# Software Engineering for a Complex Domain Model to a Platform Model

Muideen Ajagbe, Fiona A.C Polack and Richard Paige,
Department of Computer Science
University of York
York, UK YO10 5DD
Email: maa589@york.ac.uk

**Abstract**

The paper presents a practical approach to the transformation of UML-style activity diagrams to class diagrams and OO code. Activity diagrams have been widely used to model behavioral aspects of complex systems – our work uses models produced as part of simulation design of aspects of complex immune systems in York Computation Immunology Lab. Automated transformation to code makes the implementation of complex systems repeatable and maintainable.

## 1 Introduction

An effective software-engineering process for modeling complex system uses models, often shown with different UML diagram. Modeling a complex systems involves detailed behavior (abstraction) at high level due to its many simple behavior in low level [1]. As complex system involves high level of abstraction due to its increased complexity, a concrete model helps capture a system under development as a whole. In perspective, diagrams used for representing the role of a system by York Computation Immunology Lab (YCIL)[1] group are mostly behavioral hence the need to transform them to a class structure for textual and code generation that will enables easy analysis of the emergent behaviors in the system.

Model is a form of abstraction that is developed to help understand what the system entails [2]. The abstractions captured uses software engineering approach like Model Driven Engineering (MDE) which target the concerned domain of a model.

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

MDE helps capture the artefacts, data and every associated part of a domain model in a system [3]. For example, due to the concerned behavior of a complex system, models used by the YCIL group are often represented in UML activity diagram showing the system (actors) different roles. The application of MDE practices will help aid model to model (M2M) transformation needed for code generation and graphical comparison which is the result of the system's simulations.

Our paper aims to show an approach to M2M - activity diagram (AD) to class diagram (CD), model-rule syntax, code generation and how that process could be carried out. From the CD, object oriented (OO) java code can be generated as a class structure with operations and attributes providing a vivid representation of the occurrence in a system. OO code evolved from concepts of object rather than logic [4]. Here, we automate all our process by conforming to Object Management Group (OMG)[2] UML standard rule. A top down methodology approach is used in transforming our model and a rule-based approach is used with tool support for automation and model analysis.

## 2 Motivation and Related Work

In this section, we discuss the motivation for our work and the goals we want to achieve. We analyze our approach to model transformation and well as earlier work.

### 2.1 Motivation

The main motivation for the transformation of AD to CD stems from the YCIL group work. Domain model from the group is often represented using an activity diagram. The activity diagram shows actor's role in a system, therefore, there are limited data representing the whole system. This leads to the need to transform the AD to a CD so as to generate code which can be used to understanding the emergent behavior from their work or any complex system.

Furthermore, during the elicitation process of our provided domain, we determine that providing a generic basis for an effective transformation will provide a robust and fit-for purpose class diagrams which can be further refined to an agent for simulations or used to generate OO code.

As all the YCIL theses [5], [6], [7] use similar modeling diagrams to but handcraft code for simulation. The handcrafted simulation(in code) can be clearly

[2]Object Management Group Unified Modeling Language: http://www.omg.org/spec/UML/

written once, difficult to reuse and maintain. Also, a change that is easy to make in a diagram may require significant re-engineering of code. For example, the YCIL simulators are all closely related, but each has been separately engineered to address a different question or a different part of the immune system.

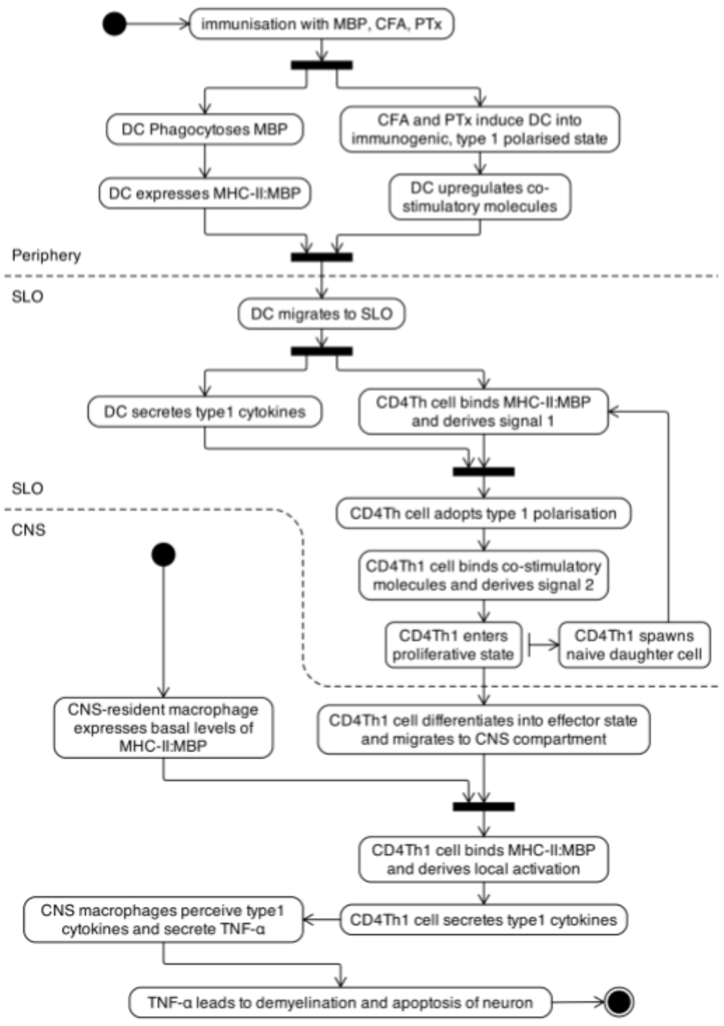

immunisation with MBP, CFA, PTx
DC Phagocytoses MBP
CFA and PTx induce DC into immunogenic, type 1 polarised state
DC expresses MHC-II:MBP
DC upregulates co-stimulatory molecules
Periphery
SLO
DC migrates to SLO
DC secretes type1 cytokines
CD4Th cell binds MHC-II:MBP and derives signal 1
SLO
CNS
CD4Th cell adopts type 1 polarisation
CD4Th1 cell binds co-stimulatory molecules and derives signal 2
CD4Th1 enters proliferative state
CD4Th1 spawns naive daughter cell
CNS-resident macrophage expresses basal levels of MHC-II:MBP
CD4Th1 cell differentiates into effector state and migrates to CNS compartment
CD4Th1 cell binds MHC-II:MBP and derives local activation
CNS macrophages perceive type1 cytokines and secrete TNF-α
CD4Th1 cell secretes type1 cytokines
TNF-α leads to demyelination and apoptosis of neuron

Figure 1: The activity diagram provided by [5] from YCIL group.

More so, handcrafted code might contain errors and may not reflect the actual representation of the models in contrast with a generated code. MDE can both properly underpin the diagram notations and support transformation to code. Our ultimate goal is to be able to:

1. Effectively provide a transformation process from AD to CD.
2. Reliably and systematically generate code from YCIL-style diagrams.
3. Reliably and systematically generate revised code when models (diagrams) are modified.
4. Support reuse and modification of the existing platform models and simulations.

### 2.2 Related Work

There is no acceptable standard approach to this type of M2M transformation as such little work has been carried out in this area. There are published articles that lean towards these transformations but they are not concrete enough. Molina *et al* [8] presents an approach to use-cases and conceptual models through business models. The approach here lay emphasis on the use of business use cases to business activity diagram. The identification of roles in the scenarios leads to creation of interaction diagrams and a class diagram which can be analyzed for matching components.

Also, Suarez *et al* [9] detail a transformation from a business model point of view. Here, use-cases are used to build an activity diagram. A prescribed translation rule is the used to translate the AD to a CD. Barros *et al* [10] presents a process to deriving class diagrams using algorithm translation. This process offers a starting point to our approach but they are not capturing enough data from a high abstraction level.

The context of transformation in [10] is built upon metamodel from some known use-case scenarios where translation of metamodel is done arbitrarily. This approach, however, enables the AD to comply to a domain specific language (DSL) and OMG standard for components of a CD. The translated AD and CD are further refined to a metamodel so as to deduce available translated components. Building on all these related work, we propose a concrete model transformation by incorporating a metamodel built using a translated CD with the aid of Eclipse Modeling Framework Ecore proposed in [11].

## 3 Model Transformation

In this section, we discuss the need for model transformation, our model categorization how it will help us with the transformation of AD to CD

## 3.1 Model

Model is made up of artefacts that is derived from capturing the abstraction of a system [1]. The act of building models is referred to as modeling. Models, once built, are amenable to automated manipulation and analysis. This gives basis for a better understanding of the composition of the model. The structure of a model is rooted in the identification of crucial element and available objects as presented in the system's information thereby eliminating duplication and inconsistency. Based on motivation from the CoSMoS approach [12], we seek to categorize our views on model into components which represent the context of our work. Our models are categorized into the following compartments:

- The Domain Model (DM)
- The Platform Model (PM)
- The Result Model (RM)

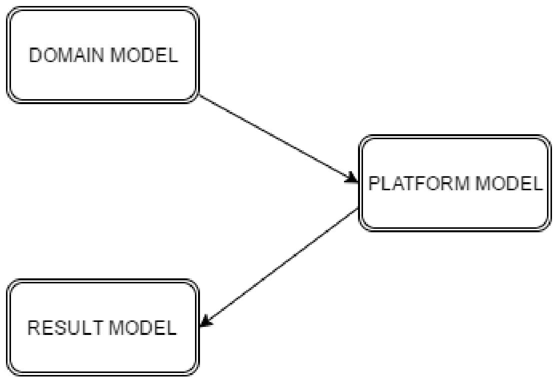


Figure 2: The compartment showing movement from DM to PM to RM.

As the DM identifies and outlines the interactions and behavior of the system domain, our PM provides a design model that details occurrences in the domain. On the other hand, the RM shows result of what has been done to the PM (mostly simulations) and act as a comparison standard for the actions of the DM. In addition, a rigid model gives basis for a good modeling thus, a fit-for-purpose code is generated and an emergent behavior can be detected by further simulation.

## 3.2 Modeling and Transformation

The process of modeling involves the crucial understanding of the system at play. Unified Modeling Language (UML) is used as the industry standard for modeling. Based on the YCIL work in [13], UML activity diagram is used to capture the DM of interest [14]. The aim is to model an AD to a CD which allows the continuous traceability of provided artefacts to code generation. As discussed in [15], the concept of domain modeling is crucial to a complex systems development. Nonetheless, without a concrete transformation of a model system in AD to a CD, little information about the system under development will be provided.

A well translated AD to CD is aimed at producing executable code. As reasoned from the above chapter, [10] proposed a means of formalizing mappings between AD and CD with the use of metamodels and the reverse engineering capability of any OO code generated. While this makes model transformation feasible, our view is to augment the data from the activity into an object which can be used to build an Object diagram (OD). This OD is then reverse-instantiated and engineered to a CD in the PM. An engineered CD is used to build a metamodel and a model representing the abstract view and the crucial details present in the system. A resulting code or textual explanation presents meaning to our research context in the RM.

## 4 Requirement for Model Transformation

In this section we discuss in details, the approach we took towards our translation of AD to CD using models derived from an engineered metamodel using MDE approaches. Due to the high level of abstraction on the YCIL work, we use the open source modeling platform in epsilon called EMF.

As stated in the above chapter, we transform our AD to an OD which we further refined to deriving a CD. We develop this method using an incremental approach. This gives room for us to modify the originating data or object from the AD. Based on Complex system and modeling simulation approach CoSMoS in [15], the AD describing the flow of action is our DM. We propose a meta-domain model consisting of the objects in the OD. This gives a basis for us to refine the OD to a PM which is represented by the engineered CD. The issues of conformance of the AD to the OD hinges on two rules which are:

1. Analyze the actions of the AD that represent data or object and represent them as Object in the OD.
2. Analyze any common properties and represent them as a Value in the OD.

It is important for us to have the right element for transformation. A well annotated activity diagram that conforms to the above rules provides a starting point towards transformation. It is also noteworthy not to leave any details out. A concrete transformation will aid the correctness of the system when it undergoes further work like transition or simulation.

## 5 Model Mapping and Rule

As discussed in the above section, the conformance guidelines helps give a robust transformation of our models represented by the AD to a more generic OD. In this section, we discuss how our defined rules and guidelines help define the composition of our model and the necessary perspective for our model transformation.

### 5.1 Model Process

In order to present a generic form of model transformation, we move to utilize the concept of UML modeling that present a systems artefacts and adopted modeling mechanism(in our view, AD, OD and CD) as the true representation of the system being modeled. Under this circumstance, transformation of model involves critical process that needs to conform to the modeled system regardless of the complexity involved. However, since there is no generally accepted form of transformation from AD to CD, we proffer a solution through our modeling process and approach. Our premise is the use of CD to generating OO code:

- We assume an OO target, so a class model provides structure of the code.
- We extract Objects and Slots from behavioral models i.e. the AD and can thus transform AD to an OD.
- We reverse-engineered our transformed OD to a CD where codes can be generated.

### 5.2 Mapping AD to OD

It is tedious and very difficult to derive OO code from behavioral diagrams as the structural features are not well detailed. However, transforming an AD to an OD requires the identification of object and thus provides a more structural view of the system's behavior. We map AD to OD by extracting and instantiating objects or data presented in the AD using the following preconditions and association basis:

### 5.2.1 Preconditions

1. Identity actions in the AD that can be refined as an Object.
2. identify characteristics of the action that can be refined as a Slot.
3. Identify data flow in and out of the activity which can be refined as a Link.
4. Objects representing the same data is instantiated as a single Object
5. An action derived from another action is translated to a super-Object.

### 5.2.2 Associations

1. Data flows in and out of the activity is examined with any accompany adjectives(label) and thus translated as a classifier or association relationship.
2. The data flows are compared for any common relation wherein any common relation creates basis for the generalization of the association between objects of the OD.
3. Decision points which are Boolean operations is translated as either 1 to Many or Many to Many relations depending on the attributes between the AD's activities.

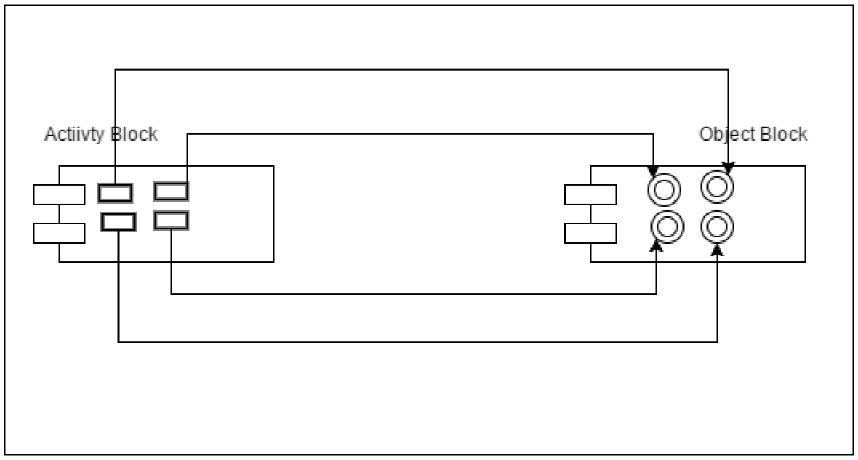


Figure 3: The modules showing transformation from AD to OD.

Schematically, this process of transformation from AD to OD is represented in Figure 3, wherein lies the module surface named activity block showing the available data or objects present in the activity diagram. The other module named object block has spherical shape representing the refined object. The arrows maps each activity data from the first block to an object in the second block.

## 6 Rule-Based Model Transformation

Having defined the context from which our model lies. We sought to make our model concrete by presenting a generic basis for transformation defined by rules. The rule is agnostic and we aim to implement it towards automation of the system in general. In the above antecedent, we discuss mapping of our process to give a rigid transformation. Here, we back-up our approach by using rules to form a more generic basis for transformation.

### 6.1 Model-Rule Syntax

We examine an operation of a system and the possible events that may occur in it. This leads us to classify actions of an activity as a model-driven event which gives form of a persistent occurrence in the system. We classify this actions as operations of create, retrieve, update and delete (CRUD). We analyze the choice of words and adjectives used to fit in the context of CRUD operations. For create:

1. IF activity is Create
2. then exist Object1 of
3. Class Creating Activity
4. AND create Object2 of
5. Class Created Activity
6. AND create link
7. BETWEEN Object1 and Object2

We apply the same rule to the other operations - Retrieve, Update, and Delete. In applying this rule, we have a generic transformation of our objects in the OD. This rule is applied to help achieve a conceptual form of the system from the activity diagram.

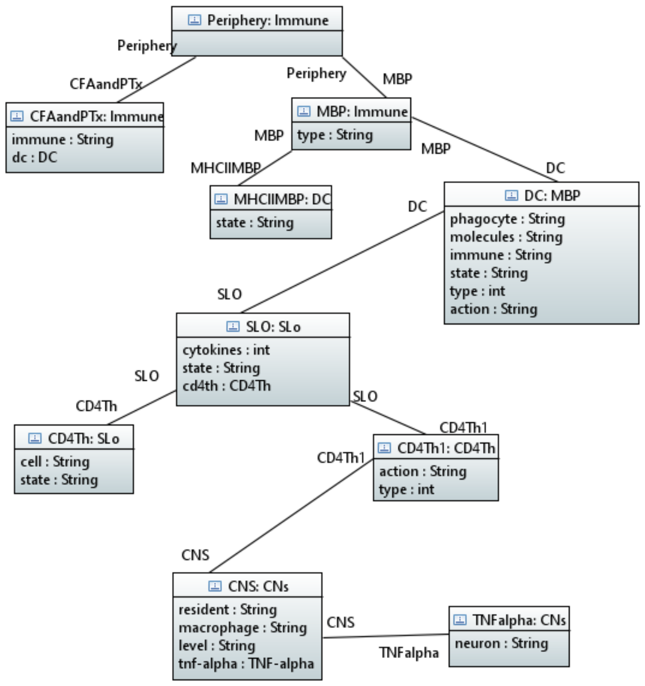


Figure 4: The transformed OD from the AD in Figure 1.

## 7 Model Reverse-Engineering

In the chapters above, we have detailed the need for a CD and how that can be achieved with the concept of objects available in the AD being used to build an OD. Pedagogy and published journals has revealed that an object diagram is an instance of a class diagram. More so, an OD represents the static characteristics of a system by using presented objects and their accompanying relations as an instance. This means the OD is made up of things contained in CD. We already established the availability of objects in an activity hence the approach towards creating an OD out of AD. Since an OD is an instance of a class, we are presented with a relationship between an OD and a CD in the form of an instance. The reverse-instantiation or reverse-engineering of an OD gives a fit-for purpose CD where further work like code generation, representation of agents, simulation and other possible solutions. In achieving a standard reverse engineering from an OD to a CD, we propose the following rules:

- First, analyze the OD model and choose the instances with more important data and association
- Second, analyze the instances that cover the functionality

- Third, analyze the optimization having the limited instances

After considering the above rules, it is imperative to understand the below concepts carefully:

- Class diagram consist classes
- The reference in the class diagram are used to connecting classes.
- Classes and references are the building blocks of the class
- Characteristics (attributes and operations) are the formal parameters or method showing the behavioral characteristics of the system.

For the full reverse engineering, the following actions is carried out;

- The Object in the OD is reverse-engineered into the CD's class as its the object type.
- The Link in the OD is reverse-engineered into the CD's association as its the instance of relationships.
- The Value or SLOT in the OD is reverse-engineered into the CD's characteristics as its the attribute type.

By using this approach, we have a more concrete transformation of AD to CD through the reverse engineering of the OD. In this circumstance, the outcome of the DM is the AD, the OD is the first meta-level of PM and the CD is the higher meta-level. With a fit-for purpose CD, codes is generated thereby generated using EMF to model the CD.

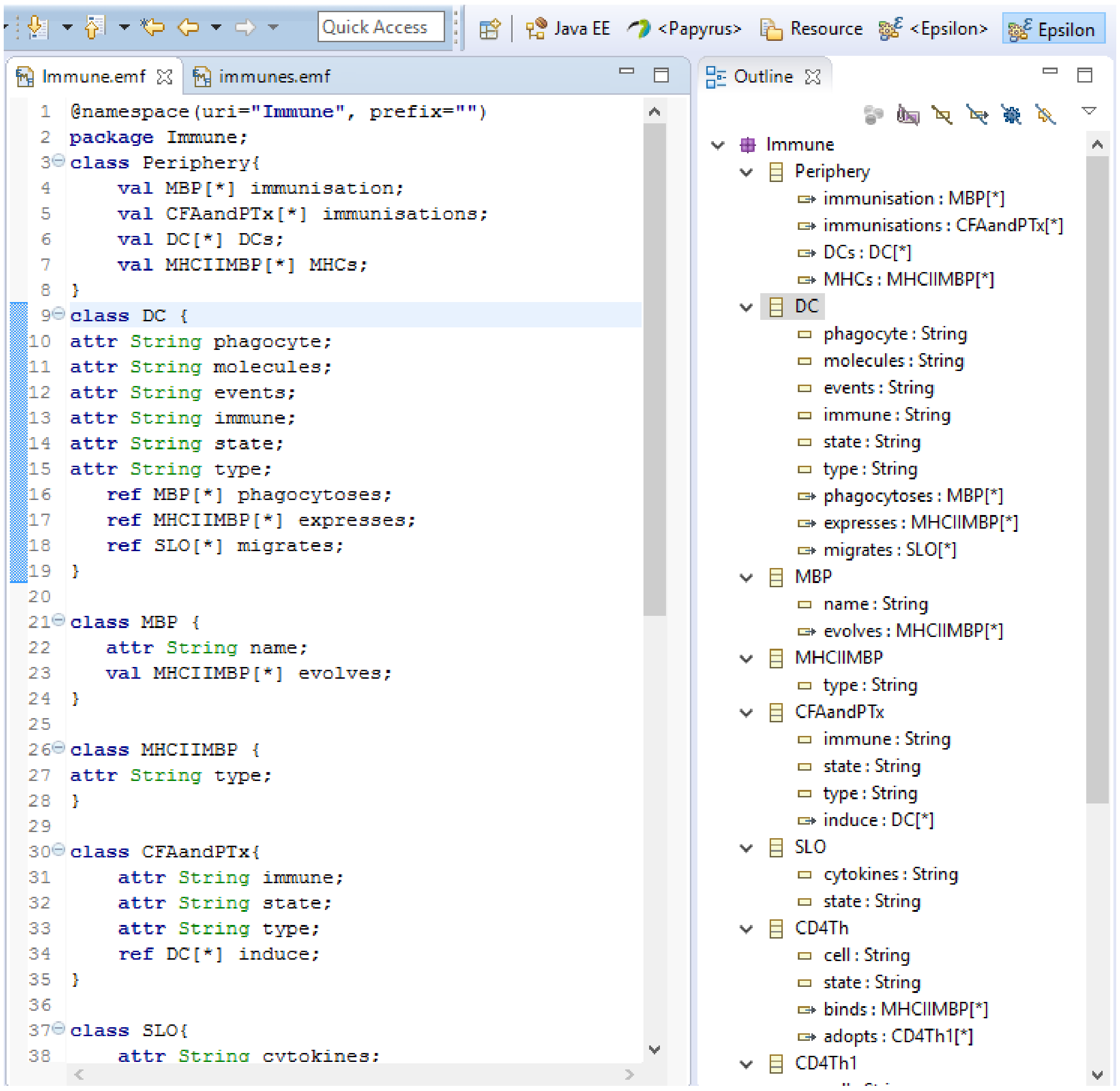
# 8 Automation

In this chapter, we discuss the automation of our models which provides basis for test cases and effective drawing of our AD and OD and the modeling of the engineered CD.

## 8.1 Papyrus

Papyrus[3] is an open source model-based engineering tool built on Eclipse. It enables an integrated environment where UML can be edited. It can be easily customized to provide support for diagram notations, exploring models and

[3]Papyrus modeling environment: https://eclipse.org/papyrus/



```
@namespace(uri="Immune", prefix="")
package Immune;
class Periphery{
     val MBP[*] immunisation;
     val CFAandPTx[*] immunisations;
     val DC[*] DCs;
     val MHCIIMBP[*] MHCs;
}
class DC {
attr String phagocyte;
attr String molecules;
attr String events;
attr String immune;
attr String state;
attr String type;
    ref MBP[*] phagocytoses;
    ref MHCIIMBP[*] expresses;
    ref SLO[*] migrates;
}

class MBP {
    attr String name;
    val MHCIIMBP[*] evolves;
}

class MHCIIMBP {
attr String type;
}

class CFAandPTx{
     attr String immune;
     attr String state;
     attr String type;
     ref DC[*] induce;
}

class SLO{
     attr String cytokines;
```

```
Immune
  Periphery
    immunisation : MBP[*]
    immunisations : CFAandPTx[*]
    DCs : DC[*]
    MHCs : MHCIIMBP[*]
  DC
    phagocyte : String
    molecules : String
    events : String
    immune : String
    state : String
    type : String
    phagocytoses : MBP[*]
    expresses : MHCIIMBP[*]
    migrates : SLO[*]
  MBP
    name : String
    evolves : MHCIIMBP[*]
  MHCIIMBP
    type : String
  CFAandPTx
    immune : String
    state : String
    type : String
    induce : DC[*]
  SLO
    cytokines : String
    state : String
  CD4Th
    cell : String
    state : String
    binds : MHCIIMBP[*]
    adopts : CD4Th1[*]
  CD4Th1
```

Figure 5: An Epsilon Emf metamodel representing the class diagram reverse-engineered from the OD.

model-based simulations. Based on this features, we use papyrus to develop a tool for our system modeling. This tool help us develop our AD and translated OD. Essentially, Papyrus assists in automating the different diagrams representing our system's complexity

## 8.2 Eclipse Epsilon

Eclipse Epsilon [4] ”is a family of languages and tools used for code generation, M2M transformation, model validation, comparison, migration and refactoring that work out of the box with EMF and other types of models”. We use it to build our metamodel (CD) and generate our OO (java) code which will help in providing a textual meaning for a resulting model of the system. Most importantly, a model editor can be generated for further reuse and easy transformation to other models. We regard using these approach for concrete modeling of a fit-for-purpose model.

## 9 Discussion and Conclusions

We made use of the approach stated earlier to provide a modeling transformation and simulation for the YCIL AD. The transformation leading to simulation (code generation) has proved effective to producing a RM which details the emergent behavior of the modeled system. We have tried this approach on several activity diagram and we hope to cover more ground by applying it more on YCIL ADs which are more complex. Further work includes seeing our objects as agent thereby developing an approach that augment an Agent Based Modeling (ABM).

As retorted earlier, there is no current standard method being used for transforming AD to CD. Starting from modeling complex autoimmune systems, we proffer an approach and tool for doing this. We showed a detail a transformation of AD to CD using objects thus enabling the generation of reusable code.

The UML diagram afforded us the opportunity to use different diagrams and our approach enables us to develop a tool that capture the domain of interest. The modeling is made as simple as possible to capture our our and the generated code is relayed to the RM which can be easily identified by the domain experts. Our model was automated using eclipse papyrus tool and EMF which enable us to draw the AD with OD and modeling of the CD respectively. The tools are open source which allows for re-usability of our tool and approach.

## References

[1] F. A. C. Polack, T. Hoverd, A. T. Sampson, S. Stepney, and J. Timmis, “Complex systems models: engineering simulations,” in *Eleventh Interna-*

[4]Epsilon :http://www.eclipse.org/epsilon/

*tional Conference on the Simulation and Synthesis of Living Systems.* MIT Press, 2008, pp. 482–489.

[2] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, 1st ed. Morgan & Claypool Publishers, 2012.

[3] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, Feb. 2006. [Online]. Available: http://dx.doi.org/10.1109/MC.2006.58

[4] E. Kindler and I. Krivy, "Object-oriented simulation of systems with sophisticated control," *International Journal of General Systems*, vol. 40, no. 3, pp. 313–343, 2011. [Online]. Available: http://dx.doi.org/10.1080/03081079.2010.539975

[5] R. Mark, "Statistical and modelling techniques to build confidence in the investigation of immunology through agent-based simulation," Ph.D. dissertation, Department of Computer Science, York, 2011.

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

[7] 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.

[8] J. G. Molina, M. J. Ortín, B. Moros, J. Nicolás, and A. Toval, "Towards use case and conceptual models through business modeling."

[9] E. Suarez, M. Delgado, and E. Vidal, "Transformation of a process business model to domain model."

[10] J.-P. Barros and L. Gomes, "From activity diagrams to class diagrams."

[11] J. Johannes, I. Savga, and T. Haupt, "Integrating fujaba and the eclipse modeling framework," 2006.

[12] P. Garnett, S. Stepney, F. Day, and O. Leyser, "Using the cosmos process to enhance an executable model of auxin transport canalisation," in *In Stepney et al*, pp. 9–32.

[13] Y. C. I. Lab, "Software engineering." [Online]. Available: https://www.york.ac.uk/computational-immunology/research/soft-eng/

[14] 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

[15] 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.