A FRAMEWORK FOR CONSTRUCTING AGENT-BASED AEROSPACE SIMULATIONS USING MODEL TO TEXT TRANSFORMATION

B. Eren Dere
Roketsan Inc. PO Box 30, Elmadag, Ankara, Turkey
and Department of Computer Engineering,
Middle East Technical University, Ankara, Turkey
eren.dere@roketsan.com.tr

Bilge Kaan Görür
Roketsan Inc. PO Box 30,
Elmadag, Ankara, Turkey
kaan.gorur@roketsan.com.tr

Halit Oğuztüzün
Department of Computer Engineering,
Middle East Technical University
Ankara, Turkey
oguztuzn@ceng.metu.edu.tr

ABSTRACT

Engineering simulations, such as simulations in aerospace industry, have a complicated nature. In order to cope with those complications and construct simulations by using loosely-coupled components, one of the approaches is the agent-based modeling and simulation approach which ensures the separation of concerns. Engineering simulations are extensively being used by researchers coming from various disciplines. These researchers are used to use model-driven tools and languages. Unfortunately, several problems may arise when only model-driven tools and languages are applied, including performance, scalability and productization issues. These problems can be mitigated by using well designed frameworks that are implemented with conventional programming languages. In this paper, a framework which builds the simulation architecture with the help of a model-based language and transforms that into code is proposed. Our study ensures code generation of an enterprise-level simulation implementation from a software architecture which can be designed using SysML.

Keywords: Model to text transformation, code generation, SysML, simulation design

1 INTRODUCTION

In parallel with developments in model-driven software engineering, model-driven design and development tools started to be employed by many engineering industries. Especially the industries that work on complex engineering problems, such as aviation and automotive, benefit from model-driven tools very often, because those tools aim to improve simulation design and development processes. For engineering simulations, Simulink (2020), Xcos (2020) and Open Modelica (2020) are some of the popular tools that allow users to design their simulations as a collection of block diagrams. By using those diagrams, developers easily implement and run their simulations by without thinking about the processes outside their scope. However, using those model-driven tools comes with two main disadvantages. The first one is the scalability problem. Interpreted languages provided by model-driven applications may cause simulations to slow down comparing to conventional programming languages because of the generic structure of model-driven tools. The performance issue becomes worse when it comes to simulate multiple entities in the same scenario. Secondly, there are maintainability problems with multi-agent systems while using model-driven

development tools. For instance, when a designer wants to add identical entities to the simulation, he/she has to copy all elements in the model multiple times. Then, the model becomes harder to be refactored and maintained. In addition, the object oriented compatibility of these tools which is promising for multi-agent simulations are insufficient (Görür and Çallı 2018).

An agent based simulation which is implemented with conventional programming languages can deal with performance and scalability issues. However, for most developers who are not familiar with software development principles, an unusual approach of simulation development may be seen complicated and time consuming. Trying to code essential parts of a simulation will prevent researchers and engineers focusing on the actual field of interest. For example, implementing the connections among simulation entities (such as subsystems in an air vehicle) and figuring out how the building blocks should constitute the whole simulation is one of the challenging issues for the model designers with conventional languages. These difficulties cause model developers to ignore conventional programming languages completely.

A simulation architecture framework that is programmed with an object oriented programming language, such as C++, may mitigate scalability and maintainability problems stated above. Secondly, to make designs easier and in a more robust fashion with non-programmer friendly properties, a visual modeling tool and a code generator which acts like a bridge between the simulation infrastructure and model-driven tool are needed. In this paper, we propose a framework that allows developers to design their models in a model-driven design tool. The framework can generate C++ code that is compatible with an agent-based simulation infrastructure that we have already implemented. The generated code provides a skeleton which can be compiled and executed after modifying it by entering initial values of agents in the target framework. Therefore, developers do not have to start developing their models from scratch. Instead of struggling with details of the simulation engine, interaction of simulation entities and unrelated concepts, developers will be able to focus on only their own duties.

In this study, we aimed to accelerate the simulation development process by starting from the early design phases of agents. Therefore, design to code transition will release the burden on model developers who do not have expertise in object oriented programming. We used Systems Modeling Language, also known as SysML (2020), to design agent-based simulations. Although SysML is not the only solution for model to code transformation, we preferred SysML, because it is a widely used standard language for system designers. It has block definition and internal block diagrams which designers are used to see for defining behavior of systems. Moreover, all parts of the model to code generation have been done by open source tools, including Eclipse Papyrus (2020), Acceleo (2020).

2 RELATED WORK

Model-driven engineering increased its popularity especially in the last two decades, due to its attractiveness to stakeholders who do not have to have deep knowledge on computational science. Despite this advantage, there are still many gaps that should be studied on model-driven engineering. Most of the problems have been addressed by (Zeigler, Mittal, and Traore 2018), (Tolk, Diallo, and Mittal 2018) and (Kautz, Roth, and Rumpe 2018). Bringing model-based systems engineering with various simulation approaches together, toughness of creating complex code generator software, needs to domain knowledge for modeling, and lack of tools for complex engineering systems are some of those problems.

Some studies in the literature focuses on model-to-model transformation, such as (Ledett et al. 2015) and (Chabibi, Anwar, and Nassar 2018). The underlying motivation of model-to-model transformations can be various, such as co-simulation. It is also possible to use the target platform for an intermediate solution for a next step. For instance, after a SysML to Simulink transformation, C++ code can be generated by using Simulink facilities (Vanderperren, and Dehaene 2006). A solution for generating a Simulink model from SysML has been proposed to help engineers to deal with difficulties while designing complex systems by Chabibi, Douche, Anwar, and Nassar (2018). In addition to model-to-model transformation from a SysML

model, they also generated some Matlab code with Acceleo templates to be used to generate the Simulink model.

Maheshwari et al. (2018b) tackles with integrating SysML and agent-based modeling approach to allow teams to trace evolution of systems architecture rapidly. Maheshwari et al. (2015) showed how a conceptual model in SysML can be used to get an executable agent-based simulation in Matlab. An approach using SysML diagrams in order to generate VHDL codes for reverse engineering purposes is introduced by Boutekkouk and Zaidi (2015). They defined their rules for code generation using Java language. On the other hand, there are some other studies that also handle the common problems of simulation with model-driven engineering. For example, a solution for generating HLA (2010) based distributed simulations using model driven architecture has been proposed by Bocciarelli, D'Ambrogio, and Fabiani (2012) to overcome the limitations of implementing complex distributed simulations. They also used SysML for the modeling studies. Moreover, in study (Bocciarelli et al. 2019) , MONADS method (Model-driveN Architecture for Distributed Simulation) which allows one to generate HLA based distributed simulation code from SysML models by using a chain of model-to-model and model-to-text transformations is proposed.

3 TARGET SIMULATION PLATFORM

In this study, the target platform of the code generation is our own simulation infrastructure that has been already implemented in C++ by considering object oriented design approaches. SysML blocks which will be designed while constructing the simulation model are represented as instances of classes in the infrastructure. The most important classes of the infrastructure are Simulation, Module, Agent and Connection and Engagement. In order to describe the code generation process in details, a brief summary of the simulation infrastructure is given in this section. A basic class diagram of our simulation infrastructure is shown in Figure 1.

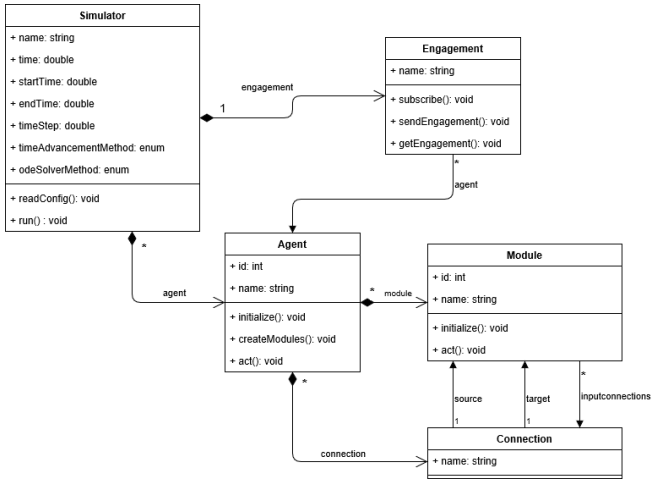

Figure 1. Class Diagram of Simulation Infrastructure

The Simulation class can be considered as the root object of our simulation infrastructure. It harbors agents, the engagement and the time advancing mechanism. Agents refer to autonomous entities in the simulation while the engagement tackles with the interaction between agents. The time advancing mechanism is responsible for driving the simulation engine by implementing both discrete and continuous simulation approaches. A discrete simulation can be executed by one of the time-stepped execution or the event-driven execution (Fujimoto 1999), while a continuous simulation is interested in using some differential equation solvers. A user can specify time management related parameters by using input files. Other features of the simulation infrastructure including plotting and logging options can be also configured by using these input files.

There are two kinds of simulation entities in the infrastructure: modules and agents. In terms of aerospace simulations, a module refers to a subsystem or a computational block that can be designed partially independent from the other components. Propulsion, aerodynamics, flight mechanics, guidance and autopilot are some of the components that can be handled as modules. The module class is an interface for all entities that will be deployed to the simulation. This interface has three main methods that are left to the developers to be implemented: initialize, act and postProcess. The initialize function is designed in case the initial conditions of the attributes are determined. The act function specifies the behavior of the module at the each time step while simulation is running. After the simulation is completed, the postProcess function runs to allow user to examine the results of the simulation. For instance, plotting, saving the simulation data and similar tasks can be handled in this function by a developer.

Agents are the autonomous entities in our simulation infrastructure which can be vehicles, radars or satellites. They inherit characteristic properties of modules and are built by connecting different modules which work together. The Agent class is also an interface to all kind of agents in a simulation. In the initialize function of an agent the developer should set the connections among its own modules. Engagements between the agents are also being set in this method. An agent's act function is also responsible for modules to be traversed and acted. In addition to the initialize and the act functions, an agent should also implement the createModules function which creates module instances that are harbored by the agent.

Connections determine the data flow between modules. In the simulation infrastructure, they are responsible for sending the necessary variable from the source module to the target. A connection object carries attributes named as Service Functions, which are leveraged from function pointers of "get" methods inside the source module. Therefore, every entity has a map which stores the connections that is needed to be used for inputs.

Engagement class ensures the connection between agents. For each step during the simulation, engagement information is being distributed to subscribed related modules before their act methods are run. Although the engagement approach is very similar to connections that has been defined above, it is intentionally differentiated from connections. The main reason of this differentiation is keeping the simulation infrastructure ready for distributed simulation concerns. Therefore the engagement approach was implemented in a compatible manner with publish-subscribe mechanism. There are two important duties of the Engagement class. The first one is managing the publication-subscription tables among agents. The subscription can be done by agents by indicating either a specific agent or an engagement data type, such as position, sensor, or radar cross section (RCS) data. The other responsibility of the Engagement class is providing engagement data to right agents at right time. Providing data operations are mostly abstracted from simulation entities. Simulation entities are not aware of how the data is provided to itself; instead they only know data provided by the Engagement. From these aspects, it can be said that the Engagement class works similar to the Run-Time Infrastructure in High Level Architecture.

4 MODEL DESIGN IN SYSML

The code generation was done by examining models in BDD (Block Definition Diagram) and IBS (Internal Block Diagram) of SysML. A BDD represents the architectural structure of the model, while an IBD focuses on the behavioral structure of the agents. Figure 2 shows the mapping between these diagrams and the simulation platform. The mapping is done by considering the studies of Macal and North (2014) and Maheshwari et al. (2018a).

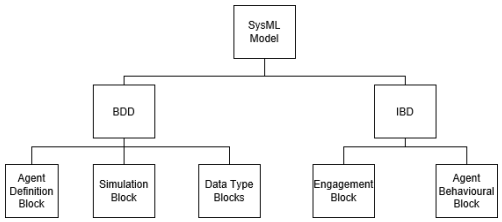

Figure 2. The mapping between SysML diagrams and the simulation infrastructure.

4.1 BDD Elements

A BDD diagram has three kinds of elements that are Simulation Block, Agent Definition Block and Data Type Blocks (Figure 3). While generating code, every element inside the BDD is traversed and C++ code is generated accordingly. Simulation Block describes which agents will be used in the simulation scenario. It also carries the information about simulation parameters, such as time management mechanism and user configurations. Data Type Blocks are implemented in order to define the types of the attributes of simulation entities. These types are generally the primitive types of C++ language and some other types which are implemented in the simulation infrastructure. Agents are described as separate Agent Models and contain module definitions that will be implemented.

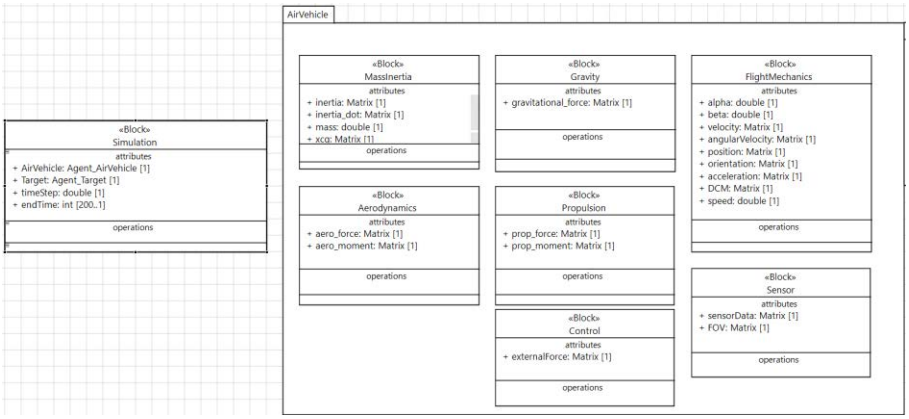

Figure 3. Simulation Block and Agent Definition Block Representation

4.2 IBD Elements

An IBD has two main elements that are Agent Behavioral Block and Engagement Block (Figures 4-5). Agent Behavioral Block represents how act in each simulation step. Inside this element, modules are connected to each other. During the code generation, the data flow between modules of an agent are generated by considering elements in the Agent Behavioural Block.

Interactions between agents are described in the Engagement Block. There is a predefined Engagement Service Block and it has three ports named RCS, SENSOR and POSITION. Model designers use references of modules that are defined in the Agent Architectural Structure in the BDD. In the generated code, the agents interact with each other's by allowing these modules to connect with the Engagement class. There is no direct communication between agents. All data should pass through the Engagement Service Block and distributed to target modules. There is a publish-subscribe relationship between the source and target modules. For example, if a module is publishing its attribute, then it means that this module sends data to the engagement service block. The engagement block manages how to deliver this data to the related modules of the subscribed agents.

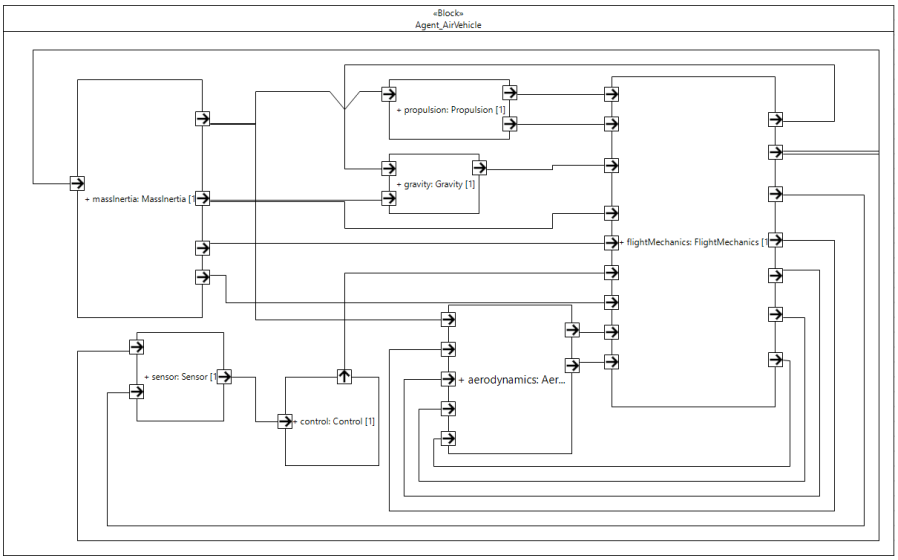

Figure 4. Internal Block Diagram of Object

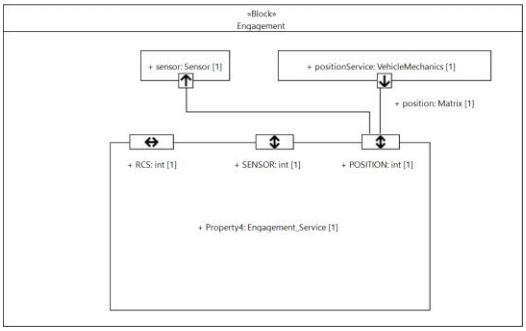

Figure 5. Engagement Block

5 CODE GENERATION

To generate C++ code from the SysML model, we used Acceleo language in this study. Generated code becomes a C++ project which is compatible with an already implemented simulation infrastructure. However, because Acceleo uses templates for code generation, it is only suitable for generating plain text. This means that Acceleo syntax is not the best option for implementing algorithms. So, whenever we encountered problems during the implementation of code generation algorithms with Acceleo, we appealed Java by invoking our Java-based code generation methods inside the Acceleo code There are three main templates used in the Acceleo code, that will be described in the next subsections. Figure 6 demonstrates our high-level code generation template that is written in Acceleo.

```
# Generate main.cpp for the simulation
# by reading the simulation block of the model
generateMainCpp(model.getSimulationBlock())

# Traverse agents inside the model
for agent in model.contents():

    # Generate .h and .cpp files for agents
    generateAgentHeader(agent)
    generateAgentCpp(agent)

    # Traverse modules inside the agent
    for module in agent.contents():

        # Generate .h and .cpp files for modules
        generateModuleHeader(module)
        generateModuleCpp(module)

    /for

/for
```

Figure 6. Pseudo Code for Acceleo code generation.

5.1 Generation of Main Simulation Code

The first step of the code generation process is generating the main function that is the entrance of the desired simulation program. During this step, agents that have been described inside the simulation block of SysML model are added into the main.cpp. A configuration file is also created in this step for main simulation parameters, such as simulation time related parameters, by considering the other attributes of the Simulation block. This process is applied by following Acceleo steps respectively:

- ***Step 1 – Generation of included files:*** Necessary header files of agents and the simulation infrastructure are included in the main.cpp.
- ***Step 2 – Declaring a simulation object and creating a configuration file:*** A global simulation object is created. This object is going to be used by all agents and modules inside the simulation. Attributes of the simulation object are set by reading the configuration file. The configuration file is a simple text file and generated in this step.
- ***Step 3 – Generation of agent objects:*** Instances of agents that have been defined in the BDD are generated in the main function and added into the simulator object as entities.
- ***Step 4 – Generation of the simulation runner:*** After all the steps above are completed, the code that executes the simulation is generated.
- ***Step 5 – Generation of delete commands:*** At the end of the main function commands for deleting the objects are added.

5.2 Generation of Agent Code

After simulation block has been processed, Acceleo code starts traversing the agent blocks inside the BDD and generates agent classes by invoking generateAgentHeader and generateAgentCpp templates. The generated .cpp and .h files are stored under a folder whose name is the same with the agent block. The agent generation process is applied by following Acceleo steps respectively. The first two rules provides generating the header file, while rest of the rules provides generating the source file.

- ***Step 1 – Generation of included files for header file:*** In addition to including parent agent class in the simulation infrastructure, the header files of modules which are found by traversing the contents of the agent in IBD are also included.
- ***Step 2 – Generation of agent class:*** The agent class is generated with the necessary attributes and methods that have been defined in the BDD. In addition to getter methods and user defined methods, a createModules method is also created for every agent. This method is responsible for creating and initializing modules of the agent and establishing connections between modules.
- ***Step 3 – Generation of included files and extern variables for source file:*** Similar to the Step 1, including some necessary header files are provided in this step. Basically, generated .cpp files include some standard header files from the simulation infrastructure (i.e. “Simulator.h”, common

data structure headers, etc.) and the related header file that becomes ready after Step 2 is completed. In addition to including headers, a reference for the Simulator object that has been already defined in main.cpp is generated.

- ***Step 4 – Creating module instances:*** All module attributes of the agent are instantiated in this function. Their names, launch times and exit times are described. Moreover, if the module is involved in engagement with a module of another agent, then according to whether that module is a subscriber or publisher, suitable code is generated.
- ***Step 5 – Establishing connections:*** Connections between modules that have been created in the previous step are established in this step. Connection information is stored in connection maps that are defined inside the modules, to be used later.
- ***Step 6 – Determining the execution order of modules:*** After connections are established, the execution order of modules is determined. In our simulation infrastructure, modules are executed in the order in which they are added to the simulation engine. Therefore, the entity order is determined from the IBD by using depth first search graph algorithm.

The initialize and the act functions of an agent are not created intentionally. Instead, agents use them from its parent class, namely Agent. Agent class' initialize function calls the initialize function of its modules in the correct order which is determined in Step 6. Similarly, Agent class' act function calls the act function of its modules in the correct order which is determined in Step 6. If a developer needs to specialize these two functions, he/she can override these methods after all the code is generated.

5.3 Generation of Module Code

After agent codes are generated, modules of each agent are traversed and their classes are generated by invoking generateModuleHeader and generateModuleCpp templates. The module generation process is applied by following Acceleo steps respectively. The first two rules provides generating the header file, while rest of the rules provides generating the source file.

- ***Step 1 – Generation of included files for header file:*** The parent module class, namely Module, is included.
- ***Step 2 – Generation of module class:*** The module class is generated with the necessary attributes and methods that have been defined in the BDD. In addition to getter methods and user defined methods, an initialize method and an act method are also created for every module. Generated getter methods provides function pointers to connection objects. On the other hand, the engagement block inside the SysML model is examined. If the module is publishing any data to the engagement services, then the sendEngagementData function is generated accordingly.
- ***Step 3 – Generation of included files and extern variables for source file:*** Similar to the Step 1, including some necessary header files are provided in this step. Basically, generated .cpp files include some standard header files from the simulation infrastructure (i.e. "Simulator.h", common data structure headers, etc.). In addition to including headers, a reference for the Simulator object that has been already defined in main.cpp is generated.
- ***Step 4 – Generation of the initialize function:*** From the SysML model, attributes of the module are iterated. Initial values of those attributes are left blank for user to fill in. After the code generation process, the developer should set the initial values by modifying the source file.
- ***Step 5 – Generation of the act function:*** Connections inside the agent are traversed and the ones coming to the module as inputs are detected. Then, all the input data are stored in local variables by fetching the inputs coming from connection object by calling the source module's get method. After code generation is completed, the designer should explicitly write what calculations need to be done with these local variables.
- ***Step 6 – Generation of the sendEngagementData function:*** If the Acceleo code detects that the module is involved in engagement in Step 1, then sendEngagementData function is generated to allow the developer to specify what data will be sent.

6 CASE STUDY

To demonstrate how code generation from SysML is done, an example air vehicle simulation model has been designed in SysML and the code generation is executed. Modeling in SysML is made by using Eclipse Papyrus environment. In this simulation, time stepped-execution is used for time advancement method and no ODE solver is used for the sake of simplicity.

In this case study, there are two agents in this simulation named as AirVehicle and Target. AirVehicle tries to arrive at Target which is a moving agent. The definition of these two agents is specified in Figure 3. In this case study, AirVehicle agent has these 7 modules Gravity, Aerodynamics, Propulsion, MassInertia, Control, Sensor and FlightMechanics. Figure 3 also shows the Agent Definition Block Representation of Air Vehicle. Attributes of each module is also defined in this view. Internal Block Diagram of AirVehicle is given in Figure 4. This view shows the behavioral logic of AirVehicle by establishing connections among modules.

For this case study we kept the business logic of Target agent very simple. The Agent Definition Block Representation and Internal Block Diagram of Target agent is given in Figure 7.

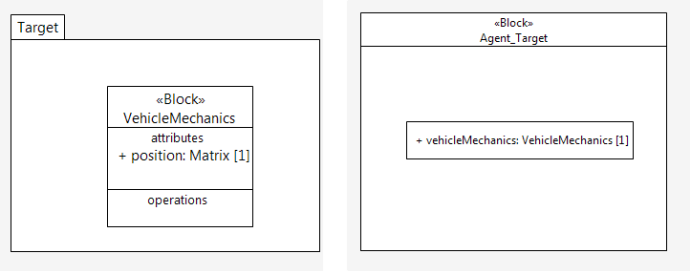

Figure 7. Agent Definition Block and Agent Behavioural Block of Target agent.

The interaction between AirVehicle and Target agents is defined in the Engagement Block in Figure 5. In this representation, VehicleMechanics module of Target publishes its position variable to the Engagement. On the other hand, Sensor module of AirVehicle subscribes to data whose type is POSITION.

After the code generation from the SysML model above is processed, all header and source files are created in a hierarchy as shown in Figure 8. Some examples of generated code for Sensor module of AirVehicle agent, VehicleMechanics module of Target agent and Main.cpp are shown in Figures 9-10. Details of the generated code files can be seen on the GitHub repository (SysML2Code 2020).

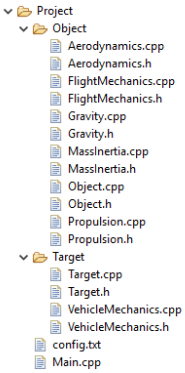

Figure 8. File structure of generated C++ project

```cpp
/*
 * Main.cpp
 *
 * Created on : 02-03-2020 03:22
 *     Author : eren.dere
 */

#include <Simulator.h>
#include "AirVehicle.h"
#include "Target.h"

void readSimulationData();

Simulator *simulator;

int main(){
    simulator = new Simulator();
    simulator->readConfig();

    /**
     * Create entities and run
     */
    AirVehicle * airvehicle1 = new AirVehicle();
    airvehicle1->setName("airvehicle1");
    airvehicle1->setLaunchTime(0);
    airvehicle1->createModules();
    simulator->addEntity(airvehicle1);

    Target * target1 = new Target();
    target1->setName("target1");
    target1->setLaunchTime(0);
    target1->createModules();
    simulator->addEntity(target1);

    simulator->run();

    return 0;

}
```

Figure 9. Generated Main.cpp

7 CONCLUSION AND FUTURE WORK

In this study, we have proposed a solution for creating agent based simulations with model driven approach using SysML and Acceleo. This way, some implementation details become transparent to engineers and researchers, allowing them to focus on their fields instead of spending time on struggling with unrelated parts of the simulation. Our case study showed that generated code is a promising starting point for simulation development process. After the user can specify the initial values of the variables in code, the generated code can be easily to compiled and run. During the study, various problems have been encountered. In model to code transformation, using SysML diagrams decreased the ability of creating convenient model to code transformation algorithms. Therefore, extensive amount of predefined rules had to be introduced. For instance, in order to declare a certain block to be recognized an Agent during the code generation, user should type “Agent_” in front of the block’s name. As a result, these rules caused disadvantages for modelers while creating SysML models. Instead of using SysML, we aim to create a custom metamodel from scratch which is directly associated with the simulation infrastructure. We plan to use Ecore tools in Eclipse Modeling Framework (2020) for creating the metamodel. Secondly, by using Sirius (2020), an editor for model implementations is planned to be implemented and embedded on the custom metamodel. This way, encountered disadvantages when SysML is used will be mitigated and optimal conditions for model driven code generation will be ensured. In addition, as introduced in (Möller et al. 2016) and (Möller et al. 2017), an extension to this study by integrating Space Reference FOM (SRFOM) for simulation interoperability is considered.

Another issue of the study is going back from the C++ code to SysML model. If a developer wants to modify the model after he/she codes business logic, he/she cannot go back to SysML model. Instead, he/she has to generate code again and migrate the business logic to new code. We are planning to figure it out by exactly separating generated code from the business logic that will be developed by developers.

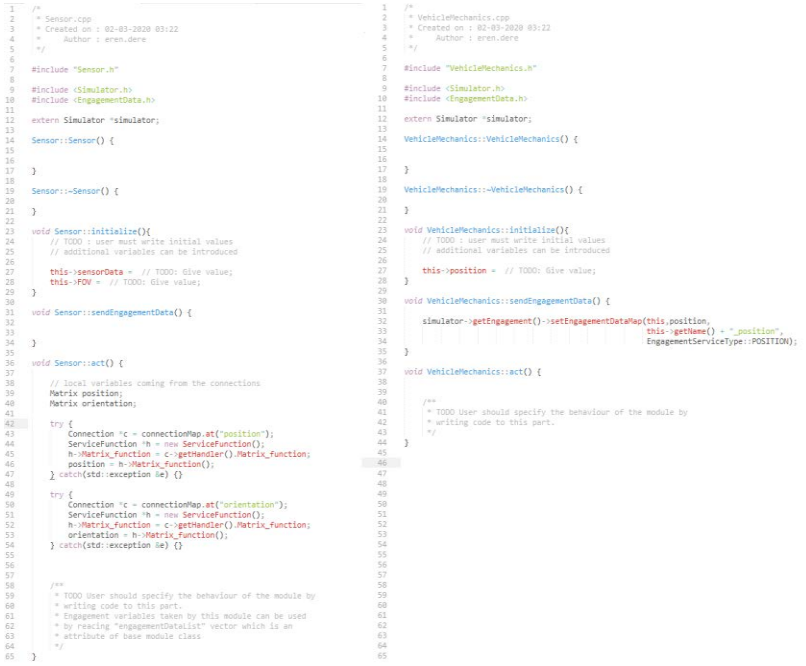

Figure 10. Generated Sensor.cpp and VehicleMechanics.cpp files

REFERENCES

Acceleo 2019. "Eclipse Foundation, Acceleo". https://www.eclipse.org/acceleo. Accessed Mar. 04, 2020.

P. Bocciarelli, A. D'Ambrogio, A. Falcone, A. Garro, and A. Giglio. A model-driven approach to enable the simulation of complex systems on distributed architectures. *SIMULATION: Transactions of the Society for Modeling and Simulation International*, 95(12), 2019. doi: 10.1177/0037549719829828. url: https://doi.org/10.1177/0037549719829828.

Bocciarelli, P., A. D'Ambrogio, and G. Fabiani, 2012. "A model-driven approach to build HLA-based distributed simulations from SysML models." In *SIMULTECH*, pp. 49-60.

Boutekkouk, F., and O. Fartas, 2015. "Automatic generation of SysML diagrams from VHDL code." In *Symposium on Complex Systems and Intelligent Computing (CompSIC)*.

Chabibi, B., A. Anwar, and M. Nassar. 2018. "Towards a Model Integration from SysML to MATLAB/Simulink." *Journal of Software* vol. 13, pp. 630-645.

Eclipse Modeling Framework 2020. "Eclipse Modeling Framework". https://www.eclipse.org/modeling/emf/. Accessed Mar. 04, 2020.

Fujimoto, R. M. 1999. *Parallel and Distributed Simulation Systems*. New York, USA, John Wiley & Sons, Inc.

Görür, B. K., A. N. Çallı, 2018. "An Object Oriented Agent Based Framework for Modeling and Simulation in Aerospace", *2018 AIAA Modeling and Simulation Technologies Conference – AIAA Scitech Forum*.

HLA 2010, "1516-2010 IEEE Standard for Modeling and Simulation (M&S): High Level Architecture (HLA) – Framework and Rules".

Kautz, O., A. Roth, and B. Rumpe, 2018. "Achievements, Failures, and the Future of Model-Based Software Engineering". In *The Essence Of Software Engineering*, edited by V. Gruhn and R. Striemer, pp. 221 – 236. Berlin, Springer.

Ledett, J., S. Cam, B. K. Gorur, O. Dayibas, H. Oguztuzun, L. Yilmaz, and A. E. Smith, 2015. "A Hybrid Transformation Process for Simulation Modernization and Reuse via Model Replicability and Scenario Reproducibility". In *Proceedings of the 2015 Alabama International Simulation Conference*.

Macal, C., and M. North, 2014. "Introductory tutorial: Agent-based modeling and simulation". In *Proc. of Winter Simulaion Conference*. pp. 6 – 20.

Maheshwari, A., A. Raz, A. Dervisevic, R. Campbell, D. A. DeLaurentis, W. Colligan, A. Murphy, and O. Kolawole, 2018a. "Minimum SysML Representations to Enable Rapid Evaluation using Agent-Based Simulation". *INCOSE International Symposium* vol. 28, pp. 1706 – 1719.

Maheshwari, A., A. Raz, D. A. DeLaurentis, A. Murphy, and O. Kolawole, 2018b. "Integrating SysML and Agent-Based Modeling for Rapid Architecture Evaluation". *Insight* vol. 21, pp. 47 – 51.

Maheshwari, A., C. R. Kenley, and D. A. DeLaurentis, 2015. "Creating Executable Agent-Based Models Using SysML". *INCOSE Internatiional Symposium* vol. 25, pp. 1263 – 1277.

Möller, B., Garro, A., Falcone, A., Crues, E. Z., & Dexter, D. E. (2016, September). Promoting a-priori interoperability of HLA-based Simulations in the Space domain: the SISO Space Reference FOM initiative. In 2016 *IEEE/ACM 20th International Symposium on Distributed Simulation and Real Time Applications* (DS-RT) (pp. 100-107). IEEE.

Möller, B., Garro, A., Falcone, A., Crues, E. Z., & Dexter, D. E. (2017, October). On the execution control of HLA federations using the SISO space reference FOM. In *2017 IEEE/ACM 21st International Symposium on Distributed Simulation and Real Time Applications (DS-RT)* (pp. 1-8). IEEE.

OpenModelica, 2020. "OpenModelica". https://www.openmodelica.org/. Accessed Mar. 04, 2020.

Papyrus 2020. "Eclipse Foundation, Papyrus". https://www.eclipse.org/papyrus/. Accessed Mar. 04, 2020.

Simulink 2020. "Mathworks MATLAB/Simulink". https://ch.mathworks.com/. Accessed Mar. 04, 2020.

Sirius 2020. "Eclipse Foundation, Sirius". https://www.eclipse.org/sirius/. Accessed Mar. 04, 2020.

SysML 2020, "OMG Systems Modeling Language". https://www.omgsysml.org. Accessed Mar. 04, 2020.

SysML2Code 2020. "SysML2Code in Github". https://github.com/yatiyr/Sysml2Code/. Accessed Mar. 04, 2020.

Tolk, A., S. Diallo, S. Mittal, 2018. "The Challenge of Emergence in Complex Systems Engineering". In *Emergent Behavior in Complex Systems Engineering: A Modeling and Simulation Approach*, edited by S. Mittal, S. Diallo and A. Tolk, pp. 79–97. John Wiley & Sons Inc.

Xcos 2020. "Scilab". https://www.scilab.org/software/xcos. Accessed Mar. 04, 2020.

Vanderperren, Y., and W. Dehaene, 2006. "From UML/SysML to Matlab/Simulink: Current State and Future Perspectives". In *Proc. Design, Automation, and Test in Europe (DATE)*, Munich, Germany.

Zeigler, B. P., S. Mittal, and M. K. Traore, 2018. "MBSE without Simulation State of the Art and Way Forward". *Systems* vol . 6.

AUTHOR BIOGRAPHIES

B. EREN DERE is a computer engineer at Roketsan A.S., Ankara, Turkey and master's student at the Department of Computer Engineering, Middle East Technical University (METU), Ankara, Turkey. His current research interests are agent based modeling and simulation and model driven engineering. His email address is dereeren@yahoo.com.

B. KAAN GÖRÜR is a leader computer engineer at Roketsan A.S. in Ankara, Turkey. He received his Ph.D. degree in Department of Computer Engineering in Hacettepe University. His research interests are parallel and distributed simulation, agent based modeling and simulation, model driven engineering, and simulation visualization. His email address is bkaangorur@gmail.com.

HALİT OĞUZTÜZÜN is a professor at the Department of Computer Engineering, Middle East Technical University (METU), Ankara, Turkey. He got his Ph.D. in Computer Science from University of Iowa, Iowa City, IA in 1992. His current research interests are model-driven engineering and distributed simulation. His email address is oguztuzn@ceng.metu.edu.tr.