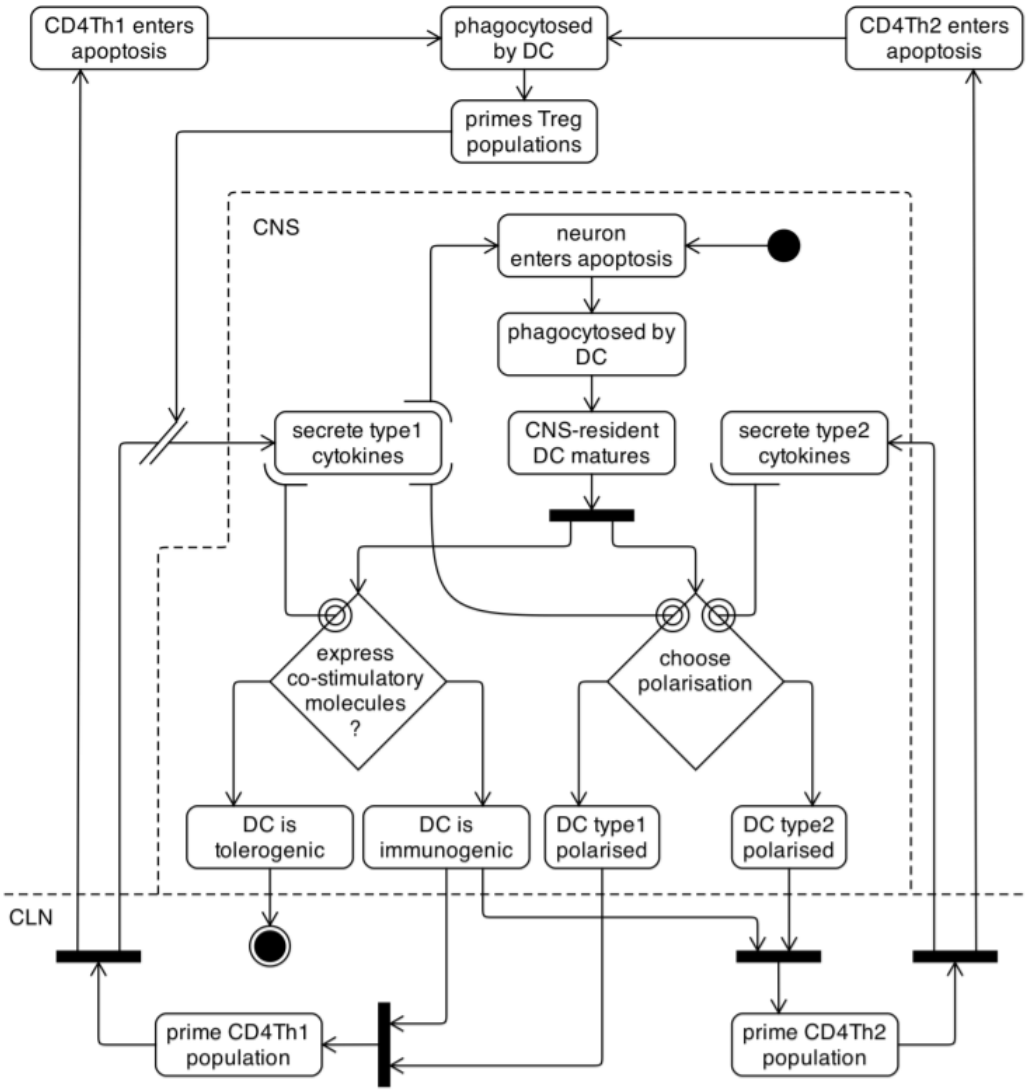


Figure 3.7: Activity diagram depicting the cellular interactions and events that lead to the deviation of the immune response from [8]. Additional notations include parallel slashes representing the collision of the primed T-cell population (Treg and CD4Th1) cells which outnumbers their CD4Th2 counterparts, and as such the CNS cytokine milieu is decomposed into primarily type 1 cytokine. In sufficient concentration, type 1 cytokine leads to neuronal apoptosis. Also, with DCs in the CNS phagocytoses of apoptotic neurons, the double circles represent DC maturation which lead to its adoption of either a type 1 or type 2 polarisation, depending on the balance of type 1 and type 2 cytokines in their local vicinity.

state. They are represented with the state machine diagram shown in Figure 3.12.

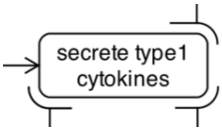
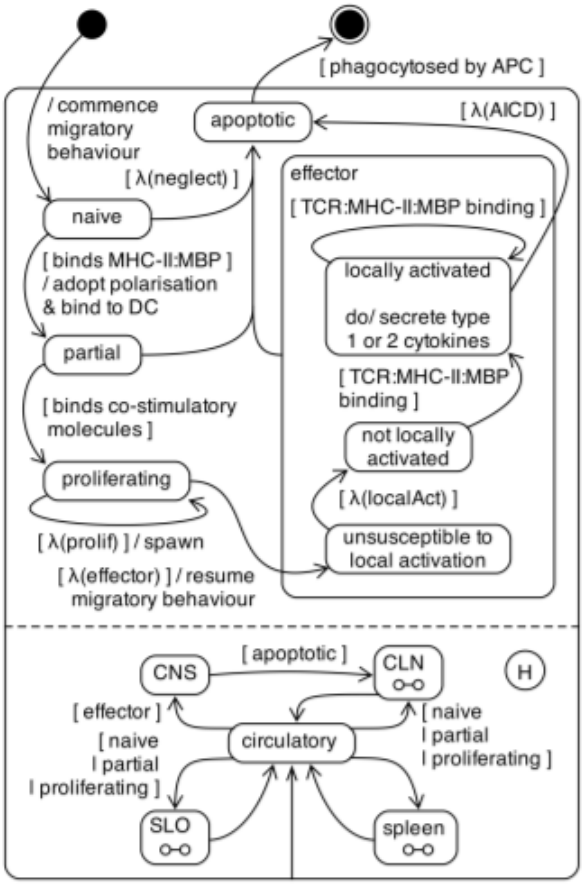


Figure 3.8: The introduce notation variation in Figure 3.7 from [8] depicting CNS cytokine milieu.



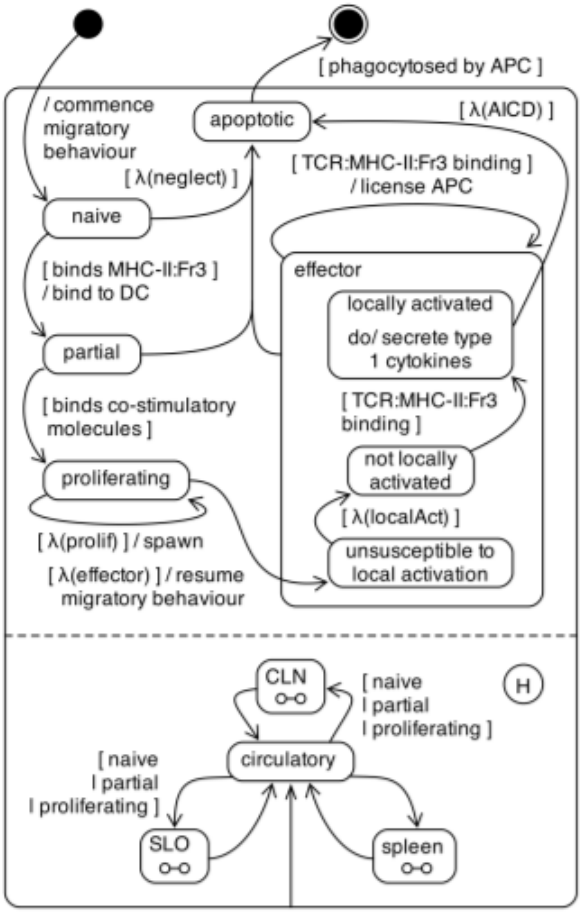
(a) Dynamics of CD4Th cells.

(b) Decomposition of the lymphoid organ states: the SLO, CLN and spleen.

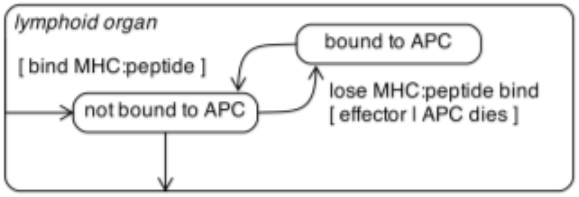
Figure 3.9: State machine diagram depicting the dynamics of CD4Th cells from [8].

Neuron and MBP Dynamics

EAE neurons express Myelin based protein (MBP) and they reside exclusively in the CNS compartment. Their dynamic behaviour is shown in Figure 3.13.



(a) Dynamics of CD4Treg cells.



(b) Decomposition of the lymphoid organ states: the SLO, CLN and spleen.

Figure 3.10: State machine diagram depicting the dynamics of CD4Treg cells from [8].

MBP in EAE is manufactured and expressed by neurons and injected into EAE system by an experimenter [8]. The MBP dynamics is detailed in the state machine shown in Figure 3.14. The state diagrams presented in this section all use un-varied UML notations.

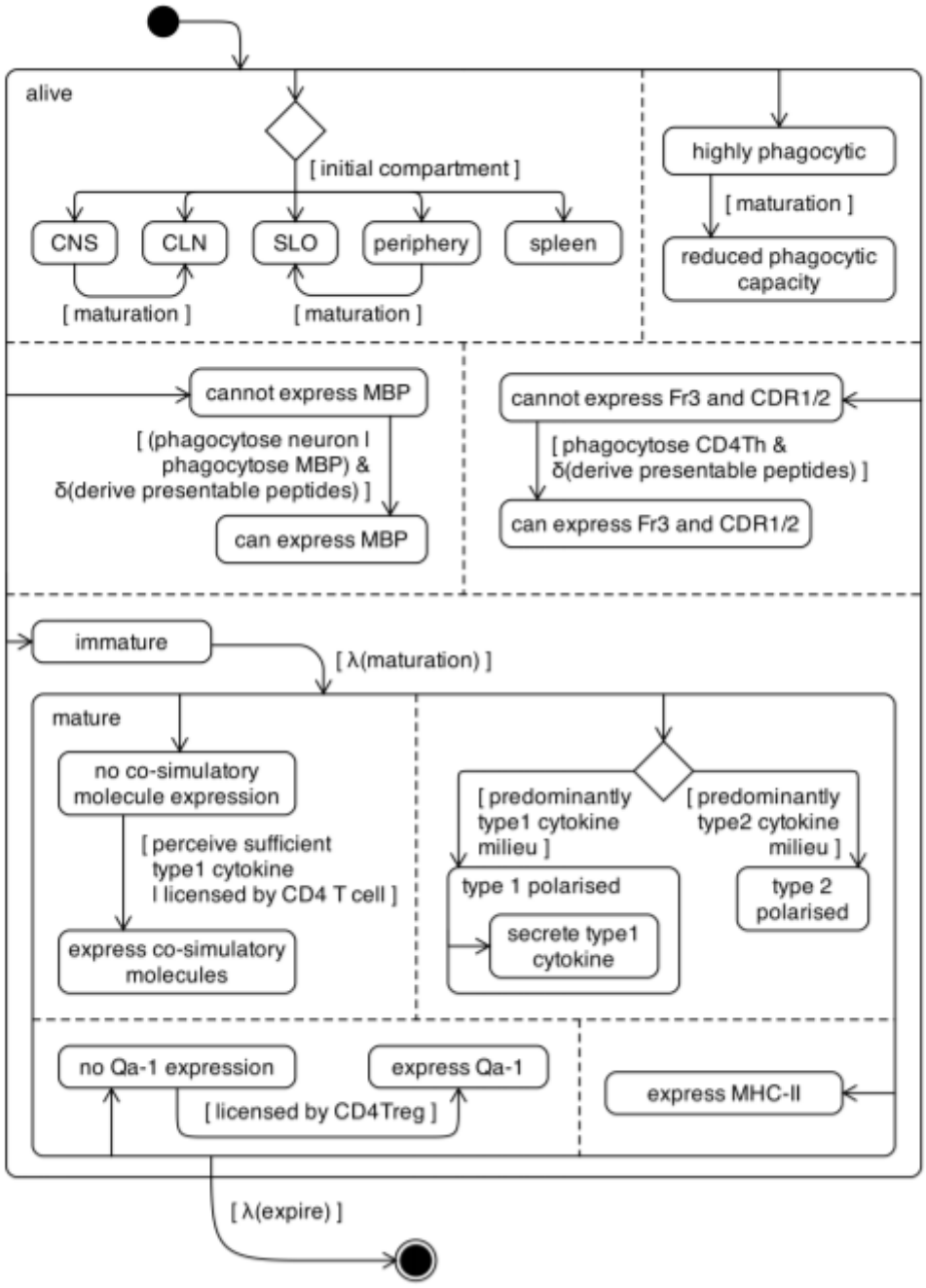


Figure 3.11: State machine diagram depicting the dynamics of dendritic cells from [8].

3.5 Analysis

The EAE *expected behaviour* diagram (Figure 3.1) presents an abstract modeling level in its domain model. It shows the relevant observations of the real domain, how they relate at different abstract level and how the interactions lead to system-wide behaviours believed to be a solid representation of the real-world system [8]. Read's model [8] has been validated by domain experts.

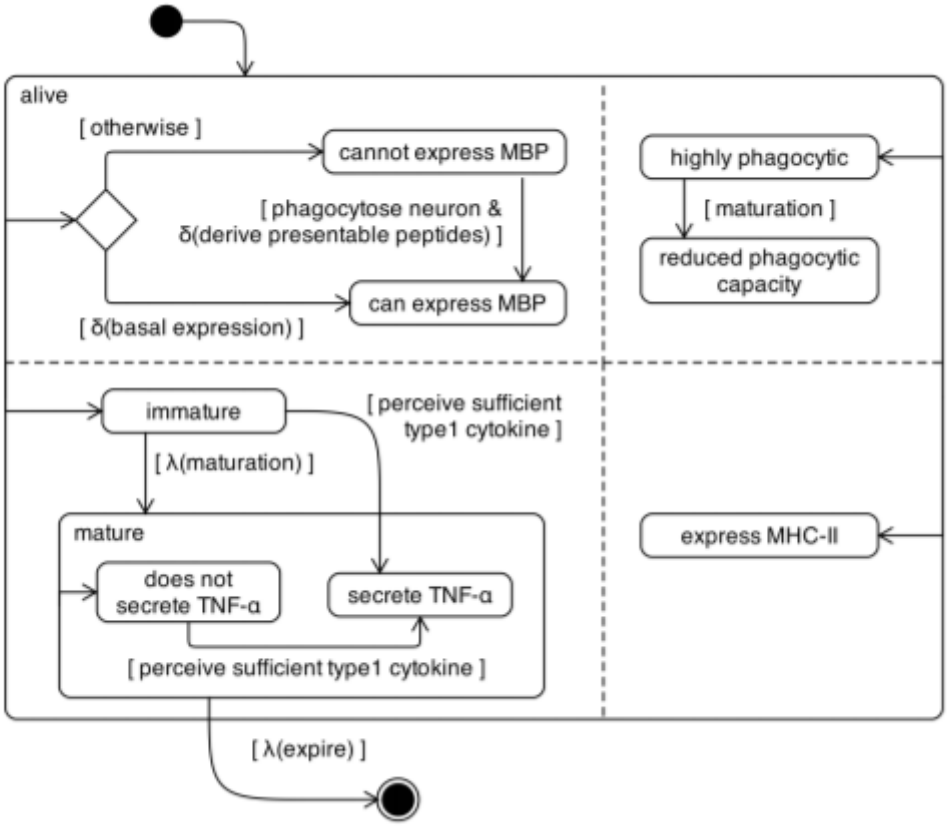


Figure 3.12: State machine diagram depicting the dynamics of CNS macrophages from [8].

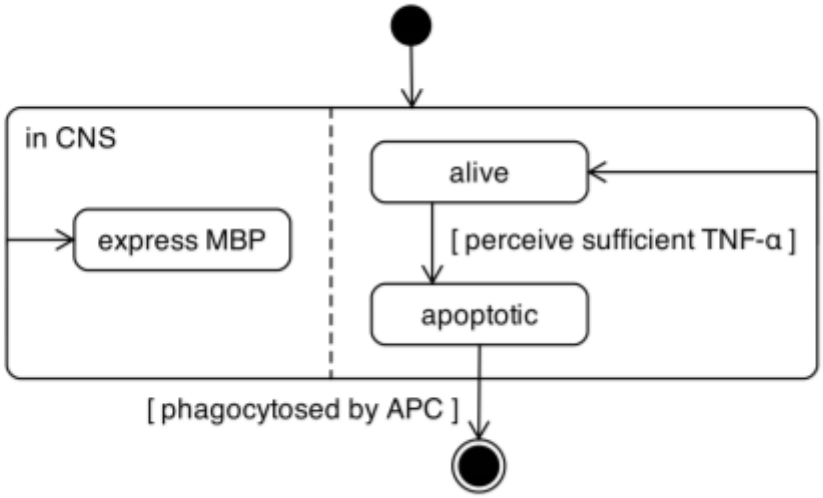


Figure 3.13: State machine diagram depicting the dynamics of neurons from [8].

UML notations are used to capture relevant behaviours of EAE and sometimes, slight notation variations are needed to capture details of the system. Read [8] states that UML does not represent the high level of concurrency-orthogonality of

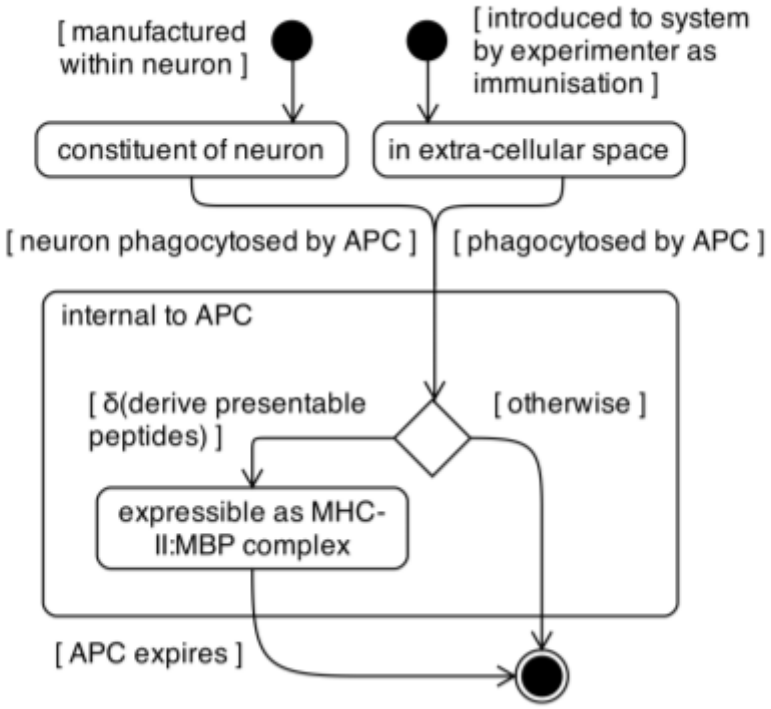


Figure 3.14: State machine diagram depicting the dynamics of myelin basic protein (MBP). from [8].

the disease. However, the diagrams help to disintegrate the high-level complexity into smaller and manageable components which can be modelled individually.

The UML activity diagram is widely used to model EAE perspectives as it has the ability to represent different abstract events and link them together as an activity. Notwithstanding, activity diagram does not represent concurrencies and stochasticity of the real domain. According to [8], "*In vivo*, there exist many populations of cells undergoing different activities depicted on the diagrams at many points in time; there is no sequential transfer of control as suggested on the activity diagram, a single cell may interact with many others at the same time, and may continue to do so after it has instigated an event in another cell " [8].

UML state diagrams have been used effectively to show dynamics of single-entities. According to [8], "Once more, it is high and partially-orthogonal dimensionality that raises issues. However, these have been satisfactorily overcome through use of guards. It is noted that many transitions depend on probabilistic or temporal conditions, and notations were devised to represent these aspects" [8].

Chapter 4

Analysis and Hypothesis

Through the review of background work in Chapter 2 and the domain analysis of EAE diagram in Chapter 3, modelling approaches were proposed. This chapter analyzes and motivates the proposed modelling approach and establishes a basis for this thesis objectives. The aim of the proposed modelling approach is outlined.

4.1 Research Background

This section summarizes motivation for our research in line with the discussions in Chapter 2 and 3.

In Chapter 3, we presented Read's domain model of EAE system [8]. This domain model provided us the system and modeling perspective of EAE, culminating into different behaviour diagram representing different level of abstraction observed in the system. Read [8] develops models and uses them as guides to implementation but does not follow precise rules or traceable transformations. This means that the validity of the implementation has not been definitively demonstrated. In general, the fitness-for-purpose of YCIL simulators relies on individual software engineers' skill, rather than repeatable validation. The target languages for implementation are usually object oriented, and code could thus be at least partially generated from models.

This chapter motivates two approaches to transforming the behaviour diagrams to a class structure diagram, using MDE tools discussed in Chapter 2 to generate OO code. Our approaches to modelling make use of the artefacts provided in Read's thesis [8] as input model in the MDE processes deployed to our transformation.

4.2 Research Hypothesis

Read [8] does not use a class diagram as the OO model of class objects as passing messages is not appropriate for an abstract model of cell interaction. Also, behaviour models imply objects; a class diagram would, if available, give a structure for code.

However, the behaviour models can not provide structure for object-oriented code generation hence the need to transform them to a class structure after they have been used to model the EAE domain of interest.

Our modelling approach to transforming EAE domain models to a class structure diagram comes from the need to generate OO code that reflects the domain of interest. Our approaches are automated and contain an output model (class diagrams) transformed from input models (activity and state diagram). We propose two approaches: a naive approach which does not preserve domain concepts, and a second approach that creates domain-relevant code structures, and makes use of more of the domain models.

As explained in Chapter 3, YCIL researchers [8, 13, 14] handcraft code for simulations based on UML diagrams validated by domain experts. According to [53], programmers often spend much time debugging code and the code often contains errors. Confidence in the implemented simulators could be impaired by an automated process of transforming domain models to generate object-oriented code such that the code demonstrably retains the validity of the models.

In this context, the hypothesis of this thesis is stated below:

Model management operations (validation, comparison and transformation) can be deployed to complex behaviour diagram thereby producing class structure artefacts that are reusable for simulators as in contrast to handcrafted code for simulator which often contain errors.

The objectives of this research is the aim of the transformation approaches that we develop and they should be capable of the following.

1. Reliably and systematically generate code from behaviour models.
2. Reliably and systematically generate revised code when models (diagrams) are modified.
3. Support reuse and modification of the existing models and simulations.

4.3 Research Scope

The scope of our research uses YCIL behaviour diagrams, and MDE tools especially EMF. The UML behaviour diagrams from Read's thesis [8] are used and they have slight notation variations to accomodate EAE biological components. EMF framework is also used to help generate object-oriented code via our developed modelling approaches. The reason for limiting our scope to MDE and using EMF is explained by [6] as "EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor" [6].

4.4 Research Methodology

An iterative and incremental development process is followed in evaluating the hypothesis. The behaviour diagram is analzyed, designed, implemented, and the output model is tested and evaluated iteratively and incrementally for each bodily compartments of EAE expected behaviour shown earlier in Figure 3.2 of Chapter 3. Figure 4.1 shows a form of iterative and incremental development process based on agile development methodology from [9].

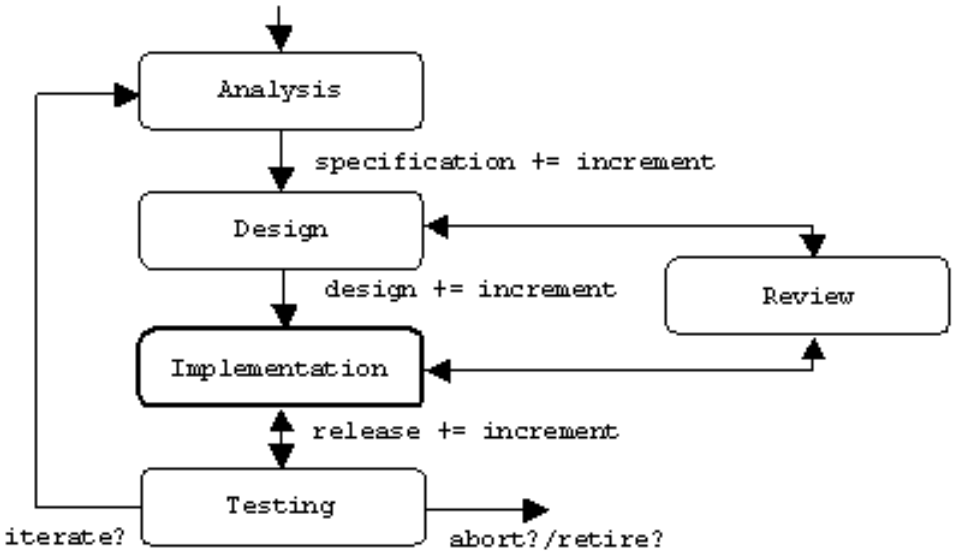


Figure 4.1: An illustration of an agile model-driven development showing iteration and incremental process from [9].

4.4.1 Analysis, Design and Implementation

The use of Read's behaviour model [8] allow us to explore model driven engineering thereby devising two approaches to transformation. These approaches are motivated by the need to automate more of our modelling processes and find an alternative to handcrafting codes. This in itself is one of the challenges identified in supporting simulators and it has motivate our hypothesis and objectives highlighted in Section 4.2.

The designing of our models enables the use of model as the first-class citizen. This process enables the development of model management operations on this model. The target output model during model implementation phase is object-oriented in nature thereby class structure diagrams are effectively designed following our approaches. Our approaches were implemented as a model transformation framework thereby simplifying the development of automatable model transformations that can be easily reused.

4.4.2 Research Model Approach

Our two approaches were deployed on Read's domain models and they help establish a model-to-model and model-to-text transformation where the input models (activity and state diagrams) are transformed to a class structure diagram and the structure diagram is used to generate Java object-oriented code. The development and adoption our approaches is rooted in their ability to enable us achieve the desired target (class structure diagram) and code for simulation.

4.5 Summary

This chapters discuss the motivation of our literature review. This thesis objectives and hypothesis were expressed and the intended methodology followed in order to explore MDE is reviewed. The two approaches developed for our model transformation is discussed in Chapter 5 and 6 .

Chapter 5

Naive Approach

This chapter discusses the naive approach developed to aid the model transformation in this research. The naive approach was inspired by work from [54]. The process and how the approach is automated is discussed in the sections below.

5.1 Introduction

The naive approach to model transformation presents a process of transforming an activity diagram to a class structure diagram. This approach is motivated by work from [54] which provides a systematic, but simple and intuitive translation to object-oriented code and it can be implemented with or without automation tool.

According to [54], "when translating activity diagrams to class diagrams each activity diagram will map to one class in the class diagram. Activity diagram in subactivity states are translated to aggregated classes. This is in accordance with the subactivity specified semantics which states that single activity graph may be invoked by many subactivity states".

Inspired by the concepts from [54], we proposed the naive approach which formalizes mappings between behaviour diagram (activity diagram) and a class structure. The mappings is done with the aid of metamodels and it helps remove any ambiguity on our naive approach. The motivation behind this approach is enumerated below.

1. If we assume an OO target (YCIL mostly uses Java Mason), then we need a class model to give the structure of the code.

2. We can extract objects and operations from activity diagram, and thus transform activity diagram to a draft class diagrams.

3. We can validate (and extend) the class model with more information by transforming the activity diagram to a sequence diagram and the draft class diagram to another sequence diagram. The two sequence diagram can be compared to-

gether for matching components and any resulting additional sequences can be used to update the class diagram.

5.2 Outline

Given the activity diagrams used by Read [8] to capture the behaviour of EAE, we can:

- Transform activity diagrams (AD) to class diagram (CD) - so as to get a object-oriented structure for code generation.
- Derive sequence diagram (SD) from AD - behaviour aligned with objects thereby giving a pointer to potential classes.
- Transform CD to SD - sequence diagram represents message sent between instances of classes (objects) therefore, when messages is sent between two objects, it implies that there is a relationship between the two class which must be shown on the class diagram.
- Compare SDs and update CD - so as to validate the correctness of the transformation and add other missing components noticed during comparison to the classes.
- Generate OO code from CD - so as to derive the code needed for simulation support.

This outline is better represented diagrammatically as shown in Figure 5.1.

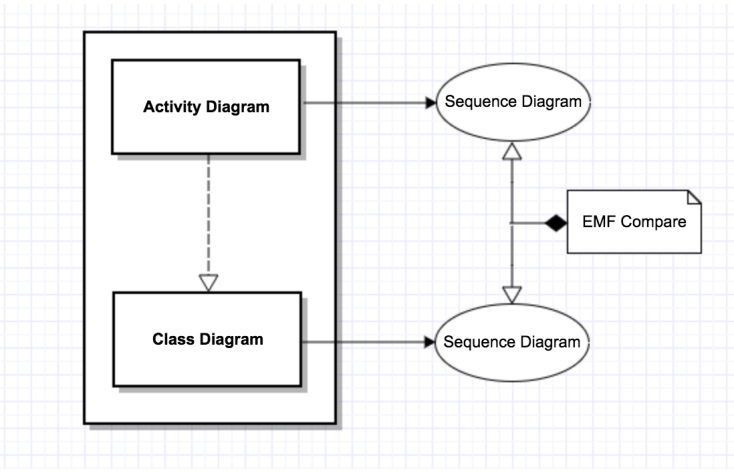


Figure 5.1: An illustration of the transformation steps from AD to CD, AD to a SD and CD to another SD. Comparing the two resulting SDs validate the CD, prior to generation of OO Java code.

5.3 Input Model - Activity Diagram

The discussion of this section raises issues of activity diagram (AD) and what EAE system did with it in relation to our approach. The UML AD defines behaviour as a flow of control using activity, sequences, conditions and state of an event [55]. A behaviour shows sequences of components that use a data flow model and controls [56].

AD is also defined as a specialized variant of state diagrams, a specialized diagram based on petri-net semantics so that it can be used in any given scenarios or domains with a wider scope [57]. Activity diagram consists of a parameterized behaviour denoted as flows of action. The flows of actions are shown as transition lines or control flow as they help connect one activity to the other. As a behaviour diagram, the AD captures the flow of events showing the behaviour of the EAE system. The activity diagrams used in this process are from Read domain model [8] and shown in Figure 3.3 and Figure 3.5 of Chapter 3.

5.3.1 Input Model Transformation Strategies

The transformation of our input model (activity diagram) entails its translation to other representative type (mostly, another model often called the output or target model). Prior to this transformation, the model is transformed to other artefacts that are closely related to the output model (class structure diagram). The closely related diagram adopted here is the sequence diagram (SD).

Sequence diagram is an interaction diagram that shows how an operation is implemented. It helps capture interactions between objects thereby showing their relationship. Also, it help model either a generic or specific instances of interactions between objects [58]. Sequence diagram depicts the order of interaction in vertical axis thereby showing when and what messages are sent while its horizontal axis shows the object instances to which the messages are sent. However, they do not represent objects structural interactions but are steps away by showing objects interactions [59].

Some of the messages being transferred between activity diagram are described below.

- Synchronous - a message that shows a "wait" semantics between objects; the sender waits for the message to be dealt with before job continues. It is a form of a method call [58].
- Asynchronous - a message that doesn't need explicit return message before job continues. It shows a "no-wait" semantics; the sender does not wait for the message to be handled before it continues thereby allowing objects to be implemented concurrently [58].
- Reply - a message that is a return from another message [58].

- Create - a message that occurs during the development of a new object. According to [58], an example is a message in Java that calls a constructor for a class.

Sequence diagrams use notations which includes frames, objects, lifelines, activations, messages, states and so on. Figure 5.2 shows different UML sequence diagram notations.

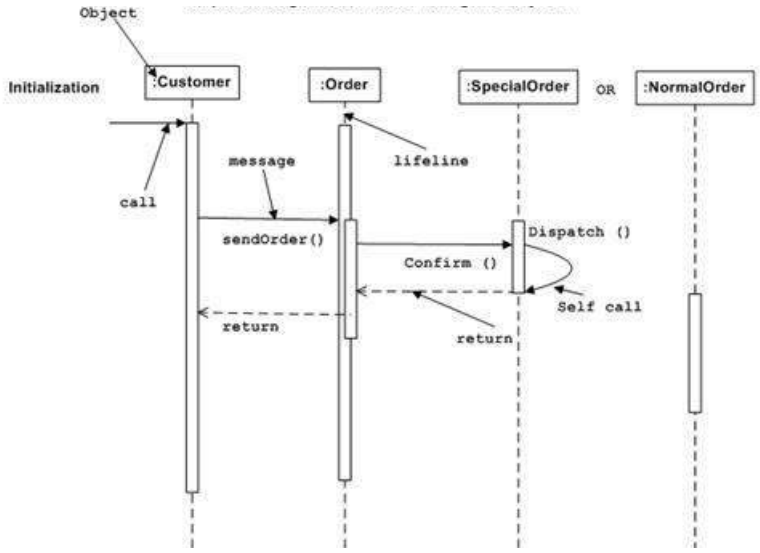


Figure 5.2: UML sequence diagram showing notations from [10].

Transformation from AD to SD

The main motivation for our transformation is to produce a more structural representation of behaviour (AD) that is aligned to objects. There exists a limitation on the AD as the generation of code from a behaviour diagram is very difficult. The objects identified with transformation to SD helps identify classes for the translation of AD to CD. Due to this variation and for the purpose of our transformation, we are able to resolve the activity diagram to a sequence diagram. We chose to transform AD to SD as other people have translated SDs to CDs. We also imagine an activity between two sequence as their interactions, hence an opportunity for transformation. Figure 5.3 shows AD components from Figure 3.3 and Figure 3.5 being transformed into a sequence diagram. Appendix A.1 shows the transformation rule used in transforming our AD to SD.

We take the following steps:

1. An action in an AD is represented as a lifeline in the sequence diagram (SD). The lifeline is an element representing each actor in an interaction while the action is an element representing each stage of an activity.

2. The transition lines or control flow between different activities are translated to asynchronous messages between lifelines of the SD.

3. The merge nodes are translated as synchronous message interaction because they bring in together alternative flows. The action here does not start until all the control flows are connected by the merge.

4. For a synchronous message interaction in the SD, the control flow or object flow between the activities and object must point to an if else logic of expression or decision nodes in the AD.

5. The ADs decision nodes and forks are translated as asynchronous message interaction as they show concurrent executions of two threads.

6. The joins are translated as a synchronous message interaction because the actions after the Join will not continue unless all the actions leading to the join is executed.

The AD joins and forks relate to input and output operations, respectively. The control flows link together to form a join hence the reason for translating AD joins manually to an output operation which is synchronous in nature. The fork releases a control flow hence an output operation which is translated asynchronously as the fork is not ordered to wait for the activity before continuing. Here, the synchronous message occurs as the sender waits until the receiver is done processing the message before continuing the jobs [60]. For asynchronous message, messages are sent without waiting for a response from the receiver before the job continues [60].

Transformation from AD to CD

To justify the transformation of AD to CD, all elements of the AD are transformed to CD elements through objects which is an instance of a class. The first step starts with mapping each activity to a class in CD. By following concepts adopted from [54], we exploit the following scenarios:

1. Activity in AD is mapped to a class object.

2. Action state is mapped to an action state object as all action state objects are regarded as action state vector - a class property.

3. Subactivity states are mapped as a subclass with a generalization relationship to its active class.

4. Transition line or control flow of activities are transformed to a relationship between classes.

5. Decision and merges are transformed to operations of the class.

6. Forks and joins are transformed to as an association relationship of the class.

The transition line is mapped as a relationship between the classes as it is triggered by the activity's action in relation to one activity from the other. Decision and merges have input and output which makes their mapping to a class method. The mapping considers simple transition and decision structure to support multiple possible flows [54].

Fork has two or more outputs in form of arcs or flows which depicts the "concurrent executions of threads" whereas the Join depicts the "synchronization of concurrent activities" [54]. For mapping to a class method, all input and output must be considered together. This gives basis for a relationship mapping between the classes. Following the steps stated above, we transformed AD from Figure 3.3 and Figure 3.5 to a CD, shown in Figure 5.4. Appendix A.2 shows the transformation rule used.

5.4 Output Model - Class Diagram

The output model of our naive approach is a class diagram and it provides the structure needed to generate OO code. A class diagram shows relationship between two or more classes and a SD shows how messages are sent between instances of these classes. Messages sent between objects implies a relationship between two classes hence the motivation for translating CD to SD. Here, our output model is shown in Figure 5.4.

5.4.1 Transform CD to SD

For the purpose of validating the translation of AD to CD, a sequence diagram that represents the model of our class diagram is transformed. The classes in our class diagram is converted into lifelines. The edges are translated to synchronous message and asynchronous messages depending on their classifier and associations. Following on from the point above - a lifeline refers to an object. A SD can have more than one object for each class.

Also, the messages become the operations and property of the class diagram as they are conceptually a form of operation calls. Figure 5.5 shows the transformation of CD to SD achieved through this strategy as we develop our SD by using the components of our CD. The structure of the model is not altered with the update from compared sequence diagrams. The following steps are proposed:

1. The classes (superclass and subclass) in the CD are represented as Lifelines in the SD. The class may contain many objects and the lifelines is an object.
2. Properties are transformed to a synchronous message
3. Operation calls of the CD are translated as asynchronous messages.
4. Relationships are transformed to a create message of the SD.

Most importantly, the two sequence diagrams derived from the activity and the class diagrams, show the interactions and sequence of events culminating into a detailed model of EAE operations and their associations as evident in the domain model [61]. We automate the listed transformation rules using the eclipse tools. Appendix A.3 shows the transformation rule used in transforming our CD to SD. The design developed here is justified as a means to comparison of two sequence diagram so as to vindicate our transformation of the CD from the AD.

5.5 Process Automation

This section describes how the naive approach steps can be automated. The purpose of automating the transformation rules for our models is to provide test cases and effective modelling of the manually designed UML diagram representing different abstraction level of EAE system. We use two open source tools, as follows.

5.5.1 Papyrus Tool

Papyrus as discussed in section 2.2.2, is an open source model-based engineering tool built on Eclipse. Papyrus provides an integrated environment where UML diagram can be created and edited. It can be easily customized to provide support for diagram notations and for manipulating models. We use papyrus to draw the AD and SD. Also, the manual translation of SD to CD is supported by the tool by providing sequence diagram components. Essentially, Papyrus assists in automating the different UML diagrams representing the different level of abstraction expressed as a behaviour model in [8]. Papyrus components contains primitive types for both Java and Ecore which can be used to generate Java code or Ecore metamodel respectively. Additionally, it enables automation of transformation steps of the naive approach by providing UML drawing components. It also provides the diagram XMI which can be exported. Figure 5.6 shows part of EAE class being created with papyrus tool.

5.5.2 EMF Compare

Eclipse Modelling Framework (EMF) Compare [1] support merge and model comparison. It provides customizable, reusable and generic tool support for comparing and merging model. We use EMF Compare for merging and comparison of the models created using the Papyrus. With EMF Compare, a merging of model components gives confidence in the model and the code to be generated when rules used are stated and verified explicitly. The comparison of our derived SDs is done using EMF compare. This comparison authenticates the translated CD and the code generated from it. Also, it provides a visualization of the compared entity either as a

[1] https://www.eclipse.org/emf/compare/

text compare or a model compare. We did not use other alternative comparison tool such as JEdit JDiff[2] and Guiffy SureMerge[3] as we don't have detailed knowledge on their utilization. Figure 5.7 shows an EMF compare in a model compare format.

Comparison

We sought to compare the derived SDs so as to validate the translated CD which in turn affords us the confidence on the generated code. For comparison, we use the two derived SDs. The first SD is derived from an AD while the second SD is from the translated CD. Most importantly, the two sequence diagrams represent the same model in our domain of interest. This relationship gives us confidence in the correctness of the translation between our AD to the CD. We compare the two sequence diagrams using the eclipse tool, EMF Compare[4]. EMF Compare is a tool designed to support comparisons of large fragmented models [62]. If the matching of the merged models shows similarity between the components of the model, there exist a relationship between the activity diagram and class diagram. The differences are also shown and any inaccuracy of the models is highlighted. The comparison done between the two sequence diagram shows relatively many matching components hence the confidence on the approach and the differences is used to update the translated class diagram so as to have a fit-for-purpose output model.

Figure 5.8 shows comparison of another YCIL work from [14]. The naive approach is deployed to [14] domain, where "extracellular stimuli binds to the cell membrane receptor and sequentially moves through each of the steps before transcription of inflammatory genes and resultant translation into inflammatory response proteins, culminating in inhibition of NF-B again by IB" [14].

The effectiveness of this approach to our case study [8] is reflected in the updated CD. This approach deciphers a practical and intuitive way to translate Read's domain model [8] to class diagrams with generated OO code. The domain model [8] is transformed to both interaction and structural diagram without compromising or adding additional information outside its scope. This approach enables easy re-usability of models for their instance comparisons. More importantly, these technique helps reduce the loss of crucial information from the domain model as information that must have been left out during AD transformation to SD is updated through informations captured from AD transformation to CD and the subsequent SD.

5.6 Code Generation

With the automation of the transformation process using papyrus tool, we move to comparing the two sequence diagrams using EMF Compare so as to validate

[2]http://plugins.jedit.org/plugins/?JDiffPlugin
[3]http://www.guiffy.com/
[4]https://www.eclipse.org/emf/compare/

and update the class structure diagram. Here, we proceed to generating Java code from the validated class diagram. Here, our papyrus tool enables us to add Java primitive types to our class, attributes, property and operations. This process of adding primitive types allow us to generate the OO code. The code generated reflects the class structure diagram when the Java primitive types are added. Listing 5.1 shows code generated for the SLO class in Figure 5.4.

Listing 5.1: A sample Java code generated from the class model in Figure 5.4.

```
1  public class SLO {
2
3   /**
4    *
5    */
6   public CD4thCellBindsMHC cd4thcellbindsmhc;
7   /**
8    *
9    */
10  public DCSecretesCytokines dcsecretescytokines;
11  /**
12   * Getter of cd4thcellbindsmhc
13   */
14  public CD4thCellBindsMHC getCd4thcellbindsmhc() {
15     return cd4thcellbindsmhc;
16  }
17  /**
18   * Setter of cd4thcellbindsmhc
19   */
20  public void setCd4thcellbindsmhc(CD4thCellBindsMHC
       cd4thcellbindsmhc) {
21    this.cd4thcellbindsmhc = cd4thcellbindsmhc;
22  }
23  /**
24   * Getter of dcsecretescytokines
25   */
26  public DCSecretesCytokines getDcsecretescytokines() {
27     return dcsecretescytokines;
28  }
29  /**
30   * Setter of dcsecretescytokines
31   */
32  public void setDcsecretescytokines(DCSecretesCytokines
       dcsecretescytokines) {
33    this.dcsecretescytokines = dcsecretescytokines;
34  }
35  /**
36   *
37   * @return
```

```
38     */
39    public String JoinOperation() {
40     // TODO Auto-generated method
41     return null;
42     }
43    /**
44     *
45     * @param CD4ThBindsMHC
46     * @param DCSecretesCytokines
47     */
48    protected void ForkOperation(String CD4ThBindsMHC, String
        DCSecretesCytokines) {
49     // TODO Auto-generated method
50     }
51
52  }
```

5.7 Evaluation and Critique

This section evaluates the naive approach, with respect to the research hypothesis: *transformation of behaviour diagram to a structure diagram using MDE from a naive approach and an extended practical approach.*

The evaluation of the naive approach transformations uses Read's UML activity diagrams [8] as the input model. As described in Section 1.3, this is our first approach deployed towards transformation therefore it is naive and presents the extraction of class diagram from an activity diagram solely.

5.7.1 Evaluating Correctness and Target-Realization

The reliability of our naive approach is viewed based on its correctness and target-realization culminating into a fitness-for-purpose model transformation. An incorruptible method of validating the naive approach is making sure every component presented in the activity diagram is reflected in the transformed class diagram - a model to model transformation.

Furthermore, the transformed class diagrams is validated through code generation by adding Java primitive types with the aid of papyrus tool. We evaluate the validity of our naive approach in the sections below.

Target-Realization

The naive-approach follows modelling based on the presented activity diagram. Target realization is the goal of achieving sets of outputs that reflects the actual EAE's domain of interest. The target-realization for the naive-approach seeks to produce

an output of sequence diagrams that can be compared together to ensure verification of the translated class diagram as illustrated in Figure 5.1. This verification is done by analyzing whether the process for transformation satisfies all the transformation processes laid down in this Chapter for our transformation.

More importantly, the translation of activity component to a class diagram and the further extraction of sequence diagrams from both activity diagram enables guided and effective transformation based on the available input model (activity diagram).

Furthermore, to assess target-realization in our naive approach, we make sure our input model uses the activity diagram and other textual information provided in Read's domain model [8] for transformation from AD to SD and AD to CD as well. Our ultimate realized target is a class structure diagram and a negation of this approach comes from the incorrect output (sequence diagram) from the activity diagram being entirely different from the sequence diagram translated from the class diagram. Our premise to target realization raises the question below.

Q1: Are the contents of the two sequence diagrams when compared shows relatively matching components?

Correctness

The outcome of our naive approach is the transformation and generation of class structure diagram and object-oriented code respectively. Evaluating the correctness of our naive approach involves getting fit-for-purpose object-oriented code. The correctness is validated with the support of papyrus tool for generation of required artefacts (code). The papyrus tool enables us to add Java primitive types which aid the generation of Java object oriented code from the class structural diagram.

To evaluate the correctness of the code as an implementation of the model, a structural testing comparing the objects and classes presented in the code is done in relations to the transformed model. The code generated has not been compared to the handcrafted code from [8], however, following our naive approach, we have confidence on the code generated in relation to the scope of the transformation as primitive types added reflects the structure of the output model in a Java OO code environment. Our premise to correctness answers the question below on the premise that using Papyrus tool to add Java primitive type, there is no change in the structure of the output model as it only give basis to generation of only Java code from the CD.

Q2: Does the code generated reflect element of the class structure diagram when Java primitive types are added?

5.7.2 Evaluating Efficiency

Our research hypothesis aims to use MDE to explore behaviour diagrams through model management operations (model transformation) hence the need for the exe-

cution of models in a very efficient, effective and consistent manner. Efficiency and consistency is a mainstay in our naive approach as the output models (structure diagram) needs to reflect all components of the activity diagram by following inspired concepts from [54] used in developing the proposed transformation steps of this chapter. This approach is efficient as it avoids the use of complex constraints declared in constraints languages like EVL.

The naive approach consistency comes from its usability in any provided activity diagram without changing any of the proposed transformation steps regardless of whether the components of the input model is altered or not. It also utilizes papyrus inbuilt validation which is repeatable through the use of EMF Compare and this produces an inbuilt quality for the approach. This approch allow us to use more YCIL case study diagram [8, 13, 14] thereby enabling approach testing. For evaluation purpose, we answer the following questions on our approach efficiency and consistency:

Q3: Will the output model structure be altered with the increasing update of the extracted class diagram from the compared sequence diagrams?

Q4: Under what situation will the translation to output model be found efficient in relation to the increased numbers of input models?

5.7.3 Case study

The sections above highlighted essential qualities that can be used to assess the effectiveness of our naive approach and described how they are evaluated. To understand our naive approach in depth, we discuss our case study by answering the questions raised (Q1-Q4).

The scope of our transformations are dependent on Read's domain model [8]. The activity diagram gives basis for the transformation hence the corresponding result (CD and OO code) reflects its element. The CD is validated via the transformed SDs which leads to the generation of Java OO code.

The transformation of AD to SD, AD to CD and CD to SD hinges on the transformation rules and these transformations evolved independent of our techniques. The naive approach technique serves as a framework for how our final result can be achieved. The translation of AD to several models (SD and CD) uses only the element presented in the domain model [8]. This helps avoid bias and shows that the naive approach techniques presented have not been optimised for our case study [8].

The code generated reflects element of the transformed CD and the primitive types added helps with generation of Java code. There is no alteration to the structure of the CD as the papyrus tool supports code generation based on the output model (CD). The results from the naive approach are limited to the case study and the papyrus tool used for drawing the transformed model with subsequent code generation.

The correctness of the approach has not been verified as we don't have access

to the hand-crafted code of the case study [8], however, we have confidence on our transformation process as they are fit-for-purpose based on ETL transformation rule. Precisely, the premise for our result verification lies in the transformation process as elements of the domain model makes up the code element and the tool deployed for code generation guarantees the generation of OO code.

5.8 Critique

A situation where a model output is expected to be correct or not is a common assumption in software testing and this is used as a yardstick for determining the correctness of a system [63]. In testing the correctness of our approach, the same scenario is played out in our model transformation format as we assume the approach used produces a correct model or not depending on the modellers' skill in relation to the stated transformation process.

Here, the limitations of our naive approach is discussed in respect to its practicality towards EAE behaviour models. The critique of this approach is the limitations on comparison as it is a tool-oriented and it only compares the input data (the two sequence diagrams) rather than further testing on other diagrams which are not part of the input model.

The naive approach loses traceability from biology components to the generated code, as we work on the model class structure. The transformation use the notations not the semantics of the activity diagram.

Domain experts (biologist) need more than one model to express the domain. However, code generated from the naive approach only expresses details from an activity diagram. Different features of cell in EAE compartment will be difficult to find in the code even with regeneration by biologist as the activity diagram expresses just the role of actors in the system instead of other details.

As stated above, our naive approach does not use all domain model of Read's EAE system [8]. Furthermore, it will be possible to generate code from more diagrams but it won't solve its biological correspondence needed for effective modelling of EAE model. This actions put limitations on the feasible deployment of our naive-approach at large, to EAE system. This leads to the development of a more solid approach presented in Chapter 6.

5.9 Summary

This chapter introduces and motivate the naive approach for transformation. Transformation processes inspired by work from [54] is also proposed. The tools used in automating this approach is discussed. The automation of the transformation process and the tool used is reviewed. Code generation as a means to further simulation in future work is also discussed. We further evaluate and critique the naive approach so as to enlist its benefit and limitations which motivates us to the second approach.

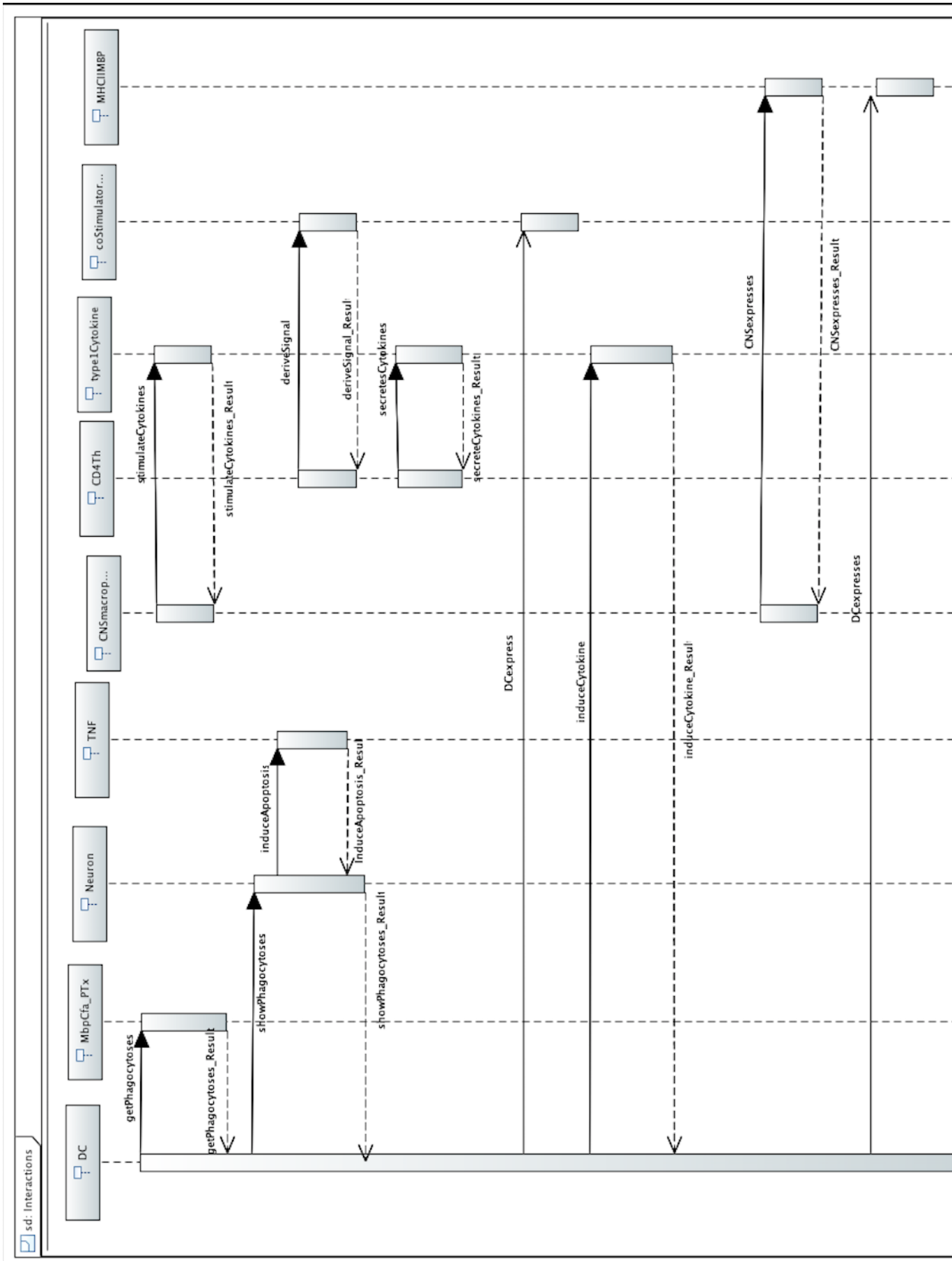


Figure 5.3: Sequence diagram transformed from the activity diagram in Figure 3.3.

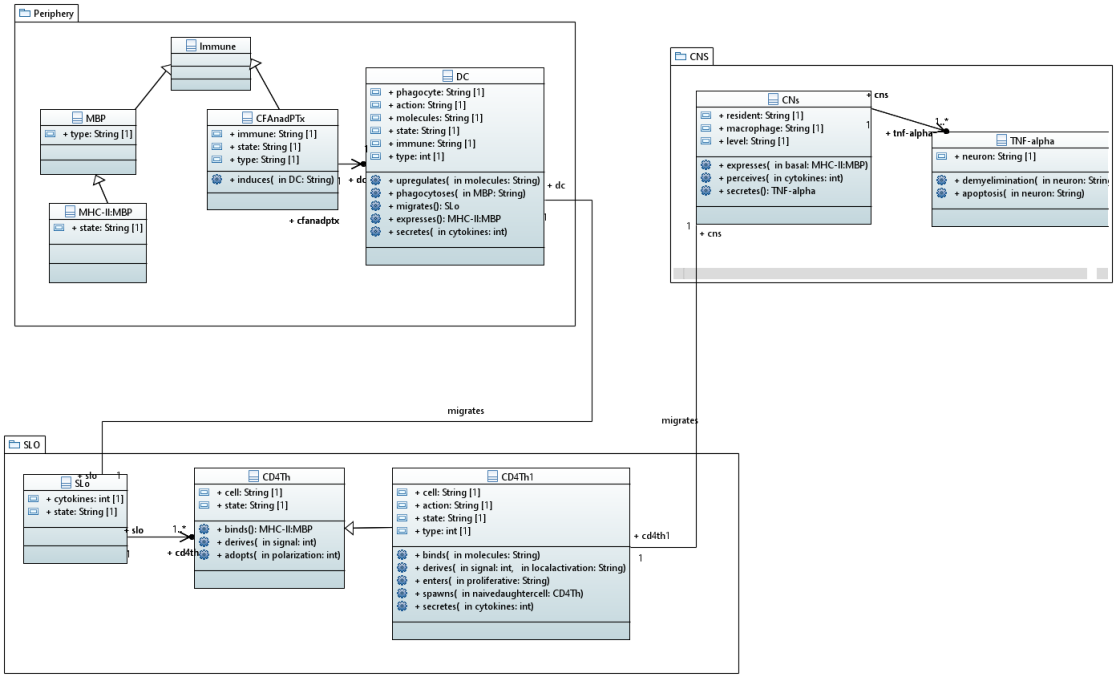


Figure 5.4: A part of EAE transformed class structure model.

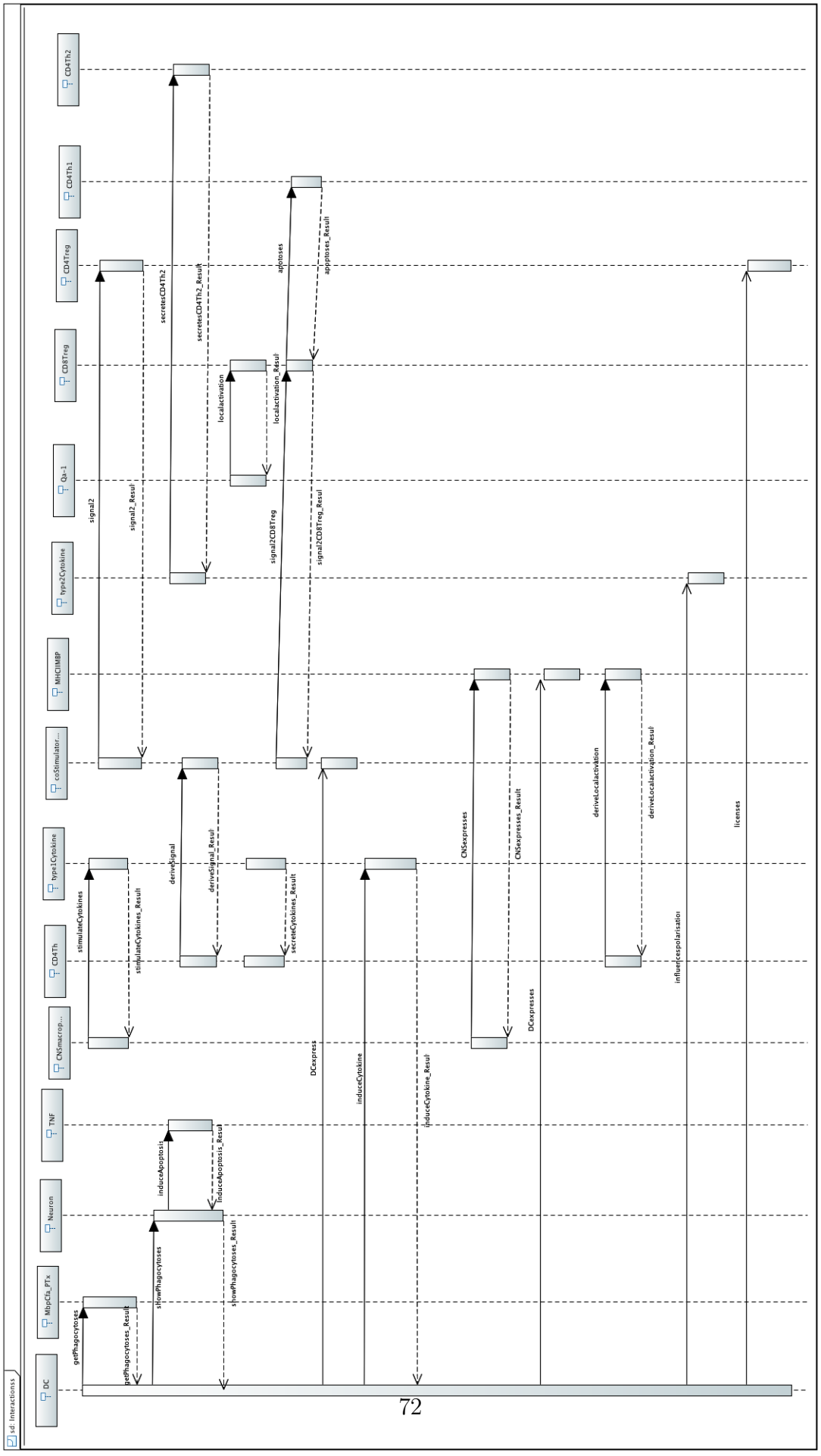


Figure 5.5: Sequence diagram showing interactions between objects of class diagram presented in Figure 5.4.

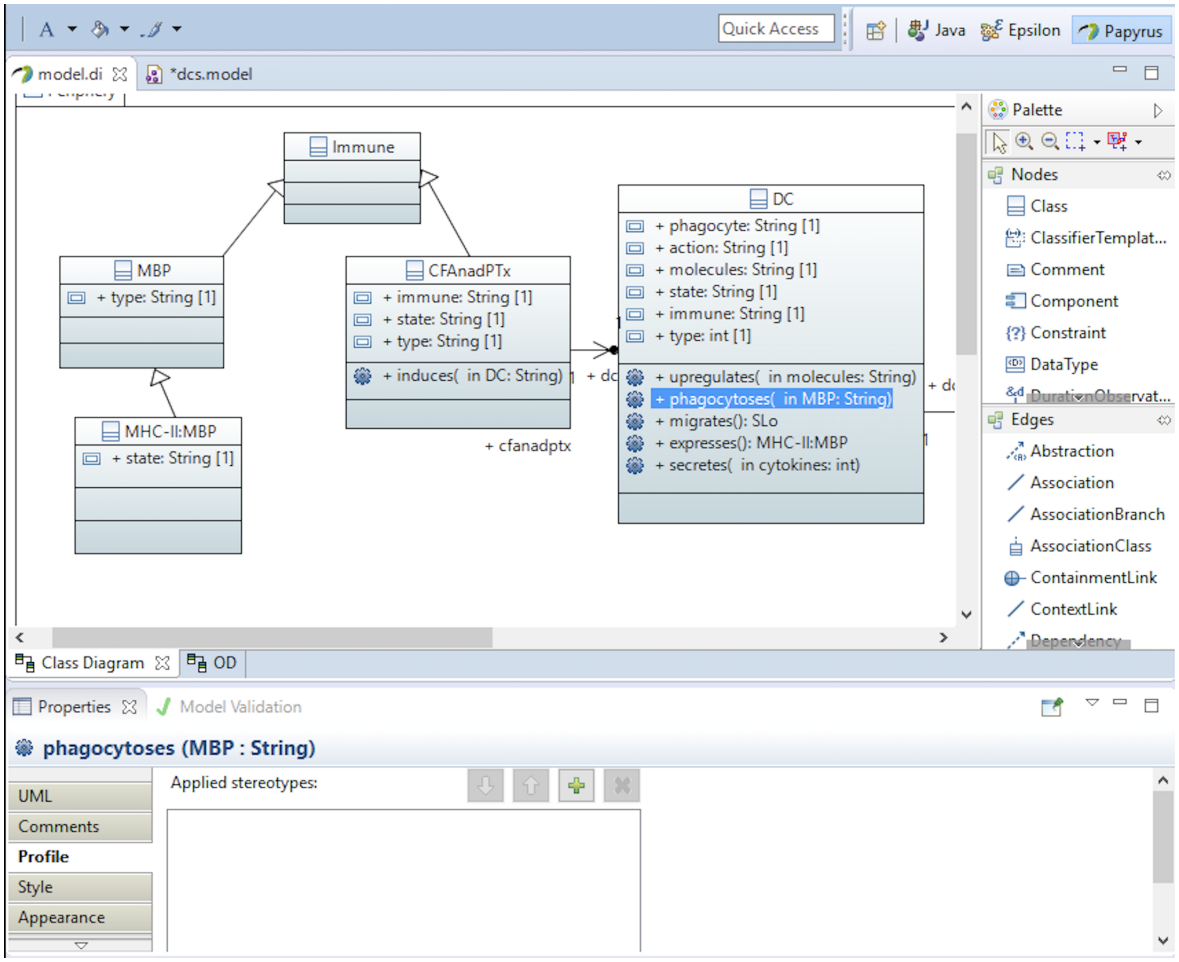


Figure 5.6: Creating a model in Papyrus, illustrating some derived EAE class structures.

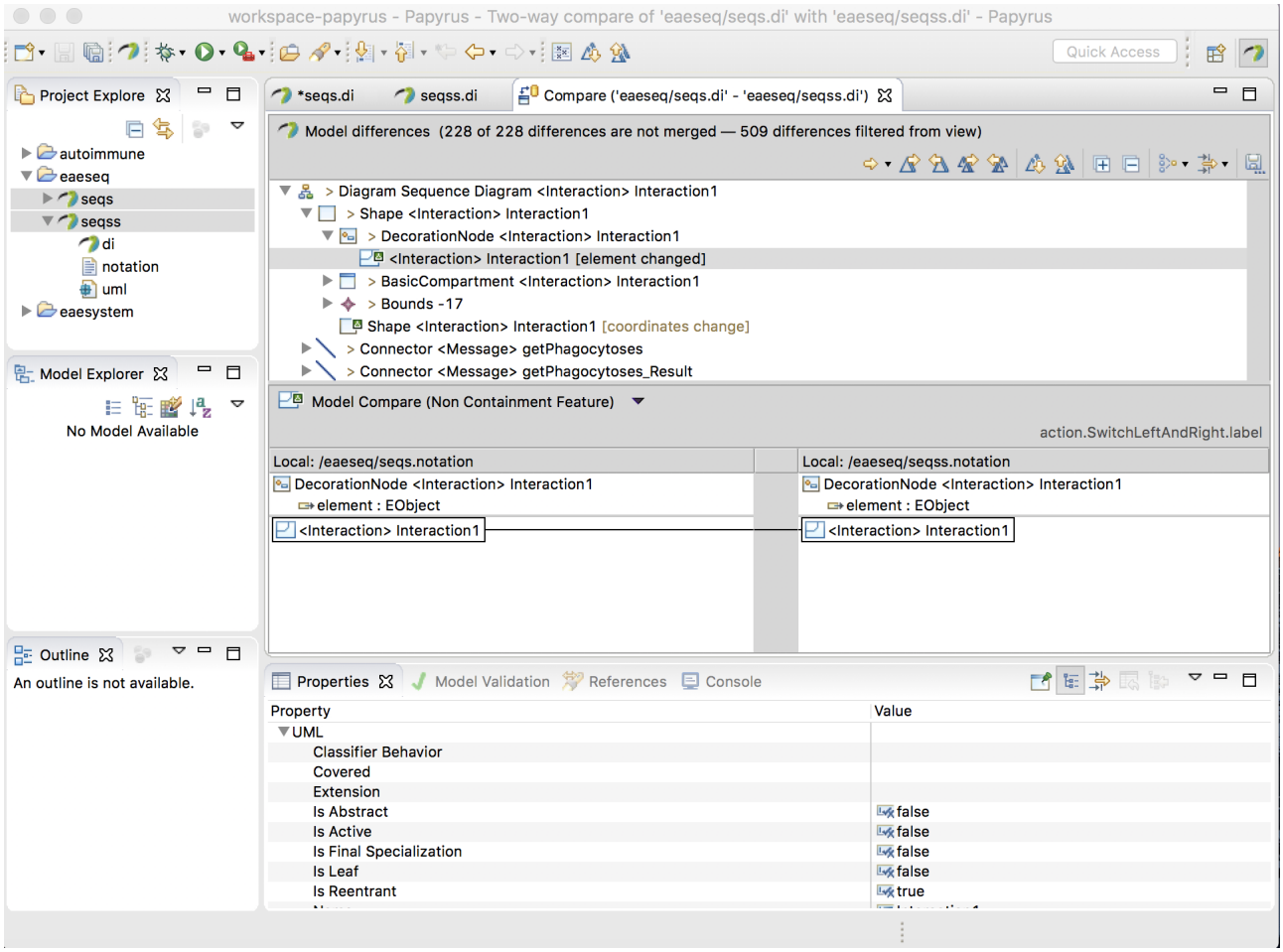


Figure 5.7: A comparison of our two sequence diagrams showing differences in merging.

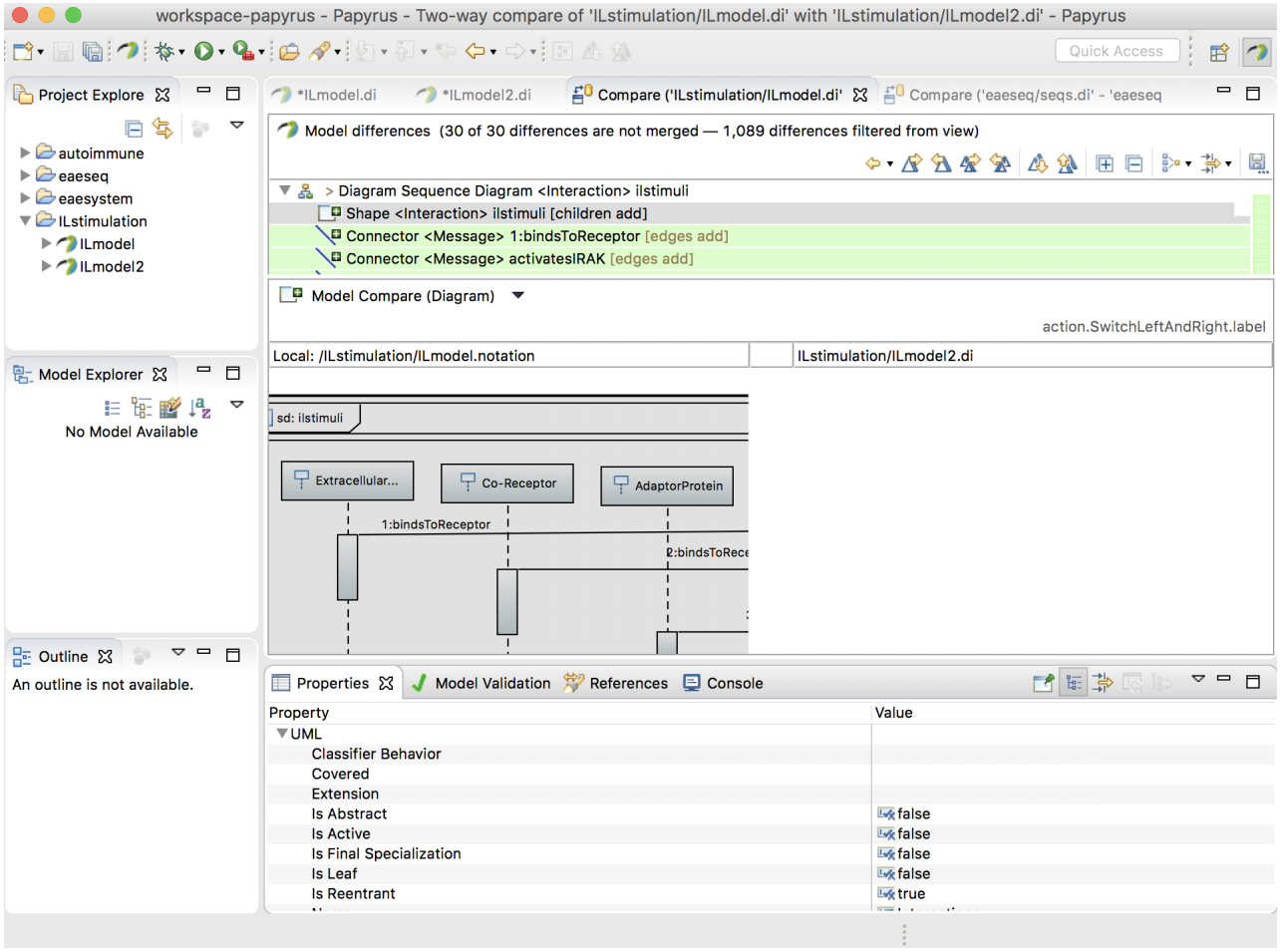


Figure 5.8: A comparison between two sequence diagrams.

Chapter 6

Second Approach

The second approach to modelling uses all the behaviour diagrams (activity diagrams and state diagrams) and other information presented in Read's thesis [8]. This approach is sophisticated as in contrast to the naive approach where only activity diagram is used as the input model for transformation. In this chapter, we discuss our approach, tools used, and its evaluation in general.

6.1 Introduction

The second approach is based on application of MDE principles, and it is implemented through an incremental process of model development. With the need to generate code from the structure of a model diagram for further simulation, it is hard to change only part of the code that relates to a biological concept. This calls for the need to have a concrete approach that reflects domain concepts of the behaviour diagram to the structural diagram. Towards the development of a structural diagram, metamodel can be developed to help with the modelling of EAE system.

The approach to transformation is broken down into three stages. The first stage utilizes several state diagrams presented in Chapter 3 by developing a metamodel for them. The second stage deals with model refinement by combining the activity diagrams and other information presented in Read's domain model [8] with the state diagram metamodel to create a target metamodel (a super metamodel capturing the whole domain model [8]). The third stage deals with the use of MDE practices to support these models. This is done by applying several model management operations thereby manipulating the models to generate an OO Java code.

6.2 Overview

The source model used in our second approach is the various state diagrams shown in Chapter 3. A metamodel is defined for the presented state diagrams. This component view of the system captures the detailed information of EAE, and supplements

the activity diagram that shows the top-level system activities of EAE. A state diagram shows the possible behaviour of any object of a class and we are provided with several state diagrams as discussed in Chapter 3. We decide to create a metamodel for these several diagram so as to have a unifying structural model that defines each individual state diagrams. This is achievable as each state diagrams presented by [8] is for a domain class of EAE.

The several state diagrams presented by Read [8] are used to define a domain metamodel which encompasses all the domain concepts expressed in the state diagram elements. The definition of the metamodel will enable a M2M transformation from a source model to a target model. Here, the states in the state diagrams denote the existence of classes while the conditions and transitions indicate the existence of properties and relationships. Also, following the concepts of transforming AD to CD in Section 5.3.1 (Naive approach), we refine our target metamodel with other textual information from Read's domain model [8].

Our second approach is conceptualized on metamodelling through transformation. Here, model transformation enables the re-use of existing state diagram to create a metamodel from which object oriented Java code can be generated. The transformation is done on a domain level as this type of transformation ensures that the target model conforms to its metamodel specifications. This gives way to a syntactically fit-for-purpose model transformation. Our approach is illustrated in Figure 6.1. The approach shows the development of a metamodel for the state diagram following OMG's standard [3] and this metamodel is transformed to a target metamodel premised with refinement from activity diagram and other textual information provided in the domain model [8].

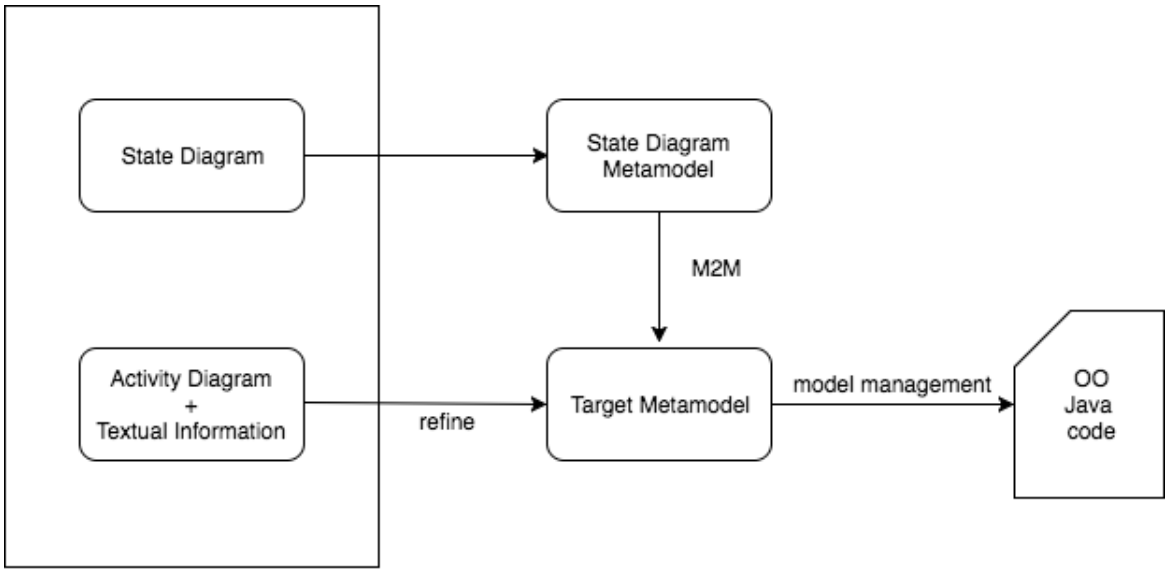


Figure 6.1: An illustration of our second approach showing its transformation process.

6.3 Source Model - State Diagram

The state diagram implies the existence of a class. The concept of state diagram is to define a system with a number of state which gives set of values to attributes of an object. This gives basis to the specification of individual behaviour entities of EAE system such as class objects. Also, the state diagram of a class shows transitions from one state to another in objects of that class. These transitions imply a call to one or more class methods that define how an object of the class changes states.

A state in a state diagram is defined by the values of the attributes in a particular object of the class. Given a class with an attribute, the state of the class is defined by the value of its attribute under a defined condition i.e. the conditions on the state transitions and the definition of the state requires an attribute, hence the transformation to a class structure attributes.

Read [8] uses state diagrams to model the dynamics of EAE cells and how they transition from one state to another. Figure 6.2 presents an example of UML state diagrams showing its notations. In order to perform a model transformation, the source metamodel and target metamodel is defined. The state diagram used from Read's domain are presented in Figures (3.9, 3.10, 3.11, 3.12, 3.13 and 3.14) and they are modelled explicitly to create a single state diagram metamodel. This metamodel is further transformed to our target metamodel which is a structural diagram.

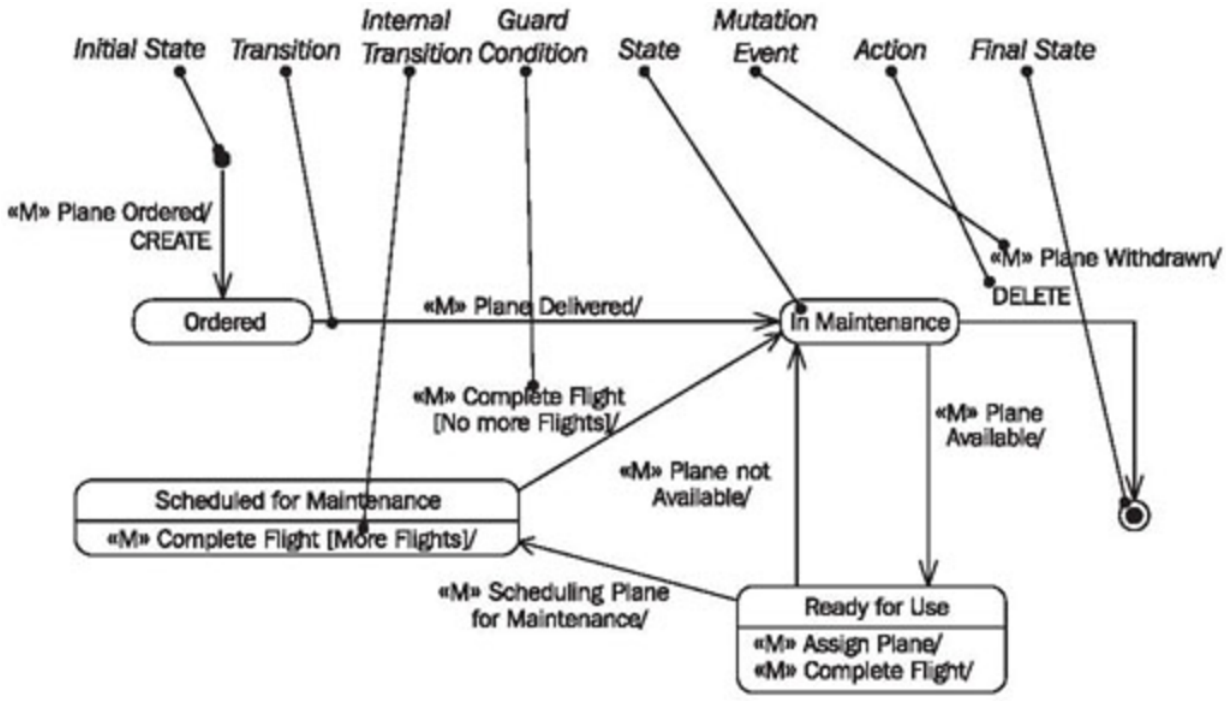


Figure 6.2: A state machine diagram showing its notations from [11].

6.3.1 State diagrams Metamodel

We present our EAE state diagram metamodel as shown in Figure 6.3 based on OMG's MOF specification [3]. The concept of MOF and its associations can be defined by a class diagram and since the state represents object of the class we support the metamodel with an ETL language for its development. The ETL rule in Appendix B.1 is used as an algorithm-like rule to support the development of the metamodel. The metamodel comprises elements of the six different state diagrams presented in Chapter 3. The state diagrams are for the single entity dynamics of the EAE system. The dynamics presented are the T-cell dynamics (CD4Th and CD4Treg), DC and CNSmacrophage dynamics as well as the Neurons and MBP dynamics.

Event of the state diagram triggers a transition between the states. The transition from one state to another is influenced by an external event. The internal event doesn't lead to change of state in the model. Actions of the events leads to methods. Here, the metamodel captures crucial and important element of the EAE state diagrams and it is defined in Ecore with the aid of an EMF editor. The purpose of developing this metamodel is to facilitate a M2M transformation to a structural model which provides an avenue to code generation. We take the following point into consideration during the development of the metamodel:

- The state diagram elements are representation of state classes.
- Transition consists of parameters and conditions that trigger events (internal and external events) and is dependent on the activity it performs.
- Activities performed by the state diagram transition are reflections of relationships and properties of the state.
- Transition from a state to another is an external transition while the internal transition does not reflect a change of state as they are predefined (entry, do, on event and exit).
- Transition executes actions or activities that are transformed into operations or class methods.

6.4 Target Metamodel - Structural diagram

The target metamodel is the final output model and it is shown in Figure 6.4. EAE models represents its real world scenarios, as such, there is a need to have a fit-for-purpose model representing its domain [8]. To achieve this, metamodels are used as a guide to its development. Here, elements of the source metamodel (state diagram metamodel) is transformed to the target metamodel through the use of model transformation. The state diagram's metamodel is used as the input model (Figure 6.3). Activity diagram and other information provided is used to refine the

metamodel and this is specified using ETL rule that transforms element of the AD to the target metamodels. By following the written rules, the target metamodel is produced for our EAE model. Appendix B.2 shows how different elements of the source metamodel as well as the AD are mapped to the target metamodel.

It is important to note that only elements that are functionally involved in object interactions are transformed to our target metamodel as it's the means of reflecting EAE domain model accurately. Figure 6.4 shows the transformed metamodel after mapping and refinement.

After the development of the target metamodel, model management operations are performed on it so as to extract, query and validate the model thereby supporting code generation. The model management operations performed are discussed in the next section.

6.5 MDE on Target Metamodel

This section describes the use of MDE practices on the target metamodel so as to generate a model which model management operations can be performed on.

6.5.1 Using Eclipse Modeling Framework to Model Target Metamodel

EMF as described in Section 2.2.1 is an Eclipse plug-ins tool which is deployed to help model our EAE target metamodel thereby generating artefacts (model and other textual artefact). The reason for using EMF is because it allows us to generate an *ecore* file (which contains defined classes) where *genmodel* file can be generated subsequently and it contains object-oriented code. Figure 6.5 shows our EAE target metamodel written in EMF.

Modelling Process

Ecore in the EMF framework perspective is used to create our metamodel. The file is written in EMF with *.emf* extension and this is where our EMF Ecore metamodel is generated from automatically. The metamdel allows for the development of model that conforms to it with the aid of the framework. The process is detailed below:

1. Create a new file with **.emf** extension and populate it with Emfatic syntax that reflects the target metamodel .
2. Generate a **.ecore** XMI-based metamodel from the Emfatic textual representation (**.emf** file).
3. Register the newly generated **.ecore** Epackages so as for the EMF to know it exist.

4. Create a model that conforms to the new **.ecore** metamodel.

Through this process, a model that represents our EAE systems and conforms to its metamodel is developed. Individual models from the metamodel is created automatically with the aid of EMF framework. The reason for creating the model is to explore individual models in relation to their domain model. After developing the model, model management operation is initiated to enable full-fledged MDE practices. Figure 6.6 shows the model of EAE class CNSmacrophage created from the target metamodel.

6.6 Using Model Management and Epsilon to Query and Validate Model

EAE model and metamodels are implemented by following the modelling process detailed in this section. The manipulation of these models is done through model management operations. This operation enable us to query EAE models, validate them against complex constraint and thereby generate Java OO code from them as well. These model managements were carried out using Epsilon tools and for the purpose of eliciting our model, the model management operations performed on them are further discussed in subsequent subsections.

6.6.1 Epsilon Object Language (EOL) to Query our Model

EOL is a crucial programming language that allows the development and query of EMF models [65]. It provides several reusable elements that can be utilized towards development of different task-specific languages. EOL is the main constituent or core of Epsilon an its an OCL-based language. Access to multiple models, user interactions and the development of conventional programming structure is made possible through EOL [66].

The use of EOL for our EAE metamodel enables us to write queries, create sample models that conform to the earlier developed metamodel and also run the queries on the new sample models. The created model is queried based on the conditions noticed in the domain model. For example, some of the TCell model are either in apoptosis or effector state so the model is queried to determine its state. Listing 6.1 shows different EOL language used in querying different part of our model.

Listing 6.1: A sample EOL language

```
1 //collect all effector state of TCell
2 TCell.all.collect(t|t.effector).
3   asSet().println();
4   var effector : Set;
5   for (t in TCell.all)
```

```
6   {
7     effector.add(t.effector);
8   }
9  effector.println();
10
11 //list of mature DCs
12 DC.all.collect(d|d.mature).sum().
13  println();
14  for (item in TCell.all)
15  {
16   item.effector.println();
17 }
18
19 //collect all TCells location and their probability
20 TCell.all.collect(t|t.migrate).asSet().
21
22 collect(t|t.probability).print();
```

After manipulating the EAE model with EOL, we proceed to validating it using EVL in the next section

6.6.2 Epsilon Validation Language (EVL) to Validate our Model

EVL is a language used for validation and it is built on top of EOL. It supports creation of constraints on models thereby evaluating inter-model constraints [67]. We deploy EVL to validate our EAE model and it enables us to identify and validate elements which violate our constraints. Also, EVL is used as a means to capturing the complexity of validation rules that the metamodel cannot express. For example, each different TCell are present in different locations. We use EVL to validate or check location of the TCell. Listing 6.2 shows several EVL language used in validating different part of our model.

Listing 6.2: A sample EVL language

```
1  context eaeimmune!TCell{
2
3  //Existing TCell must belong to a location
4  constraint HasLocation {
5
6   check : TCell.location <>``""
7
8   message : ``Location " + self + " exists"
9
10    }
11 }
12
```

```
13 context eaeimmune!CD4Th1 {
14
15 //Every CD4Th1 must have a state
16 constraint HasState {
17
18 check : self.state <> ``''
19
20 message : ``CD4Th '' + self + `` must have a state''
21
22  }
23 }
24
25 context eaeimmune!TCell{
26
27  //TCell must belong to a location
28
29  constraint HasLocation {
30
31  check : TCell.location <>``''
32
33  message : ``Location '' + self + `` exists''
34
35   }
36 }
37
38 context eaeimmune!CNSmacrophage {
39
40 //CNSmacrophage action is exhibited when it either secretes
      or express MHCIIMBP or TNFalpha
41
42 critique CNSmacrophageaction {
43
44 check : MHCIIMBP.allInstances.exists(m|m.type = self) or
45 TNFalpha.allInstances.exists(t|t.type = self) or
46 CNSmacrophage.allInstances.exists(c|c.`extends` = self)
47
48 message : ``CNSmacrophage action '' + self.mature
49
50 fix {
51 title : ``Delete CNSmacrophage '' + self.mature
52
53 do {
54
55 delete self;
56
57    }
```

```
58     }
59   }
60 }
```

After resolving the constraints and validating the TCell model, we proceed to generating Java OO code from the model.

6.7 Code Generation

With a fit-for-purpose model generated, we proceed to generating OO Java code from the model. The purpose of generating the Java code is to provide an object-oriented code structure which can be used in future as basis for simulation. The premise for generating the code lies in the correctness of our target metamodel. The metamodel developed captures elements of the domain model [8] through the use of ETL rule for transformation.

As expressed earlier, the developed metamodel followed OMG's standard [3] and it includes all aspects of the model. The validity of the obtained metamodel is justified in the domain model as the Java classes of the code generated reflects the domain classes of our EAE system. The class properties are reflected in fields of the Java class and the association reflects objects or collection of objects depending on the properties multiplicity. The Java class methods is evident in the domain's actions triggered by an event.

The code is generated automatically by the use of EMF tool. To achieve this, by right clicking on the root node of the *.ecore* file, we move to Eugenia which leads us to generating our EMF GenEditor. The generated editor is in *.genmodel* extension. Right-clicking on the GenEditor node enable us to generate all possible code with 'Generate All' i.e. edit, editor and test code. The code generated are Java code of EAE's EMF model implementation. By clicking on the edit code, Java code representing EAE class is available. Figure 6.7 shows the generated Java Code of EAE CNS macrophage class.

6.8 Simulation

The practicality of the generated code is rooted in its guide towards the actual simulation done by [8]. EAE's domain model is a project of the York Computation Immunology Lab (YCIL) group. YCIL process involves creating a domain model as presented by Read [8] and a Platform model which details how states and interactions are captured, a process we adopted through transformation to sequence diagram. Often times, assumptions are mostly made when further parameters are needed in forming the platform model e.g. the numerical values of some parameters. This process gives intuitive view of how the generated code can be adopted for the simulation as the interaction and states of the systems is reflected as an agent of the simulation.

The Java MASON is an open source environment which it allows portability of our code. It dedicates Java to maximize flexibility of Agent Based Simulations (ABM) development. The environment is optimised to run fast and it allows the use of our code to define classes of agents. This process will be further explored in our future work.

The simulator created for both the domain model and the platform model by YCIL group has been created in Java and utilises the MASON simulation toolkit which is run from the command line. Java MASON is used by the group because it is a multi-agent simulation as the EAE system is inherently complex with many objects which is referenced as an agent. Here, models are self-contained which makes it easy to run them on other Java frameworks. With the use of Java MASON, our output models reflected in the code is independent and can be altered at any time. Also, the models can be migrated to different platforms. Future work for the simulation looks into how our Java code can be optimized following YCIL approach in adopting Java MASON and ABM.

6.9 Evaluation and Critique

The evaluation of our second approach is done on the merit of its practicality to modelling our EAE system. MDE was explored fully through model management operations with the second approach. This was achievable through the implementation of model as the foci of our modelling process. The implemented model is transformed and validated while constraint is specified and code is generated.

This section discusses evaluation of the use of more diagrams to capture biological correspondence thereby generating object oriented code. Also, the critique of the second approach in line with its limitation is reviewed in this section.

6.9.1 Approach Evaluation

The practicality of the second approach is rooted in its use of more time dedicated to enhancing models and behavioural diagram presented in the thesis. This approach allows us explore transformation and its validity as there is no standard approach to this type of modelling. The transformation in this approach is supported by ETL language thereby giving confidence on the transformation process.

The second approach allow us to work from a set of models, and impose constraints via a constraint language. These constraints are automated and enable us to check the consistency of our models against applicable rule. Also, the second approach gives a result of consistent output with the automation of the transformation rules. The transformation rule is not analysed but its implementation leads to a consistent model.

The second approach enables EAE biological components to be traceable through implementation. Also, behaviour exhibited in EAE biological representation can be traced to the code generated. In this approach, by analysing the

behaviour of a cell in the compartments, we can automatically regenerate the diagram to show generated code is valid.

6.9.2 Limitation

Among the limitation of the second approach is its inability to self-validate as in contrast to the naive approach that uses EMF compare. Deploying second approach to Read's domain model [8] allow us to generate consistent model, however, there is no more model to try out. Our transformation rule was validated using EVL but it has not been reviewed.

6.10 Summary

In this chapter, we discuss our second approach to modelling where state diagram is used as the input model towards exploring MDE. State diagram with each transition as presented as a domain of each class in EAE is reviewed. The transformation of the model to a target metamodel is discussed. Also, the final output model that becomes the class structure diagram is reviewed. MDE tools and model management operations used to transform, validate and impose constraints on model is presented. Finally, the approach is evaluated on its approach and limitations.

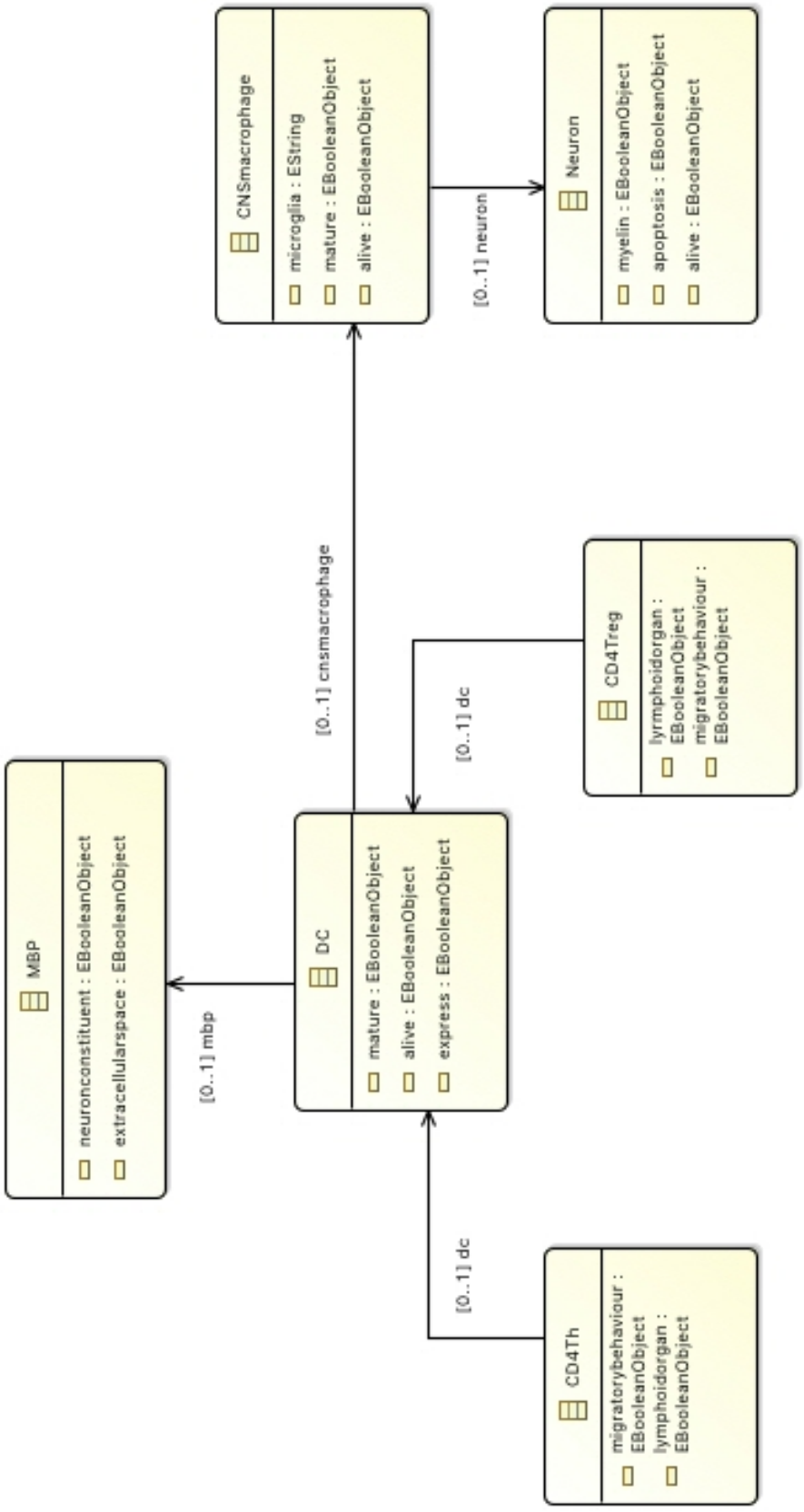


Figure 6.3: The developed state diagram metamodel.

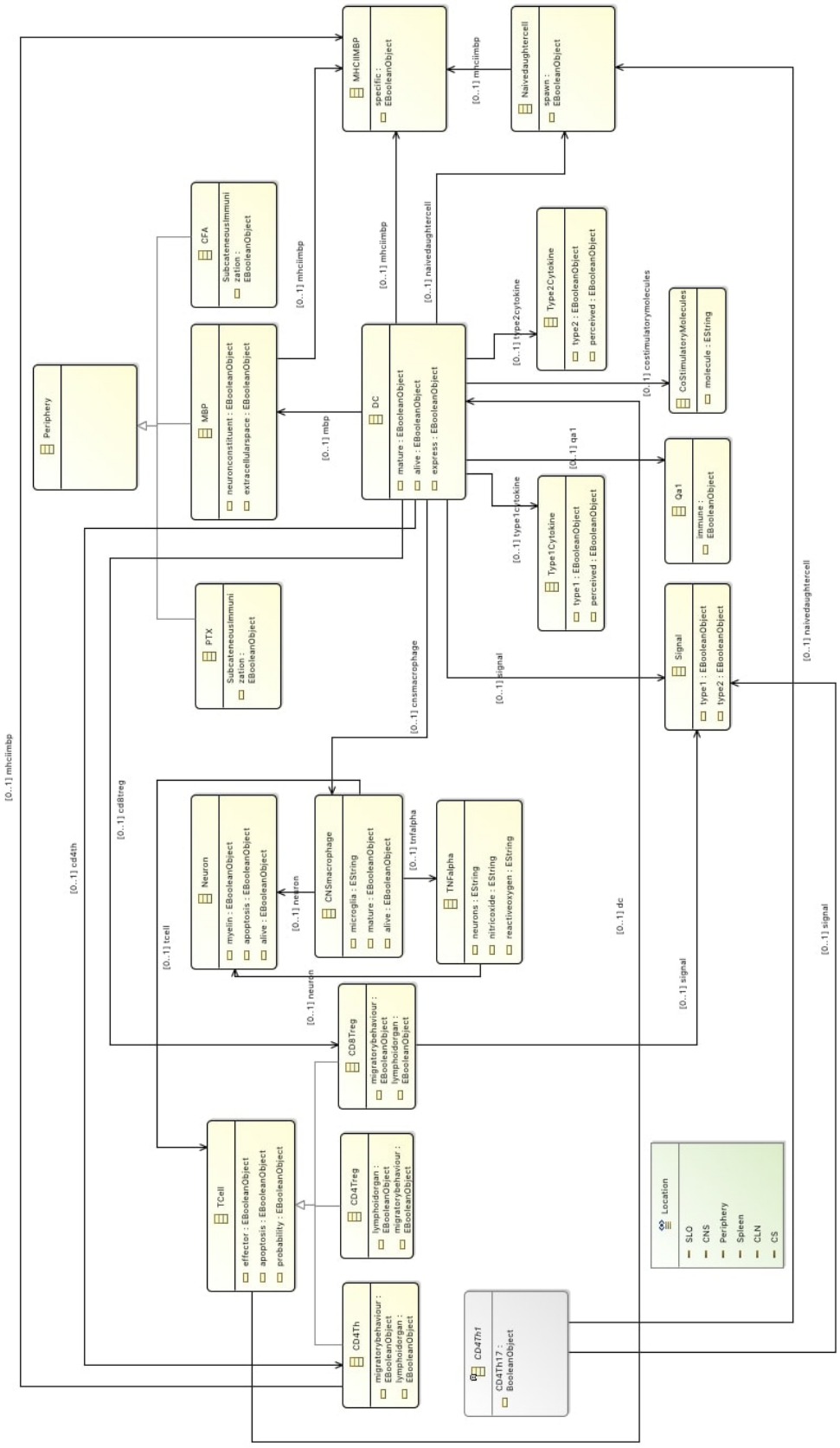


Figure 6.4: The EAE target metamodel

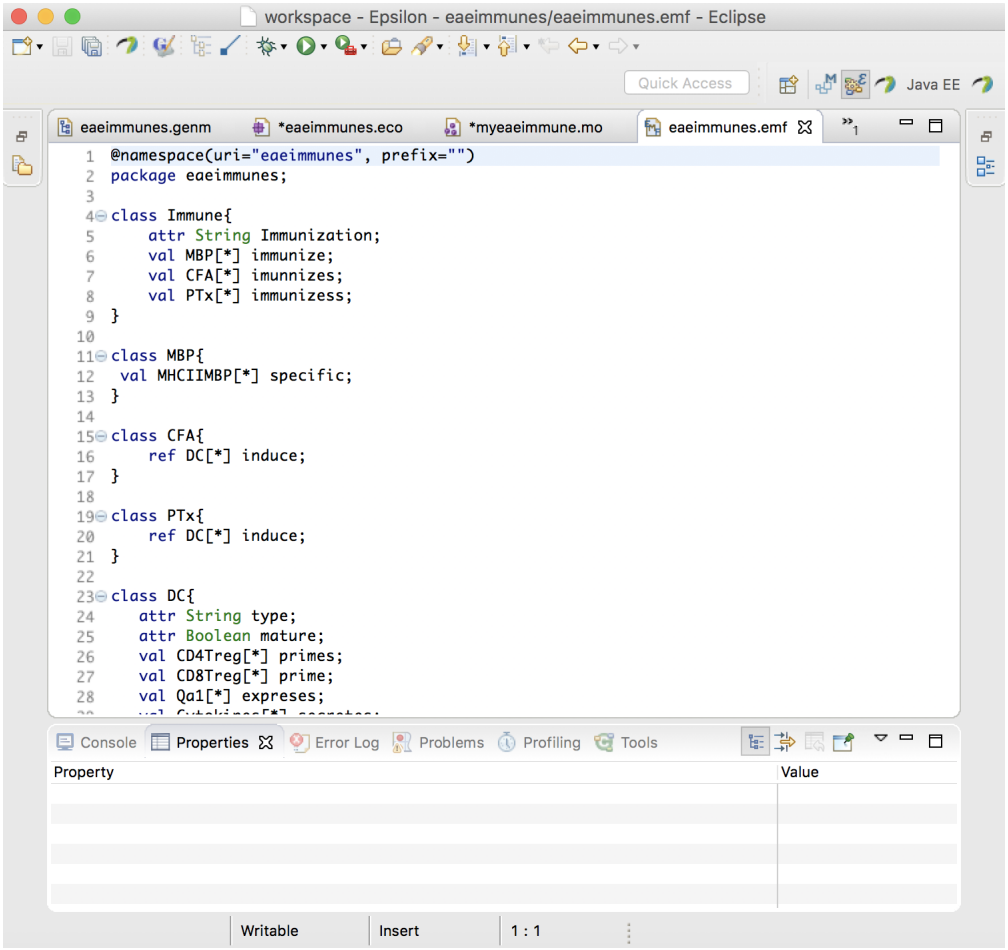


Figure 6.5: The EAE metamodel editor in emfatic text.

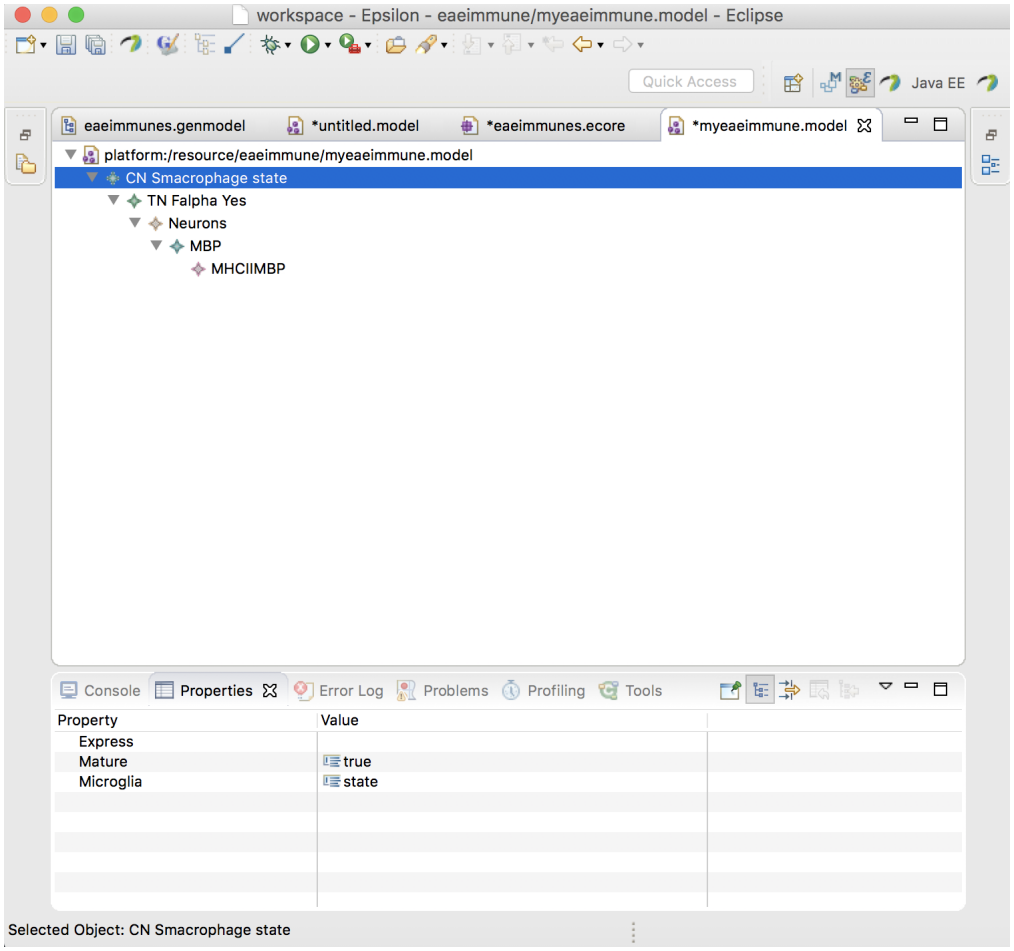


Figure 6.6: A model class of the EAE system.

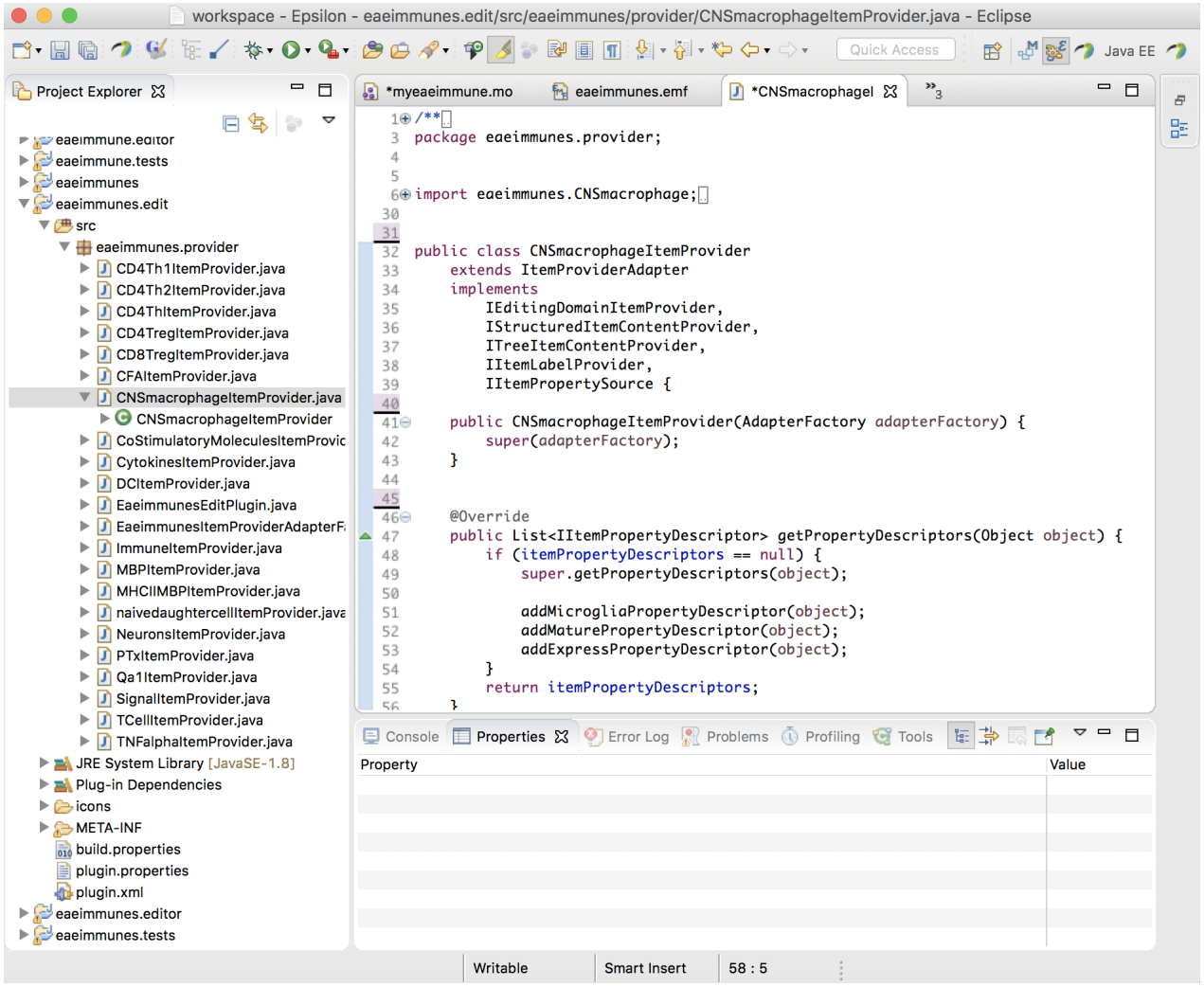


Figure 6.7: An OO Java code for CNS Class

Chapter 7

Conclusion and Future Work

This thesis examined the exploration of MDE through the lenses of behaviour diagrams. Although MDE is not conceptualized as an insoluble approach towards eliciting domain of interest, it however allows the presentation of real world system as a model and the use of this model to improve systems with increased productivity and reusability [68]. Model transformation from different abstract level to another is crucial to set up EAE input model (activity and state diagrams) to an output model (structure model) in order to generate necessary artefacts needed hence the deployment of other model management operations to aid this process.

This research has solely focus on developing and automating modelling approaches in contrast to hand-crafted codes for simulation thereby enabling the tinkering of MDE practices. This thesis has provide both a naive and second approach to modelling which gives the basis for exploring the following hypothesis:

Naive and second approaches to Transformation can be used as a step to a practicable model management operations which enables MDE to be explored through behavioural diagram.

In light of the research hypothesis, the following objectives were defined in this thesis:

- Support transformation of EAE's behaviour model to a structural model.
- Reliably and systematically query, validate, compare and transform EAE models through model management operations.
- Reliably and systematically generate code from EAE's behaviour models.

7.1 Thesis Contributions

The core contributions of this thesis in the context of limitations of behavioural diagram and the research hypothesis are outlined below.

Investigation of modelling approach concession for EAE. Through a review of our domain analysis, we identified challenges to elicitation of EAE models,

and investigated this challenge through exploration of MDE. We use the behaviour diagrams presented in Chapter 3 by conducting thorough analysis to decipher approach to modelling it.

Towards initiating our naive-approach, we analyze the sequence of events present in an activity diagram and therefore decide to transform it to a sequence diagram. Furthermore we argued that a dynamic behaviour is associated on the top-level with the activity diagram hence there is not a full detailed dynamism of behaviour model because state diagrams from the domain model shows the system single-entity dynamics. The activity diagram did captures the behaviour of EAE but its limited to the system-level overview and perspective according to [8].

To capture EAE system-level overview, modelling perspective and the single-entity dynamics as presented in Chapter 3 by [8], we proceed to using our second approach whereby an exploration of the state diagrams in Chapter 3 is utilized. The concept here is backed up by the use of state diagram to depict EAE single-entity dynamics - a full implementation of the domain model [8]. This two modelling approach have been motivated in Chapter 4.

In contrast to handcrafting code for modelling and simulation, our Naive and traditional approaches automate models and make it the pivot of every model operations. This is done by making the behavioural diagrams from our case study [8] as the input model. The refinement and manipulation of this input model utilizes MDE practices thereby exploring it in terms of model management operation.

Naive approach technique: The technique used in this approach is conceptualized on a behaviour input model (activity diagram) and a resulting structural output model (class diagram). We proceed to extracting class diagram from the activity diagram through the use of information presented in the activity. This class diagram is transformed to a sequence diagram due to its illustration of objects interactions and an object is an instance of a class. The comparison of the two sequence diagrams follows and the class diagram is subsequently updated from additional data gathered from the sequence diagram from the activity diagram. The limitation on this approach involves its constriction to only modelling the actions exhibited by the activity diagram.

Second approach technique: This approach presents a hands-on technique where state diagrams is used to develop a metamodel. This developed metamodel is transformed to a target metamodel and with is refinement with AD, OO code can be generated. The transformation follows OMG's standard and ETL rule, thus, reducing the loss of vital information crucial to future simulation. Our premise on using state diagram relies on the conception of illustrations of classes through state charts. Class diagram are known to model the dynamic flow of control within an object of a class. This approach affords us to model the full dynamic component of EAE as the overall behaviour of the system through starting-level and transitions is highly detailed as a dynamic behaviour using the state diagram.

7.2 Future Work

The future work on this paper aims to build on our shortfalls. With this approach, we have created class structure diagram and generate object-oriented code. This is a step towards agent based modelling (ABM) as they are created in an object framework. We aim to develop a substantial generic modelling tool that can be used to elicit the second approach and make it scalable. Also, the second approach will be geared towards support for creation of ABM and object-oriented based simulation. Further simulation in Java MASON as motivated in Section 6.8 will be explored as future work.

7.3 Concluding Remarks

EAE domain analysis from [8] presents biological illustration through UML behavioural diagram with slight notations and variations. These diagrams have limitations based on their ability to completely maintain the biological aspect of these models. The representation of this diagrams as an input models and their consequent transformation to a structure model through the use of model management operations enables the use of MDE. Exploration of MDE for this purpose is vital basis for automated modelling of EAE system.

However, achieving an efficient and practicable modelling approach to EAE is challenging as there needs to be confidence on the transformation process, the use of more diagrams from [8] and maintenance of the biological aspect of EAE model with the subsequent generated code.

The naive approach to transformation is limited in its capacity to modelling EAE system as it could only reflect its premises which is the activity diagram. This approach is only suitable to model the structure aspect of EAE activity diagram in contrast to its well-defined dynamic behaviour as reflected in other behavioural system (state diagram).

In addition to this, the second approach as premised on the use of state diagram for key biological entities enables model reusability and interoperability. For instance, more diagrams is widely utilized and the model developed from it through MDE can be reused when the model artefacts are merged or demerged.

This thesis has explored MDE through EAE behaviour diagram. We also conceive and automate two approaches to modelling this diagram. In addition, we propose processes that can be used towards motivating the transformation of activity diagram to a class diagram. Furthermore, for a fit-for-purpose output model, the use of model management (constraints, validation and generation) language and their effectiveness towards EAE domain of interest is proposed. Finally, as a result of our approach, we have decipher a goad for further research in how to provide a more concrete model exploration through UML behaviour diagram for object oriented structures in simulations.

Appendix A

Naive Approach

The transformation rules are applied as a guide implementation to the models and are written in ETL. A typical ETL has a name, source and target. According to [40], "the *name* of the rule follows the *rule* keyword and the *source* and *target* parameters are defined after the *transform* and *to* keywords. Also, the rule can define an optional comma separated list of rules it extends after the extends keyword. Inside the curly braces ({}), the rule can optionally specify its guard either as an EOL expression following a colon (:) (for simple guards) or as a block of statements in curly braces (for more complex guards). Finally, the body of the rule is specified as a sequence of EOL statements" [40].

A.1 AD to SD Transformation Rule

Listing A.1: ETL used to transform Activity Diagram to Sequence Diagram

```
1  rule Activity2Sequence
2   transform a : Activity!ActivityD
3   to s : Sequence!SequenceD {
4
5   s.name :=   elements.    + a.id;
6   s.contents := a.sequence.equivalent();
7  }
8
9  rule Action2Lifeline
10   transform l : Lifeline!Sequence
11   to a :  Action!Activity  {
12
13   a.element = l.element;
14 }
15
16 rule TransitionLine2Amessage
17   transform a :  Amessage!Sequence
18   to t :  TransitionLine!Activity{
```

```
19
20    t.name = a.name;
21  }
22
23  rule MergeNode2SMessage
24    transform s : SMessage!Sequence
25     to m : MergeNode!Sequence {
26
27    m.element = s.element;
28  }
29
30  rule Fork2Amessage
31    transform a : AMessage!Sequence
32     to  f : Fork!Activity  {
33
34    f.name = a.name;
35  }
36
37  rule Join2Smessage
38    transform s : SMessage!Sequence
39    to j : Join!Activity {
40
41     j.name = s.name;
42  }
```

Listing A.1 shows ETL rule used in transforming different components of our EAE activity diagram metamodel to elements of sequence diagram metamodel. The rule (Action2Lifeline) transforms action in AD to a Lifeline in CD where the name of the action is executed as a corresponding lifeline. The second rule (Transition-Line2Amessage) executes a transition line to an asynchronous message, the third rule transforms (MergeNode2SMessage) a merge node to a synchronous message. Also, due to the input and output nature of messages involved, the fourth rule (Fork2Amessage) transforms a fork to an asynchronous message while the fifth rule (Join2Smessage) executes the ADs join to a synchronous message.

A.2 AD to CD Transformation Rule

Listing A.2: ETL used to transform Activity Diagram to Class Diagram

```
1  rule Activity2Class
2    transform a : Activity!ActivityD
3    to    c : Class!ClassD
4
5       guard: not a.endNode {
6
7   c.name :=  a.name + 'Action';
8  }
```

```
 9
10 rule ActionState2ClassProperty
11  transform a : Activity!ActionState
12  to      c   : Property!Class{
13
14  c.name = a.name;
15 }
16
17 rule SubactiveState2SubClass
18  transform s : SubactiveState!Activity
19  to    a : SubClass!Class{
20
21  a.name = s.name;
22 }
23
24 rule TransitionLine2ClassRelationship
25   transform t : TransitionLine!Activity
26   to c : Relationship!Class {
27
28   c.name = t.name;
29 }
30
31 rule Merge2ClassOperation
32   transform m : Merge!Activity
33   to c : Operation!Class {
34
35   c.name  = m.name;
36 }
37
38 rule Decision2ClassOperation
39   transform d : Decision!Activity
40   to c : Operation!Class {
41
42   c.name  = d.name;
43 }
44
45 rule Fork2ClassAssociation
46   transform f : Fork!Activity
47   to c : Association!Class {
48
49   c.name = f.name;
50 }
51
52 rule Join2ClassAssociation
53   transform j : Join!Activity
54   to c : Association!Class {
```

```
55
56    c.name = j.name;
57  }
```

Listing A.2 shows ETL rule used in transforming our EAE activity diagram metamodel to an object-oriented class diagram metamodel. Here, the rule transforms Activity Diagram into CD elements. The first rule specifies the transformation of Activity to a class while the second rule shows how action state is transformed to the class property. The third rule transforms subactive state activity to a subclass. The fourth rule transforms transition line to class relationship while the fifth rule transform the activity's merge to a class operation. Decisions are transformed to operation of the class a well. Fork and Join are transformed to the Class association respectively.

A.3 CD to SD Transformation Rule

Listing A.3: ETL used to transform Class Diagram to Sequence Diagram

```
1  rule ClassD2SequenceD
2    transform c : Class!ClassD
3    to s : Sequence!SequenceD {
4
5    s.element = c.element;
6
7    if (t.parent.isDefined()) {
8      var e : new Lifeline!Sequence;
9      e.source ::= c.parent;
10     e.target = s;
11   }
12 }
13
14 rule Class2Lifeline
15   transform s : Lifeline!Sequence
16   to c : Class!Class {
17
18   guard : (not s.isAbstract)
19   c.element = s.element;
20  }
21
22 rule Property2SMessage
23   transform p : Property!Class
24   to s : SMessage!Sequence {
25
26   s.element = p.element;
27 }
28
29 rule Operation2AMessage
```

```
30    transform o : Operation!Class
31    to m : AMessage!Sequence {
32
33   m.element = o.element;
34  }
35
36  rule Association2Message
37  transform a : Association!Class
38    to m : Message!Sequence {
39
40   m.element = a.element;
41  }
```

In Listing A.3, we use ETL to transform a model that conforms to a Class diagram metamodel to a model that conforms to a Sequence diagram metamodel. When the class is translated to lifeline, the rule executes to transform class property to SDs operation messages.

Appendix B

Second Approach Transformation Rule

This section discusses the transformation rule used in the second approach transformation.

B.1 SD to SD metamodel (Structure) Transformation Rule

Listing B.1: ETL used to transform State Diagram to State Diagram Metamodel

```
1  rule StateDiagram2StateMetamodel
2   transform m : StateDiagram!Diagram
3    to   p : ObjectOriented!Metamodel {
4    p.name := 'uk.ac.york.cs.' + m.id;
5    p.contents := m.states.equivalent();
6  }
7
8  rule State2Class
9    transform s : StateDiagram!State
10   to   c : Class!Class
11      guard: not s.isFinal {
12
13   c.name :=  s.name + 'Class';
14 }
15
16 rule Event2Properties
17  transform p : Properties!Class
18   to   e : Event!State
19
20   e.name = p.name;
21 }
```

```
22
23 rule ExternalTransition2Association
24   transform a : Association!Class
25   to e : ExternalTransition!State {
26
27   e.element = a.element;
28 }
29
30 rule InternalTransition2Operation
31   transform o : Operation!Class
32   to i : InternalTransition!State {
33
34   i.element = o.element;
35 }
36
37
38 rule Transition2Parameter
39   transform p : Parameters!Class
40   to s : Transition!State {
41
42   s.element = p.element;
43 }
```

Listing B.1 specify rules used in transforming state diagram to a state diagram metamodel.

B.2 Transformation Rule for SD metamodel and AD to EAE Target Metamodel

Listing B.2: ETL used to transform state diagram metamodel and AD to the target metamodel

```
1 //rule for state diagram metamodel to target metamodel
2 rule Class2Class
3  transform c : Class!ClassMM
4  to s : Class!Class {
5
6  s.name = c.name
7 }
8
9
10 rule Attribute2Attribute
11  transform c: Attribute!ClassMM
12   to s : Attribute!Class
13
14   guard: not c.endClass {
15
```

```
16    s.name := c.name + 'Class';
17   }
18
19  rule Association2Association
20    transform a: Association!ClassMM
21   to s : Association!Class {
22
23   s.name = a.name
24  }
25
26  //rule for AD metamodel to target metamodel
27  rule Activity2Class
28    transform a : Activity!ActivityD
29    to    c : Class!ClassD
30
31       guard: not a.endNode {
32
33   c.element  :=  a.element  + 'Action';
34  }
35
36  rule ActionState2ClassProperty
37   transform a : Activity!ActionState
38   to      c   : Property!Class{
39
40   c.element = a.element;
41  }
42
43  rule SubactiveState2SubClass
44   transform s : SubactiveState!Activity
45   to    a : SubClass!Class{
46
47   a.element  = s.element ;
48  }
49
50  rule TransitionLine2ClassRelationship
51    transform t : TransitionLine!Activity
52    to c : Relationship!Class {
53
54    c.element  = t.element ;
55  }
56
57  rule Merge2ClassOperation
58    transform m : Merge!Activity
59    to c : Operation!Class {
60
61    c.merge  = m.name;
```

```
62 }
63
64 rule Decision2ClassOperation
65   transform d : Decision!Activity
66   to c : Operation!Class {
67
68   c.element   = d.element ;
69 }
70
71 rule Fork2ClassAssociation
72   transform f : Fork!Activity
73   to c : Association!Class {
74
75   c.element  = f.element ;
76 }
77
78 rule Join2ClassAssociation
79   transform j : Join!Activity
80   to c : Association!Class {
81
82   c.element  = j.element ;
83 }
```

Listing B.2 shows ETL rule used in transforming different components of a structure class diagram metamodel to another. The rule shows transformation of a class to another class, an attribute to another attribute and a class association to another class association.

Bibliography

[1] "Metamodeling," https://sites.google.com/site/raquelpau/metamodeling, accessed: 2017-02-17.

[2] P.-A. Muller, F. Fondement, F. Fleurey, M. Hassenforder, R. Schnekenburger, S. Gérard, and J.-M. Jézéquel, "Model-driven analysis and synthesis of textual concrete syntax," *Software & Systems Modeling*, vol. 7, no. 4, pp. 423–441, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10270-008-0088-x

[3] OMG, "Unified modelling language 2.1.2 infrastructure specification," OMG, Specification Version 2.1.2, November 2007. [Online]. Available: http://www.omg.org/docs/formal/07-11-04.pdf

[4] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[5] D. Cetinkaya and A. Verbraeck, "Metamodeling and Model Transformations in Modeling and Simulation," in *Proceedings of the Winter Simulation Conference*, ser. WSC '11. Winter Simulation Conference, 2011, pp. 3048–3058. [Online]. Available: http://dl.acm.org/citation.cfm?id=2431518.2431880

[6] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[7] D. Kolovos, L. Rose, G. Dominguez Antonio, and R. Paige, "The epsilon book," http://eclipse.org/epsilon/doc/book.

[8] M. N. Read, "Statistical and modelling techniques to build confidence in the investigation of immunology through agent-based simulation," Ph.D. dissertation, University of York, Department of Computer Science, 09 2011.

[9] "Agile model-driven development," http://www.cs.sjsu.edu/~pearce/oom/se/agile.htm, accessed: 2017-02-06.

[10] "UML - interaction diagrams," https://www.tutorialspoint.com/uml/uml_interaction_diagram.htm, accessed: 2017-02-07.

[11] "Statechart diagram," https://sourcemaking.com/uml/modeling-it-systems/the-behavioral-view/statechart-diagram, accessed: 2017-02-13.

[12] F. A. C. Polack, T. Hoverd, A. T. Sampson, S. Stepney, and J. Timmis, "Complex systems models: engineering simulations," in *Eleventh International Conference on the Simulation and Synthesis of Living Systems*. MIT Press, 2008, pp. 482–489.

[13] D. Moyo, "Investigating the dynamics of hepatic inflammation through simulation," Ph.D. dissertation, Department of Computer Science, York, 2014.

[14] W. Richard, "An agent-based model of the IL-1 stimulated nuclear factor-kappa b signaling pathway," Ph.D. dissertation, Department of Computer Science, York, 2014.

[15] J. Greenfield and K. Short, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Indianapolis, IN: Wiley, 2004.

[16] J. Bézivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10270-005-0079-0

[17] J. Bézivin and O. Gerbé, "Towards a precise definition of the OMG/MDA framework," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ser. ASE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 273–. [Online]. Available: http://dl.acm.org/citation.cfm?id=872023.872565

[18] A. Rensink and J. Warmer, Eds., *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4066. Springer, 2006.

[19] A. M. Starfield, K. Smith, and A. L. Bleloch, *How to Model It: Problem Solving for the Computer Age*. New York, NY, USA: McGraw-Hill, Inc., 1993.

[20] I. Kurtev, "Adaptability of model transformations," Ph.D. dissertation, University of Twente, Netherlands, 2004.

[21] R. F. Paige, N. Matragkas, and L. M. Rose, "Evolving models in model-driven engineering," *J. Syst. Softw.*, vol. 111, no. C, pp. 272–280, 01 2016. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2015.08.047

[22] R. F. Paige, D. S. Kolovos, and F. A. Polack, "A tutorial on metamodelling for grammar researchers," Science of Computer Programming, to appear, Tech. Rep., 2014.

[23] L. Rose, E. Guerra, J. de Lara, A. Etien, D. Kolovos, and R. Paige, "Genericity for model management operations," *Software & Systems Modeling*, vol. 12, no. 1, pp. 201–219, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10270-011-0203-2

[24] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "Rigorous methods for software construction and analysis," J.-R. Abrial and U. Glässer, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages, pp. 204–218. [Online]. Available: http://dl.acm.org/citation.cfm?id=2172244.2172257

[25] L. M. Rose, "Structures and processes for managing model-metamodel co-evolution," Ph.D. dissertation, University of York, Department of Computer Science, 07 2011.

[26] D. Kolovos, "An extensible platform for specification of integrated languages for model management," Tech. Rep., 2008.

[27] J. Bézivin, "On the unification power of models." *Software and System Modeling*, vol. 4, no. 2, pp. 171–188, 2005. [Online]. Available: http://dblp.uni-trier.de/db/journals/sosym/sosym4.html#Bezivin05

[28] R. F. Paige, P. J. Brooke, and J. S. Ostroff, "Metamodel-based model conformance and multiview consistency checking," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 3, Jul. 2007. [Online]. Available: http://doi.acm.org/10.1145/1243987.1243989

[29] ISO, "Information technology – Z formal specification notation – Syntax, type system and semantics," International Organization for Standardization, Tech. Rep. ISO/IEC 13568, 2002. [Online]. Available: http://standards.iso.org/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip

[30] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 307–318. [Online]. Available: http://dx.doi.org/10.1109/ASE.2009.94

[31] J. Alvarez, A. Evans, and P. Sammut, "Mml and the metamodel architecture," *Workshop on Transformation in UML, co-located with the European Joint Conferences on Theory and Practice of Software (ETAPS)*, 2001.

[32] S. Kelly and J. Tolvanen, *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470036664.html

[33] J. Cabot and M. Gogolla, *Object Constraint Language (OCL): A Definitive Guide*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 58–90. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30982-3_3

[34] D. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. New York, NY, USA: John Wiley & Sons, Inc., 2002.

[35] I. Kurtev, "State of the art of QVT: a model transformation language standard," in *Applications of Graph Transformations with Industrial Relevance*,

ser. Lecture Notes in Computer Science, A. Schürr, M. Nagl, and A. Zündorf, Eds., vol. 5088. Berlin: Springer Verlag, October 2008, pp. 377–393. [Online]. Available: http://doc.utwente.nl/62484/

[36] S. Melnik, *Generic model management : concepts and algorithms*, ser. Lecture notes in computer science. Berlin, New York: Springer, 2004. [Online]. Available: http://opac.inria.fr/record=b1100774

[37] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Softw.*, vol. 20, no. 5, pp. 42–45, 09 2003. [Online]. Available: http://dx.doi.org/10.1109/MS.2003.1231150

[38] S. Kolahdouz-Rahimi, K. Lano, S. Pillay, J. Troya, and P. Van Gorp, "Evaluation of model transformation approaches for model refactoring," *Sci. Comput. Program.*, vol. 85, pp. 5–40, Jun. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2013.07.013

[39] J. Miller and J. Mukerji, "Mda guide version 1.0.1," Object Management Group (OMG), Tech. Rep., 2003.

[40] "Epsilon etl," http://www.eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.oo2db, accessed: 2017-03-04.

[41] K. Ma, B. Yang, Z. Chen, and A. Abraham, "A relational approach to model transformation with QVT relations supporting model synchronization." *J. UCS*, vol. 17, no. 13, pp. 1863–1883, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/jucs/jucs17.html#MaYCA11

[42] D. S. Kolovos, "An extensible platform for specification of integrated languages for model management," Ph.D. dissertation, University of York, Department of Computer Science, 2009.

[43] M. Elaasar and L. Briand, "An overview of UML consistency management," Carleton University, Tech. Rep., 08, sCE-04-18.

[44] J. Henriksson, F. Heidenreich, J. Johannes, S. Zschaler, and U. Aßmann, "Extending grammars and metamodels for reuse: the reuseware approach," *IET Software*, vol. 2, no. 3, pp. 165–184, 2008.

[45] C. Brun and A. Pierantonio, "Model differences in the eclipse modelling framework," p. 2934, 2008, upgrade.

[46] S. Gérard, C. Dumoulin, P. Tessier, and B. Selic, "Papyrus: A UML2 tool for domain-specific language modeling," in *Proceedings of the 2007 International Dagstuhl Conference on Model-based Engineering of Embedded Real-time Systems*, ser. MBEERTS'07. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 361–368. [Online]. Available: http://dl.acm.org/citation.cfm?id=1927558.1927582

[47] "Papyrus modeling environment," https://eclipse.org/papyrus/index.php, accessed: 2017-02-08.

[48] D. Kolovos, R. Paige, and F. Polack, "The Epsilon Object Language (EOL)," in *Proc. European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, ser. Lecture Notes in Computer Science, A. Rensink and J. Warmer, Eds., vol. 4066. Springer, 2006, pp. 128–142.

[49] D. S. Kolovos, L. M. Rose, S. B. Abid, R. F. Paige, F. A. C. Polack, and G. Botterweck, "Taming EMF and GMF using model transformation," in *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I*, ser. MODELS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 211–225. [Online]. Available: http://dl.acm.org/citation.cfm?id=1926458.1926479

[50] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot, "A research roadmap towards achieving scalability in model driven engineering," in *Proceedings of the Workshop on Scalability in Model Driven Engineering*, ser. BigMDE '13. New York, NY, USA: ACM, 2013, pp. 2:1–2:10. [Online]. Available: http://doi.acm.org/10.1145/2487766.2487768

[51] P. S. Andrews, F. A. C. Polack, A. T. Sampson, S. Stepney, and J. Timmis, "The CoSMoS process, version 0.1: A process for the modelling and simulation of complex systems," Department of Computer Science, University of York, Tech. Rep. YCS-2010-453, Mar. 2010.

[52] F. A. C. Polack, P. S. Andrews, T. Ghetiu, M. Read, S. Stepney, J. Timmis, and A. T. Sampson, "Reflections on the simulation of complex systems for science," in *2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, March 2010, pp. 276–285.

[53] S. V. Mierlo, "Explicitly modelling model debugging environments," in *Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015, Ottawa, Canada, September 29, 2015.*, 2015, pp. 24–29. [Online]. Available: http://ceur-ws.org/Vol-1503/05_pap_mierlo.pdf

[54] J.-P. Barros and L. Gomes, "From activity diagrams to class diagrams." [Online]. Available: http://www.disi.unige.it/person/ReggioG/UMLWORKSHOP/Barros.pdf

[55] B. Donald, "UML basic: An introduction to the unified modeling language," 2003. [Online]. Available: http://www.therationaledge.com/content/jun{_}03/f{-}umlintro{_}db.jsp

[56] O. M. G. (OMG), "Meta-object facility (mof) specification, version 2.5." [Online]. Available: http://www.omg.org/spec/UML/2.5

[57] B. P. Douglass, *Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks, and Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[58] M. Felici, "Sequence diagrams," http://www.inf.ed.ac.uk/teaching/courses/seoc/2011_2012/notes/SEOC08_notes.pdf, accessed: 2017-02-07.

[59] OMG, "OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1," Object Management Group, August 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1

[60] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[61] OMG, "OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1," Object Management Group, August 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1

[62] E. Compare, "Emf compare-userguide version 3.1.0.201506080946." [Online]. Available: https://www.eclipse.org/emf/compare/overview.html

[63] D. Peters and D. L. Parnas, "Generating a test oracle from program documentation: Work in progress," in *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '94. New York, NY, USA: ACM, 1994, pp. 58–65. [Online]. Available: http://doi.acm.org/10.1145/186258.186508

[64] M. Lenzerini, "Data integration: A theoretical perspective," in *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '02. New York, NY, USA: ACM, 2002, pp. 233–246. [Online]. Available: http://doi.acm.org/10.1145/543613.543644

[65] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. C. Polack, "The design of a conceptual framework and technical infrastructure for model management language engineering," in *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ser. ICECCS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 162–171. [Online]. Available: http://dx.doi.org/10.1109/ICECCS.2009.14

[66] D. S. Kolovos, R. F. Paige, and F. Polack, "The Epsilon Object Language (EOL)," in *Proceedings of the Second European Conference on Model Driven Architecture Foundations and Applications*, ser. Lecture Notes in Computer Science, vol. 4066. Springer International Publishing, 2006, pp. 128–142.

[67] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "Rigorous methods for software construction and analysis," J.-R. Abrial and U. Glässer, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages, pp. 204–218. [Online]. Available: http://dl.acm.org/citation.cfm?id=2172244.2172257

[68] I. Ráth, G. Varró, and D. Varró, "Change-driven model transformations," in *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*,

ser. Lecture Notes in Computer Science, A. Schürr and B. Selic, Eds., vol. 5795, Springer. Springer, 2009, pp. 342–356, springer Best Paper Award and ACM Distinguished Paper Award Acceptance rate: 18%. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04425-0_26