Complex System Simulation: Processes and Transformation from Activity Diagram to Class Diagram

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

A model is an abstract representation of a complex system according to Brambilla *et al.* [1]. Modeling is the process of constructing a model [2]. This process requires collaborating with experts in the complex system's field, where data on the system is processed for modeling. Model Driven Engineering (MDE) is an approach to software engineering whereby artifacts, data and everything involved is seen as a model. MDE is an approach that helps with detection and prevention of errors that may arise during developments of these models [3]. A good model free of errors provides a good representation of our system.

Models, once constructed, are amenable to automated manipulation and analysis, sometimes called model management. There are many different model management operations. One is model simulation, which is used for the modeling of complex systems [2]. As the model represents our system, we apply MDE to help with the engineering of the modeling processes. Also, we use simulation for the engineering of complex systems so as to comprehend its emergent behavior and operation. This emergent behavior could be examined in real world

system. Most importantly, a result of modeling simulation provides a tool or an approach for the generation and testing of code (hypothesis) which provides result to our system's complexity.

The goal of developing better (more repeatable and rigorous) support for building complex system simulation drives us to modeling. Modeling practices used to design simulators of complex systems can seem very inaccurate due to the high level of abstractions involved; yet using MDE practices to support effective modeling also faces many obstacles in the ability of the modeler to maintain consistency with the different models of the same system. As the Unified Modeling Languages (UML) provides multiple diagram styles for the same modeling, the interpretation of these multiple diagrams in the same models can lead to inconsistencies [1]. Any inconsistency that arises can negate the correctness of the process, tool and code generated and validation of the behavior of the complex system.

The purpose of this paper is to detail our process for supporting simulation of complex systems through transformation of the behavioral model - activity diagram (AD) to a structural model - class diagram (CD) which enables easy generation of code. Complexity is due to a systems' behavior so modeling complex systems require us to focus on behavioral models (rather than the usual focus on state - CDs). CD is very crucial to our approach as it gives a good basis for generating Java object oriented(OO) code - it provides a class structure, with operations and attributes. The diagrams are the manually translated artefacts while the models are the automated form of this diagrams. Consistency of transformed model and validation of the diagrams is a key component that poses a major challenge about this process. A success of this process will provide a confidence level and correctness to the transformation and complex systems simulation. Also, it will lead to a crucial process of transforming AD to CD. The method applied for carrying out this process is detailed in section 4, 5 and 6.

Modeling complex systems involves the attempt to represent their complexity by modeling their explicit behavior (of agents). Most aspects of complex systems are modeled using a behavioral model, such as an activity diagram. The onus is on the modeler to transform the activity diagram to code; for instance, using an object oriented (OO-based) approach. A specific example of this appears in a number of YCIL theses[1] [4] [5] [6] which handcraft code in Java Mason (OO-based agent platform). Java Mason as defined by Luke [7] "is a multiagent simulation toolkit designed to support large numbers of agents relatively". It is a distinct multiagent simulation that is developed in java to enable the customization of java simulation and also enabling the addition of functions to less-weighty simulations.

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

We aim to eliminate the code handcrafting and also automate more of the process for building a simulation. To do this, we propose to extract a class diagram provided scenarios using model transformation. However, it is defensible to say both diagrams are trying to model the behavior of the same system with different notations. Their differences are based on the interpretations by the different domain experts. As the system is presented by the domain experts in form of an activity diagram, a software engineer can transform the provided diagram to a class diagram targeted to the designated object oriented platform model which helps with our transformation to a simulation model that can be used to help understand the system's behavior.

Using the complex system and modeling simulation conceptual framework (CoSMoS) [8], the YCIL researchers' [9] ideas are expressed with UML models. The different stages expressed in the framework inspired us to create an approach and process to designing a tool for the development of simulations of autoimmune system, starting from the activity diagrams provided by the YCIL groups [4] [5] [6] [10]. This inspiration is discussed in the paragraph above where we are able to move from the domain model which represents the AD to the platform model where CD is derived. Our movement to Simulation platform involves using software engineering practices to automate our diagram transformation and simulate it by generating OO code. Our approach is necessary as there is limited work on this process of translating AD to CD simulation for code generation. Code derivation by transformation or generation requires a class model to give the code its structure as a class diagram is a structural diagram.

The organization of our paper starts with the introduction which also detailed the background and inspiration for our work. We discuss the YCIL approach [9] in section 2 and we link it to our work. Our approach detailing our target and what we aim to achieve is discussed in the section 3. Subsequently, we discussed AD and its transformation and how it relates to YCIL [9] and our approach in section 4. Section 5 is on class diagram transformation and its processes. Automation and the tools used is presented in section 6. In section 7, we present a comparison, and it lead us back to the automation of our system. It is followed by code generation for our systems' validation and verification. Section 9 presents an approach to further discussion. The paper concludes with summaries our approach, findings and assumptions.

2 YCIL Approach

This section reviews the simulation work of YCIL group [4] [5] [6] [10] which follows a simulation framework; The CoSMoS "conceptual framework". The framework has various interconnected steps that help to guide our understanding of a system's complexity. CoSMoS framework can guide the UML, math-

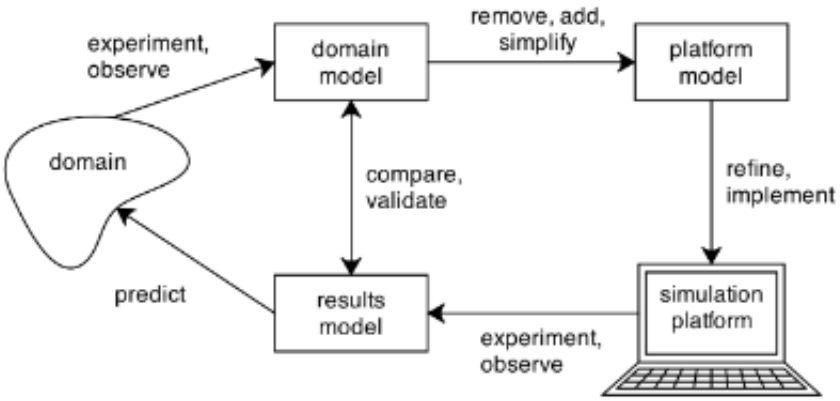

Figure 1: The complex system and Modeling Simulation(CoSMoS) framework followed in modeling our complex system [9].

ematical and in silico modeling for instantiating a system's domain on a high abstraction level [9]. Crucially, as part of the domain and platform modelling, a set of models that highlights the behaviors underpining complexity of the system is created. These models are the basis for implementing the simulation and the result shows a pattern in the real-world system i.e. system of interest [8]. To achieve a fit-for-purpose simulation, the steps in the framework are explained.
Domain — The real world system being modeled. This is the system of interest and information about it is provided by the domain expert.
Domain Model — This is a model that captures the scope of interest and their emergent properties. It identifies and outlines the interactions and behavior of the domain. It is based on the domain expert's analysis. Systems to be simulated are model using a "top-down" methodology [11].
Platform Model — This provides a design model that designs how the domain of interest will be simulated. A "bottom-up" approach is used here for easy construction of the different parts of the domain model to a platform model [11].
Simulation Platform — By using software engineering processes, a simulator is developed from the platform model. The simulation platform provides "executable implementation" [11].
Results Model — The result model is used as a comparison standard for the domain model so as to be able to interpret the simulation results in the context of the real domain [11].

In the YCIL work, behavioral models are created, but are not used directly in generation of code. This means that the changes to the models have to be hand-coded and the correctness of the implementation (validity, compared to the domain model) must be taken on trust.

3 Our Approach

In this section, we discuss the motivation for our work and the goals we want to achieve. Premise and plan for our work is reviewed. Also, we give an overview of our approach which details the process of our simulation.

3.1 Motivation

All the YCIL theses [4], [5], [6] use similar modelling diagrams to but hand-craft code for simulation. The handcrafted code can be clearly written once and easily reused. However, 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.

The ultimate goal is to be able to:

1. Reliably and systematically generate code from YCIL-style diagrams.
2. Reliably and systematically generate revised code when models (diagrams) are modified.
3. To support reuse and modification of the existing platform models and simulations.

3.2 Premise and Plan

Our premise and plan is rooted in using class diagrams for generating OO code.

Our target is listed:

1. If we assume an OO target (YCIL mostly used 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 draft class diagrams.
3. We can validate (and extend) the class model with information from other class diagrams (e.g. state diagrams and use case models).

3.3 Overview of our approach

The steps are described in more detail in section 4 and 5.

The steps to be described in those sections are pointed below:

- Transform activity diagrams (AD) to class diagram (CD).
- Derive sequence diagram (SD) from AD.
- Derive class diagram (CD) from AD.
- Elaborate CD (How).
- Transform CD to SD (needs motivating).
- Compare SDs and update CD.
- Generate code from CD.

The above phases capture our thinking and the key design decisions that we have made. For the purpose of carrying out transformation, an activity diagram provided by Read [5] from the YCIL group [4] [5] [6] is used.

4 Activity Diagram

The discussion of this section raises issues of activity diagram (AD) and what YCIL users are doing with it in relation to its usefulness to our approach. The UML AD defines behavior as a "flow of control" using activity, sequences, conditions and state of an event [12]. A behavior shows sequences of components that use a data flow model and controls [13]. 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 [14]. Activity diagram consists of a parameterized behavior denoted as flows of action. This flows of actions are shown as transition lines or control flow as they help connect one activity to the other. As a behavior diagram, the AD captures the flow of events showing the behavior of the complex system to be simulated by the YCIL group [4] [5] [6] [10].

4.1 Process of transforming an AD to SD

The main motivation for this translation is to produce a more structural representation of behavior models (AD). There exists a limitation on the AD as the

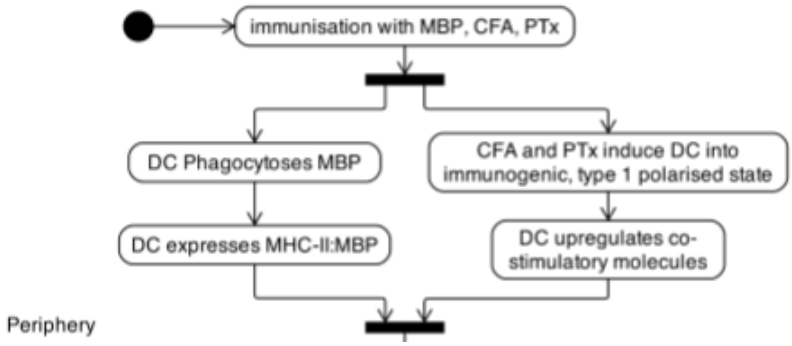

Figure 2: An extract of an activity diagram from [4].

generation of code from a behavior diagram is very difficult. The transformation to SD helps provide basis for the translation of AD to CD which helps provide a class structure for easy generation of code. Sequence are known to lead to different states of an activity and this in turn speaks to the representation of activity as a variant of dynamic models like sequence diagram and state diagram. 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.

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 actors 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 AD's 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.

These steps are followed and we have a resulting sequence diagram that reflects the behavior modeled in our activity diagram. We chose to follow this process because the translation to SD gives concise representation of our activity diagram and provides a basis for validation and verification of our models.

4.2 Graphical representation

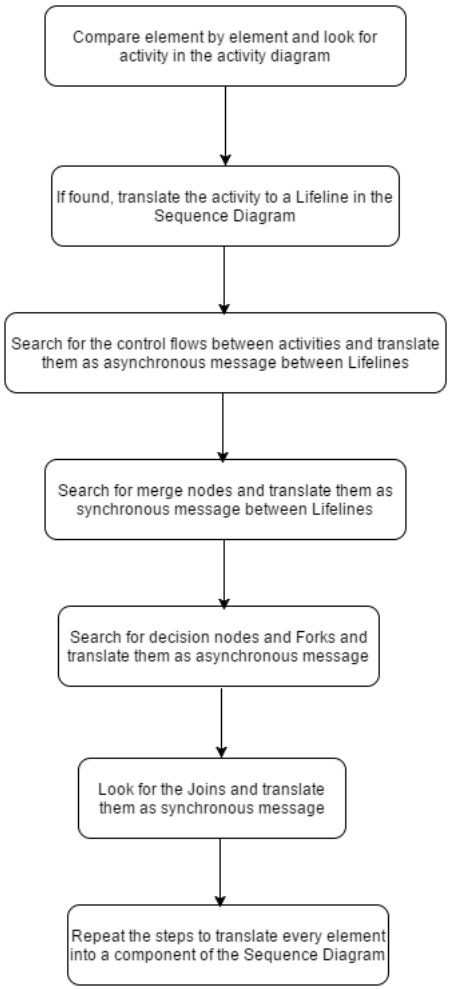

Figure 3: Graphical representation of Transformation Process

The above graphical representation steps were followed to automatically

transform our AD to SD.

4.3 Transformation Pseudo-code

```
1: Set element A, C, M, D, J to 0
2: while (A, C, M, D and J < n-1)
3: add 1 to A
4: if el [A, C, M, D and J] == Key
5: return A, C, M, D, J
6: Else
7: return -1
```

The above pseudocodes represent the flow of events in our AD provided in Figure 2.

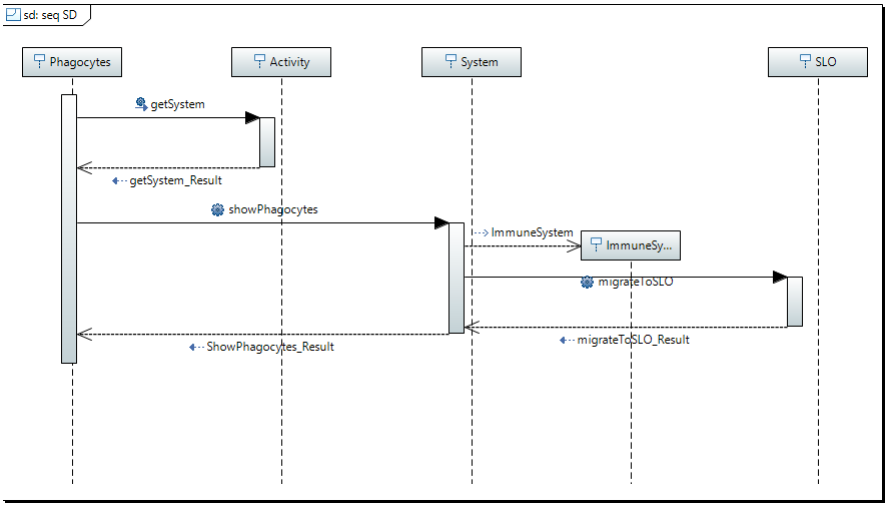

Figure 4: Sequence diagram transformed from the activity diagram in Figure 2.

It is important to note that the AD's joins and forks relate to input and output operations respectively. The control flows link together to form a joint 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. A synchronous message occurs when a sender waits until the receiver is done processing the message before continuing the jobs [15]. An asynchronous message, messages are sent without waiting for a response from the receiver before the job continues [15].

5 Class Diagram Transformation

The complex systems are modeled by the domain expert using AD. This leads us to developing a CD which can be used to generate code as reiterated in section 1 on the purpose of our code generation. The generated code is analyzed and used in understanding the behavior and complexity of our system. To generating reusable and accurate code, a CD should be able to reflect the domain of the existing system correctly. We manually translate our activity diagram to a class diagram in figure 2. It is crucial to transform the provided AD to CD so as to facilitate generation of code which can be used to determine the emergent behavior and complexity of our system. Explicitly, in building our class diagram, we also propose that a class diagram should be derived from use case scenarios, if provided in the system's domain.

5.1 Process of transforming an AD to CD

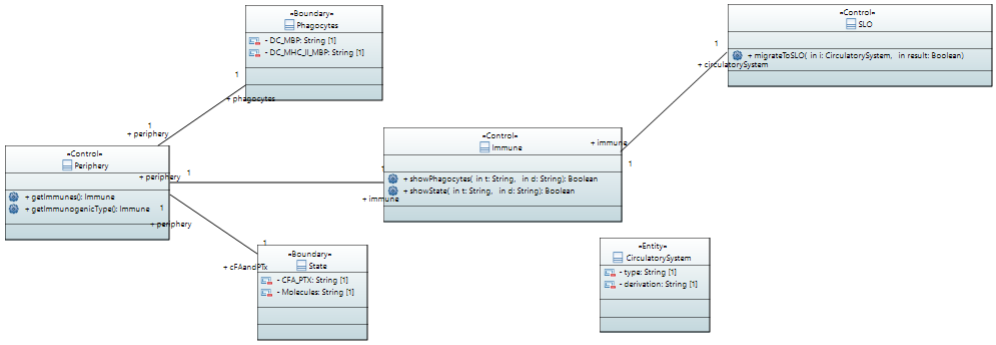

Figure 5: The derived class diagram.

To transform AD to CD, we propose the following steps:

1. Each activity in the AD is translated to a class in the CD.
2. Subactive AD is interpreted as an attribute of a class in the CD.
3. Merges, nodes, forks and joins are mapped to class method.
4. AD elements are translated to a CD element.
5. The control flows between the activities determine the relationship between the classes.

We chose to translate our AC to CD using this process as full representation of the activity as classes facilitate generation of code. The design developed here is justified as a means to generating code whereas, the code becomes a test case to understanding our systems' complexity.

5.2 Process of transforming a 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.

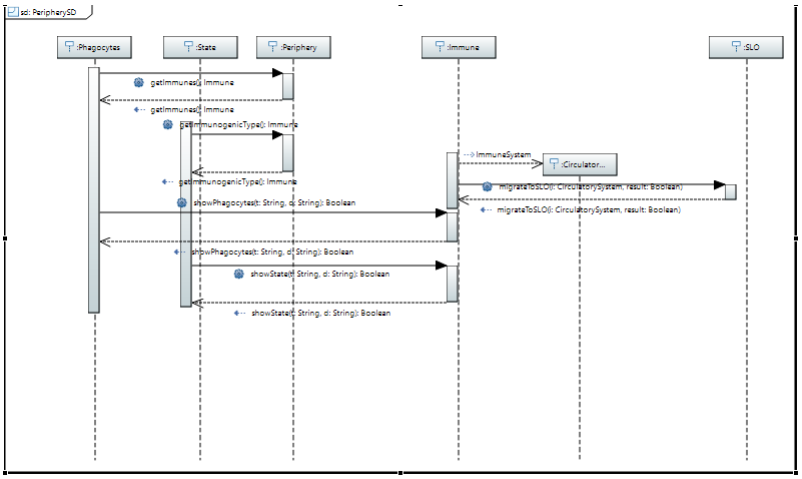

Figure 6: Sequence diagram extracted from the class diagram in Figure 5.

Also, the messages become the operations and property of the class diagram as they are conceptually a form of operation calls. We develop our SD by using the components of our CD. The following steps are proposed:

1. The classes in the CD are represented as Lifelines in the SD. The class may contain many objects and the lifelines is an object.
2. Operation calls of the CD are translated as messages (synchronous or asynchronous) between the lifelines depending on their types.
3. The owner types in the CD (classifiers and association) are translated as synchronous and asynchronous message in the SD.

4. The parameters and the return type that defines the operation and property of the CD is interpreted as the operations of the message in SD.

The two sequence diagrams, derived from the activity and the class diagrams, reflects the interactions, emergent behaviors and the order of operation in the models. They present an abstract view similar to the meta-models derived from models. We automate the listed transformation rules above using the eclipse tools. 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.

6 Automation

The purpose of automating the transformation rules for our models is to provide basis for test cases and effective modeling of the manually designed complex system diagram. By doing this, we are able to provide simulation as defined in section 1 to our complex system.

6.1 Papyrus

[2] An open source model-based engineering tool built on Eclipse. Papyrus enables an integrated environment where UML can be edited. It can be easily customized to provide support for diagram notations, exploring models and model-based simulations. Based on this features, we use papyrus to develop a tool for our complex system modeling. This tool involves developing the AD and the derived SD. Also, the manually translated CD is developed and with the aid of the tool, the SD is developed from the CD. Essentially, Papyrus assists in automating the different diagrams representing our system's complexity.

6.2 EMF Compare

[3] Eclipse modeling framework (EMF) compare enables merge and comparison for instances of a model. It provides a customizable, steady, reusable and generic tool support for comparing and merging model. We use EMF compare for merging and comparison of the models developed using the Papyrus. With EMF compare, a merging of model components gives us confidence in the model and the code to be generated. The comparison of our derived SDs is done using EMF compare. This comparison authenticates the translated CD and the code

[2] Papyrus modeling environment: https://eclipse.org/papyrus/
[3] EMF Compare: https://www.eclipse.org/emf/compare/

generated from it. We use EMF compare as it is an open source tool that can be used for anything with Java. Also, it provides a detail visualization of the compared entity either as a text compare or a model compare. We believe other alternative options like Guiffy SureMerge and JEdit JDiff are not as rigid and supportive for diagram comparison.

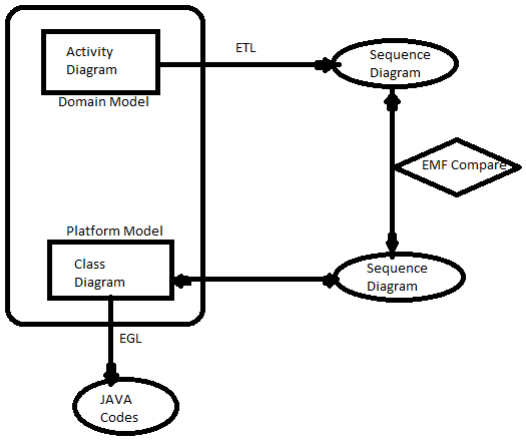

Figure 7: Our process for transforming an AD to a SD. Comparing the two resulting SDs validate the CD, prior to generation of OO Java code.

7 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 have 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 (activity diagram). We believe that a relationship exists between the two sequence diagrams as they represent the same model of the domain system i.e. the AD and the translated CD. This relationship gives us confidence in the correctness of the translation between our AD to the CD. We compare the two sequence diagrams using an eclipse tool, EMF Compare and URL. EMF Compare is a tool designed to support comparisons of large fragmented models [16]. 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.

8 Code Generation

With the transformations executed in the papyrus tool, after the comparison of the sequence diagrams using EMF Compare, we proceed to generate Java code from the validated class diagram. The CoSMoS framework shows that the action of generating code means moving from platform model to the simulation platform where the code is generated [8]. 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 code. The generated code is tested thereby revealing patterns of behavior which is translated into a result model for the domain expert to understand.

```
1
2
3  public class Phagocytes {
4
5      /**
6       *
7       */
8      private String DC_MBP;
9      /**
10      *
11      */
12     private String DC_MHC_II_MBP;
13     /**
14      * Getter of DC_MBP
15      */
16     public String getDC_MBP() {
17          return DC_MBP;
18     }
19     /**
20      * Setter of DC_MBP
21      */
22     public void setDC_MBP(String DC_MBP) {
23          this.DC_MBP = DC_MBP;
24     }
25     /**
26      * Getter of DC_MHC_II_MBP
27      */
28     public String getDC_MHC_II_MBP() {
29          return DC_MHC_II_MBP;
30     }
31     /**
32      * Setter of DC_MHC_II_MBP
33      */
34     public void setDC_MHC_II_MBP(String DC_MHC_II_MBP) {
35          this.DC_MHC_II_MBP = DC_MHC_II_MBP;
```

Figure 8: Code generated for the Phagocytes class from the CD in Figure 5.

9 Discussion

We made use of the CoSMos approach to provide a modeling simulation for the YCIL AD. This simulator has proved effective allowing us to generate OO code so as to understand the emergent behaviors of the modelled system. To be able to cover more ground, we aim to be able apply the approach we have created to more YCIL ADs. In future, we will seek to rationalize our code by exploring ways to link "activity" derived class that are in fact objects of one class. Further work includes comparing our code to the handcrafted code from

the PhD researchers as well seeing our objects as agent, thereby developing our approach to suit Agent Based Modeling (ABM).

10 Conclusion

There is no current standard method agreed for transforming activity diagram to class diagram. Starting from the modeling of complex autoimmune systems, we have developed an approach and tool for doing this. In doing that, we showed a detailed and simple process of transforming activity diagram to a class diagram thus enabling the generation of executable and reusable code. The modeling is made as simple as possible to accommodate our work, and the patterns in the generated could can be easily related to a result model which is understandable to the users (domain experts). This process was automated using the eclipse papyrus tool. The tool enables us to draw the modeled and translated diagram. This diagram helps us in generating the code observed to understanding our complex system. The tools used are open source which enables for easy reusability of our tool and approach.

As the UML afforded us the use of different diagrams, our approach, we argue, enables us to develop a tool that capture the domain of interest. We further argue that it can be used for a system of similar process and easy transformation of activity diagram to class diagram.

References

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

[2] 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.

[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] 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.

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

[6] 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.

[7] S. Luke, "Multiagent simulation and the MASON library," 2011.

[8] 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.

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

[10] 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 *ICECCS 2010: Fifteenth IEEE International Conference on Engineering of Complex Computer Systems*. IEEE Press, March 2010, pp. 276–285. [Online]. Available: http://www.cosmos-research.org/docs/iceccs2010-reflections.pdf

[11] 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.

[12] 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

[13] OMG, "OMG unified modeling language version 2.5," 2015. [Online]. Available: http://www.omg.org/spec/UML/2.5/

[14] R. Eshuis and R. Wieringa, "Comparing Petri Net and Activity Diagram Variants for Workflow Modelling: A Quest for Reactive Petri Nets," in *Petri Net Technology for Communication-Based Systems: Advances in Petri Nets*, ser. Lecture Notes in Computer Science, H. Ehrigh, W. Reisig, G. Rozenberg, and H. Weber, Eds. Berlin/Heidelberg, Germany: Springer, 2003, vol. 2472, pp. 321–351. [Online]. Available: http://doc.utwente.nl/61800/

[15] F. Martin, "Uml sequence diagram," 2014. [Online]. Available: http://www.informit.com/articles/article.aspx{?}p{=}169507{&}seqNum{=}3

[16] EMFCompare, "EMF Compare-UserGuide Version 3.1.0.201506080946." [Online]. Available: https://www.eclipse.org/emf/compare/overview.html