EmbeddedMontiArc: A Implementation Case Study

Haller, Heithoff, Sezer
RWTH Aachen

Abstract

Nowadays, modeling languages for cyper-physical systems play an important role in software engineering. A corresponding example is the automotive branch where safety has a high relevance. Apart from that, the magnitude and quantity of software projects rises constantly, as software development needs spread among scientific and technical disciplines. For this case, Domain Specific Languages (DSLs) are used in order to cope with these specialized tasks and environments, thus there exists a necessity for tools to define these DSLs comfortably. There are two kinds of problems: One of the problem is data-oriented (e.g. Amazon), and the other problem is decision-oriented (e.g. autonomous driving car) which is our use case. Moreover, implementation of the above-mentioned modeling language is still a problem to make them work properly. Component and Connector (C&C) approaches are used as a common method to describe the architecture of DSLs. Testing on cyber-physical systems is still necessary, as testing in real-life would cause higher costs. One difficulty which makes simulation difficult is also that physical laws must hold in such systems. MontiCar is such an environment or respectively a framework you can use to do agile and modeldriven software development in which the core language of this is EmbeddedMontiArc [Armin]. In order to cope with the rising complexity and recurring nature of DSLs, parser-generators or software language workbenches can be of great value.

This paper represents a case-study, evaluating the ease of use and re-usability

☆

of MontiCore for reactive systems. Our exact task and the topic of our seminar thesis is to analyse the components and behaviour of the Pacman game. Furthermore, it also belongs to our task that we write about our experiences among other things how long we need to learn EMA and which difficulties we face during our project. To that it also belongs that we shall write what was missing in EMA to fullfill our tasks and how long we needed to create a model. Moreover this seminar paper will be based on several research questions which are among other things: RQ1: Is EmbeddedMontiArc suitable for other systems? RQ2: Is it possible to integrate other simulators in a recent amount of work? RQ3: What kind of background knowledge is needed to model C&C in EMA? RQ4: What features are good and what are not suited?

Keywords:

---

1. Introduction

We are a group of three people and our seminar paper - as mentioned in the abstract - contains a case study about the EmbeddedMontiArc system. The people in this group have different preliminary knowledges which makes the writing of a case study somewhat more complicated on the one hand, and on the other hand it makes it more exciting as everyone brings up their own knowledge and idea. This seminar thesis will be based on the research questions which are mentioned in the abstract. This introductory chapter[1] will give you an idea of our approach and methods with regard to the research questions.

The first research question **RQ1** was about the question "Is EmbeddedMontiArc suitable for other systems?". In order to answer this question we thought about four items namely Objective, Theory, Method and Evaluation. Concerning Objective we can say that we should implement a model in other systems such that we can see whether EmbeddedMontiArc is suitable for other systems. Apart from that, we thought about the theory of the first research question. That is, it seems to be possible since other software (e.g. autopilot, SuperMario, PacMan etc.) runs on EmbeddedMontiArc. Our method for **RQ1** will be to implement some features for PacMan or SuperMario. With reference to the evaluation of **RQ1** we will answer a questionnaire which contains following items:

- Performance
- The effort of installing
- How good is the IDE integrated?
- Intuitiveness

The second research question **RQ2** was "Is it possible to integrate other simulators in a recent amount of work?". Here, we are geared to the same items

[1] Author: Sezer

(Objetive, Theory, Method and Evauation) as above and these items also hold for the following research questions. The objective of **RQ2** is to implement a PacMan Simulator/SuperMario Simulator. Its theory is the same like for the **RQ1** namely it should be possible as other programs are run by Embedded-MontiArc. The method of **RQ2** is whether people with different expertises are capable of implementing (Expert vs. Non-Expert). This also needs the question "how much time it is needed?" and "How many explanations are needed in order to be able to do the implementation?". Here, the evaluation is also a questionnaire as follows:

- Time to implement
- Help with implementation
- Are there any bugs?

The third research question **RQ3** deals with the question "What kind of background knowledge is needed to model C&C in EMA?". We subdivide this question into two subquestions. The first subquestion concerns with a simple model. The corresponding objective in the first subquestion is to implement a simple model for PacMan (e.g. PacMan runs away from the ghosts or PacMan runs along the wall). The theory is "What are components and for what are they used?". The method is more or less the same in **RQ2** ("Are people with different expertises are capable of implementing?" and "How much time and how many explanations are needed?") and "Do the people need a workshop?". The questionnaire of the evaluation is as follows:

- Time to implement
- Help with implementation
- Which preliminary knowledges helped us?
- Quality of the components

The second subquestion of **RQ3** which refers to a more advanced model has the objective to implement an advanced model for PacMan with a good controller. The theory of the second subquestion is also "What are components and for what are they used?". The questionnaire for the second subquestion is as follows:

- Time to implement
- Help with implementation
- Which preliminary knowledges helped us?
- Quality of the components

Concerning the fourth research question **RQ4** we can say that the objective is the idea of improvement. There is no theory in **RQ4**. The coresponding method is to make notes during implementation and questionnaire in the method part. The questionnaire is composed as follows:

- Intuitiveness
- Completeness of features
- Which bugs have been occurred?
- Which features have been good?
- Which features have been bad?
- Which features have been missing and how were they translated?

2. Context

The following section consists of three parts. The first one is a brief introduction to C&C models. The tools used for this study follow up second. Lastly, the used case study method is presented.

2.1. C & C models

In the following a short introduction in Connector and Component (C&C) model based software development is given. C&C modeling divides a task into Components and Connectors.

A *Component* represents a computation. It has predefined inputs and outputs, where the output data is obtained by some kind of mathematical transformation of the input data. A *Connector* represents interaction mechanisms by connecting outputs with inputs. By making this division, the paradigm ensures modularity and therefore reusability. It can be used for modelling software with high demands for testing and verification such as software for self-driving vehicles [1][2]. Another benefit is that a graphical representation is always possible and more efficiently obtainable compared to other text based development, especially non model driven development. The structure of C&C models also benefits code generation techniques in order to transform models into source code for various target systems. Well established examples of C&C modeling and development are SysML[3], AADL[4], Simulink[5] and Labview[6]. The latter two are used in the automotive domain to model behaviour of Electronic Control Units (ECUs) and test their functionality.

2.2. MontiCore and EmbeddedMontiArc

MontiCore[ref], MontiCAR[ref] and EmbeddedMontiArc[ref] are tools developed by the Chair of Software Engineering of RWTH Aachen University[7]. *MontiCore* is a language workbench intended for agile and model-driven software development. Its primary objective is to enable efficient development of Domain Specific Languages (DSLs) which enhance the development process for Domain Experts. *MontiCAR* is a composition of such DSLs, used as an language set for Cyber-Physical Systems [8]. Figure 1 shows the DSLs which are part of MontiCar and their respective connections. The components directly used in this studies implementation are EmbeddedMontiArc, EmbeddedMontiArcMath and Stream. *EmbeddedMontiArc* represents the core language of MontiCar. It implements a C&C DSL which can be used to write C&C models, verify, test

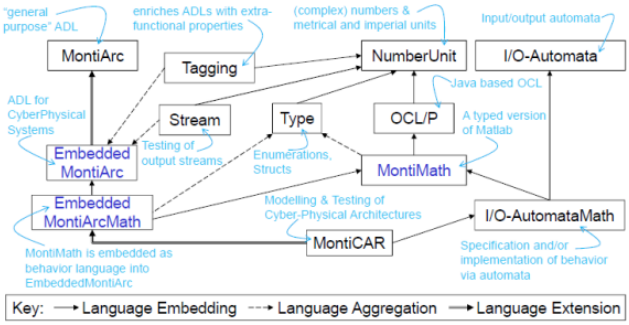

Figure 1: Composition of MontiCAR language family[8]

and deploy them to another architecture. Due to its modularity different simulators and Stream tests can be integrated. See the chapter Modelling for more information. Figure 2 depicts a usage of the EmbeddedMontiArc DSL.

```
component PacManControllerSimple {
  ports
      in Z(0m: 342cm) ghostX[4],
      in Z(0m: 426cm) ghostY[4],
      in Z(0 : 1 : 3) ghostDirection[4],
      in B ghostEatable[4],
      in B ghostEaten[4],
      in Z(0m: 342cm) pacManX,
      in Z(0m: 426cm) pacManY,
      in B pacManEaten,
      in Z(1:oo) pacManLives,
      in Z(0:oo) pacManScore,
      in Z^{22,19} map,

      out Z(0 : 1 : 3) newPacManDirection;

  instance Fallback fallback;

  connect fallback.out1 -> newPacManDirection;
}
```

Figure 2: Example Component with Connectors

EmbeddedMontiArcMath is a DSL for implementing mathematical expressions, thus used for transforming the input values of a Component into its output values. It is also able to declare other variables than the defined inputs and logical structures like if-statements and loops. Example usage of EmbeddedMontiArcMath is shown in figure 3 .

The *Stream* DSL allows to implement test cases by defining the expected output values for a given input. Multiple values can be tested in one Stream test, as shown in figure 4 which shows an example Stream for a sum function.

```
component NearestGhost {
    ports
        in Z(0cm: 342cm) ghostX[4],
        in Z(0cm: 426cm) ghostY[4],
        in Z(0cm: 342cm) pacManX,
        in Z(0cm: 426cm) pacManY,

        out Z(0:1:3) nearestIndex;

    implementation Math {
        Q min = 3430;
        Z index = 0;

        for i = 0:4
            Q distX = ghostX(i) - pacManX;
            Q distY = ghostY(i) - pacManY;
            if (distX < 0)
                distX = distX * (-1);
            end
            if (distY < 0)
                distY = distY * (-1);
            end
            Z dist_sqr = distX + distY;
            Q dist = sqrt(dist_sqr);
            if(dist < min)
                min = dist;
                index = i;
            end
        end

        nearestIndex = index;
    }
}
```

Figure 3: Example EmbeddedMontiArcMath implementation

```
stream Sum for Sum {
  t1: 1 tick 2 tick 3;
  t2: -1 tick 0 tick 10;
  result: 0.0 +/- 0.01 tick 2.0 +/- 0.01 tick 13.0 +/- 0.01;
}
```

Figure 4: Example Stream implementation]

2.3. Performing a case study in Software Engineering

This study roughly follows the guidelines stated by Runeson and Hst [9] by presenting the objective, the specific case, method and aquiring both quantitative and qualitative data. Quantitative data is acquired by asking a set of predefined questioned and answering them on a scale from 1 to 10. The qualitative data is obtained via requiring subjects to formalize how they gave the quantitative rating. The quantitative data is analysed by calculating the mean of each question, and the quantitative by summarizing the subject's writings.

2.4. Subject knowledge prior to study

In the following a short summary of the three subject's state of knowledge is given:

... This section is going to be finished once the implementation step is over

Knowledge level	General implementation	Model Based SE	C&C Models
Subject 1	..	..	..
Subject 2	..	..	..
Subject 3	..	..	..

Figure 5: Subject's prior knowledge overview

...

3. Approach

3.1. Stream Testing

To address **RQ1** and **RQ3** we assigned two groups the task to model a Controller for PacMan and SuperMario respectively and interview the results afterwards. In the future we will refer to the groups as *group A* and *group B*. Group A consists of one subject who is familiar with EmbeddedMontiArc and group B consists of two subjects who have no experience with EmbeddedMontiArc. These groups were selected random among the students of a computer science seminar. EmbeddedMontiArc comes along with stream tests in order to check a component against a condition as stated in the previous chapter. We can use those tests to define the conditions the controllers need to fulfill. Those conditions are taken from use cases. For PacMan the most general acceptance test would be to never let the PacMan die. Due to the fact that stream tests cannot be defined unlimited and that this test might be hard to implement we defined the following deterministic tests for PacMan and SuperMario. We first state those for PacMan:

3.1.1. PacMan

- Given the scenario in fig. 6, the PacMan must be controlled to the left (in order to not die)
- Given the scenario in fig. 7, the PacMan should be controlled toward the blue ghost, sending him away from his current position.

- Given the scenario in fig. 8 (Zwei Wege sind gleichwertig, einer hat einen Cookie drin) the PacMan should navigate to the cookie.

Figure 6: PacMan has to go left in order to live

Figure 7: Ghosts are eatable

Those scenarios can be tested easily within a few ticks via stream testing.

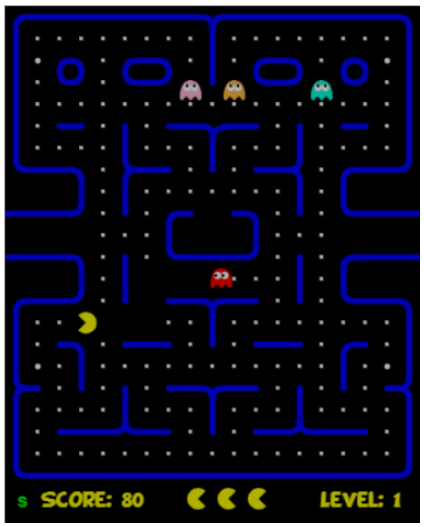

Figure 8: Pacman can choose between going left or down

3.1.2. SuperMario

In the following the scenarios a SuperMario model has to master are listed:

- Given the scenario in fig. 9, Mario has to build up speed in order to jump over the obstacle.
- Given the scenario in fig. 10, Mario has to jump in order to get coins, mushrooms or flowers.
- Given the scenario in fig. 11, Mario has to eat the mushroom in order to grow.
- Given the scenario in fig. 12, Mario has to jump on the evil mushrooms or evade them.

The two groups were then assigned to use these tests as a guideline to develop their controllers.

3.2. Preparations

The Code of the PacMan emulator (Link?) and SuperMario emulator [10] we used was available in Html5 and JavaScript. C&C-Components in Embed-

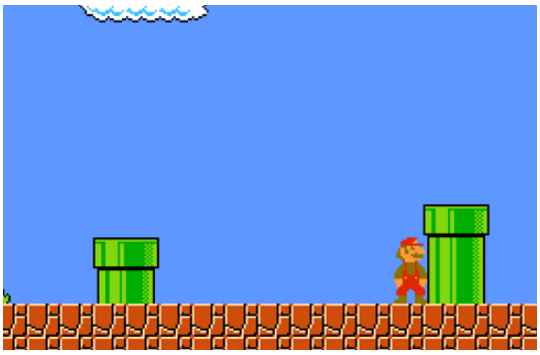

Figure 9: Mario has to go left in order jump over the obstacle

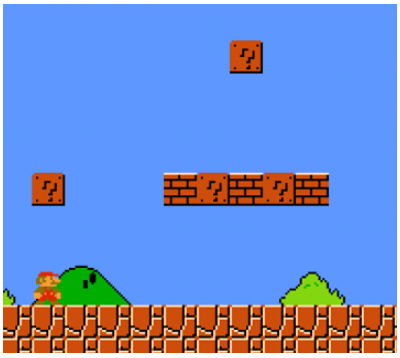

Figure 10: Mario with loot boxes over him

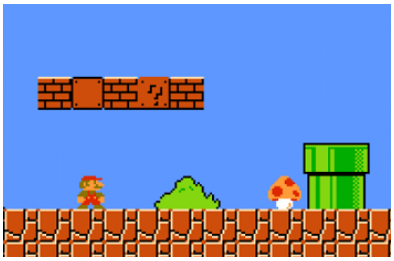

Figure 11: Mario with a mushroom

dedMontiArc can be translated to C++ code and then to a webassembly which uses JavaScript (see [**?**]). This JavaScript file can be given inputs according to the component and calculates the outputs on execution. To combine these two files, there is an additional interface needed to extract the information for

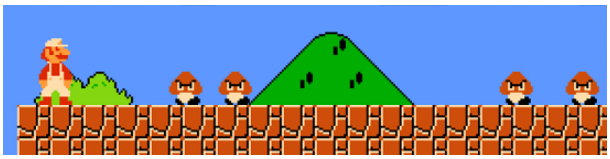

Figure 12: Mario with enemies

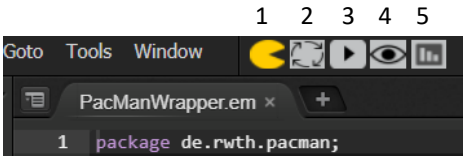

Figure 13: Main options for the PacMan project in the ide

the inputs out of the emulator and then give the calculated outputs into the emulator. For the purpose of implementing the controllers the subjects were assigned to use the EmbeddedMontiArcStudio. EmbeddedMontiArcStudioV1.6.2 did neither support a simulator of PacMan nor of a simulator SuperMario. So an additional step to awnser RQ2 *Is it possible to integrate other simulators in a recent amount of work* it for the groups to integrate the simulators into the EmbeddedMontiArcStudio. In order to be able to do so, group A is instructed by an expert (Jean-Marc) which files need modification and what to add. After that group A instructed group B the same way.

4. Implementation

In this chapter the implementation is described. First, the necessary steps for integrating a new Simulator into the IDE are shown. In the second part the modelling of the controllers for Pacman and Supermario are discussed.

4.1. IDE integration

To integrate a simulator into the EmbeddedMontiArcStudio several steps were necessary. In figure. 4.1 you can see the top view of the EmbeddedMontiArc's ide. The five added features here are as follows:

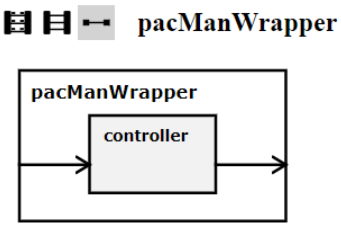

Figure 14: Visualization of the PacMan wrapper

1. This opens a new tab where you can play a normal game of PacMan
2. Generate the WebAssembly of the main component
3. This opens a new tab in which the simulation of the component takes place
4. Generates the visualization of the main component and shows it in a new tab
5. Generates the reporting of all components and shows it in a new tab

The features needed to be implemented properly in different places in order to work along the logic of the ide. Each one calls a batch script which again runs the jar for the demanded task for the specific files. In addition, for feature 1 and 2 extra plugins were required which got implemented by group A (expert) and can be reused for SuperMario. A full list of the files edited can be found in the attachment.

4.2. Modelling

This chapter contains the modelling of Pacman and Supermario respectively.

4.3. Interface

Listing 1: Interface of the Pacman Wrapper

```
ports
        in Z(0cm: 180cm) ghostX[4],
```

```
        in Z(0cm: 210cm) ghostY[4],
        in Z(0 : 1 : 3) ghostDirection[4],
        in B ghostEatable[4],
        in B ghostEaten[4],
        in Z(0cm: 180cm) pacManX,
        in Z(0cm: 210cm) pacManY,
        in B pacManEaten,
        in Z(0:oo) pacManLives,
        in Z(0:oo) pacManScore,
        in Z^{22,19} map,
        out Z(0 : 1 : 3) newPacManDirection;
```

The project's main component is PacManWrapper. The main task of the wrapper is to provide a shared interface. Listing 1 shows the input and output ports. As for the inputs, the ghosts' and the PacMan's position are given, the direction the ghosts are facing, information about the ghosts' vulnerability, as well as the current map. The only output port is the new direction the PacMan should walk.

The wrapper also holds the current controller. This way the controller is easily exchangeable without changing any of the code needed for the ide. All input ports of the wrapper are connected to the corresponding ports of the controller and the output port of the controller is also connected to the output port of the wrapper.

To connect the web assembly of the main component with the PacMan emulator a new JavaScript file was created. Its main functionalities is to extract the needed informations out of the emulator, pass it to the component, execute it and then give the output back to the emulator. In order to be able to extract needed information out of the emulator some modifications were needed. In its original state the emulator did not offer access to the current game object, thus the PACMAN class was extended by these functions. Due to the fact PacMan is a playable game, its input is given as a key-press-event in JavaScript. So the

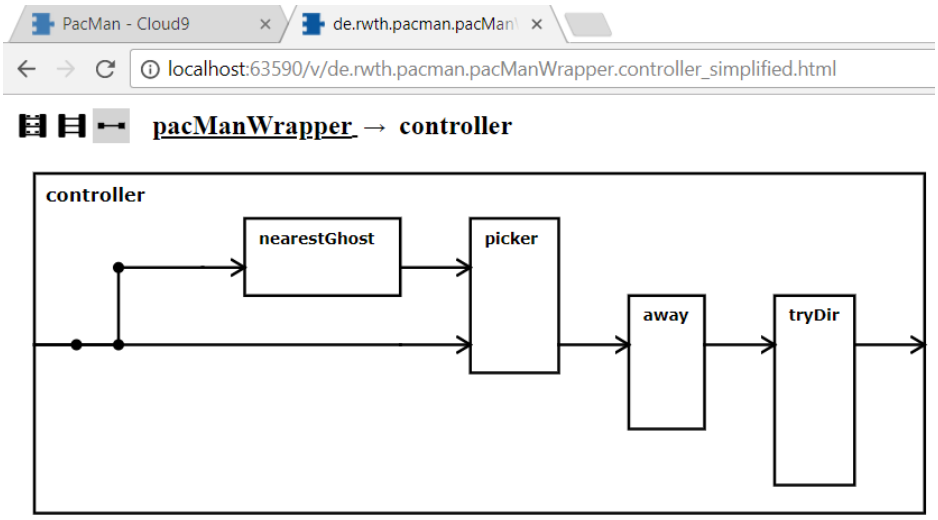

Figure 15: Visualization of the PacMan controller (simple)

output of the web assembly, which is a number from 0 to 3, is mapped to a corresponding key-press-event which then gets triggered. The emulator is running with 30 frames per second, which also leads to 30 iterations of the game per second. Because the emulator is running asynchronously the component is executed at a double of that rate in order to track every position change.

4.4. C&C modeling - PacMan

In fig. 4.4 the design of the simple controller is shown. It has four subcomponents:

- nearestGhost: Gets the x - and y - position of every ghost and the x - and y - position of the PacMan. It then iterates over all ghosts and calculates the nearest ghost and gives back its index.
- picker: Gets all ghost informations as input as well as an index and gives back the ghost information of the ghost at this index.
- away: Gets one ghost's informations as well as PacMan's and calculates a new direction for the PacMan facing away from the ghost. The output is one of the four possible directions mapped from the numbers 0 to 3.

- tryDir: Gets as input the position of PacMan, the current map as well as a direction the PacMan should try to walk. If there is no wall blocking the way the initial direction is outputted. On the other hand, if there is a wall blocking the way it tries to walk orthogonally left or up. If it fails it will walk right or down respectively.

The controller connects the subcomponents in the shown order: It calculates the nearest ghost, passes its index to the picker which then again passes the corresponding ghost to the *away* component. This calculates the direction facing away from said ghost and the *tryDir* component then avoids running into walls. This leads to a controller that runs away from the ghosts with some success but it is only determined by the nearest ghost and has no other goals. Due to the fact that *tryDir* always trys to walk to the left (or top) first, this can lead to some stuttering as soon as the PacMan walked enough to the right that there is again space to the left.

5. Evaluation

In the section the sampled data is analyzed. First the results of the quantitative analysis are presented, followed by statements made by the subjects and a final summary.

5.1. Quantitative Analysis

To be added after implementation completion...

5.2. Qualitative Analysis

5.2.1. Subject writeups

In this section I (Sezer) will present some evaluations concerning the facts and problems of the simulator and IDE. Apart from that, I will also clarify my problems I faced during our approach.
First of all, I had to understand what "C&C" means and I also had to clarify what components are in our context. Furthermore, I had to get accustomed to

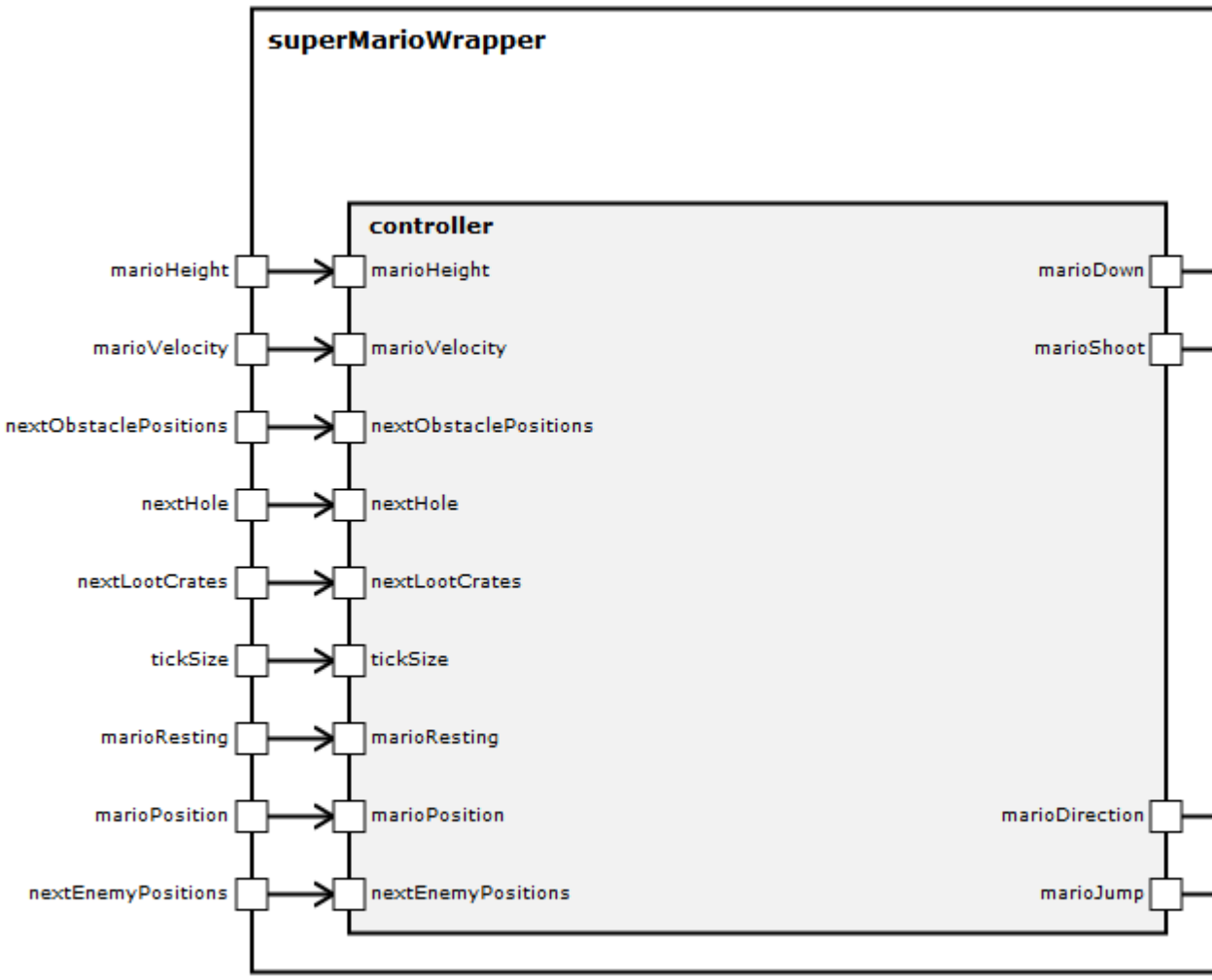

Figure 16: Visualisation of the Supermario wrapper model

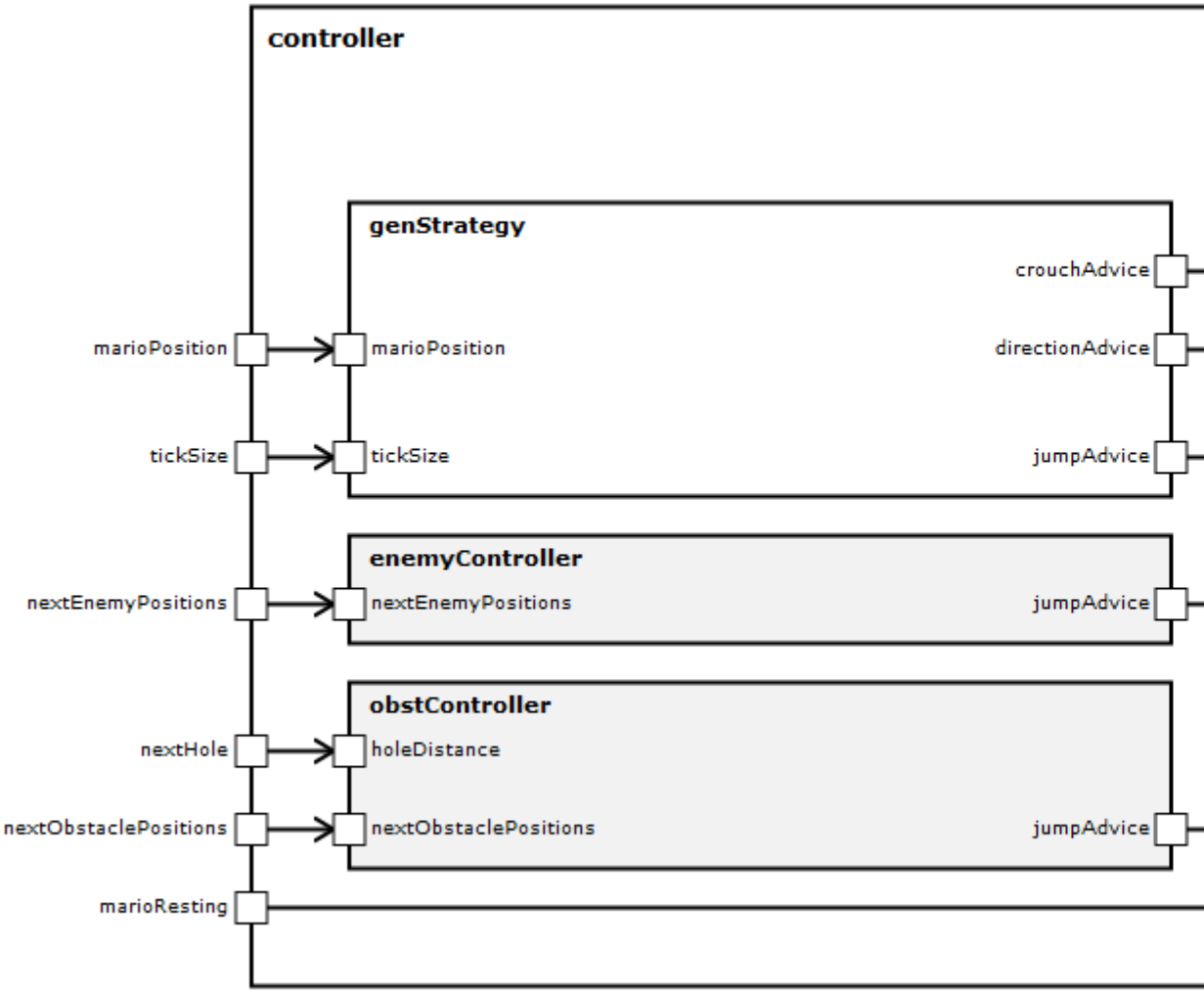

Figure 17: Visualisation of the Supermario controller model

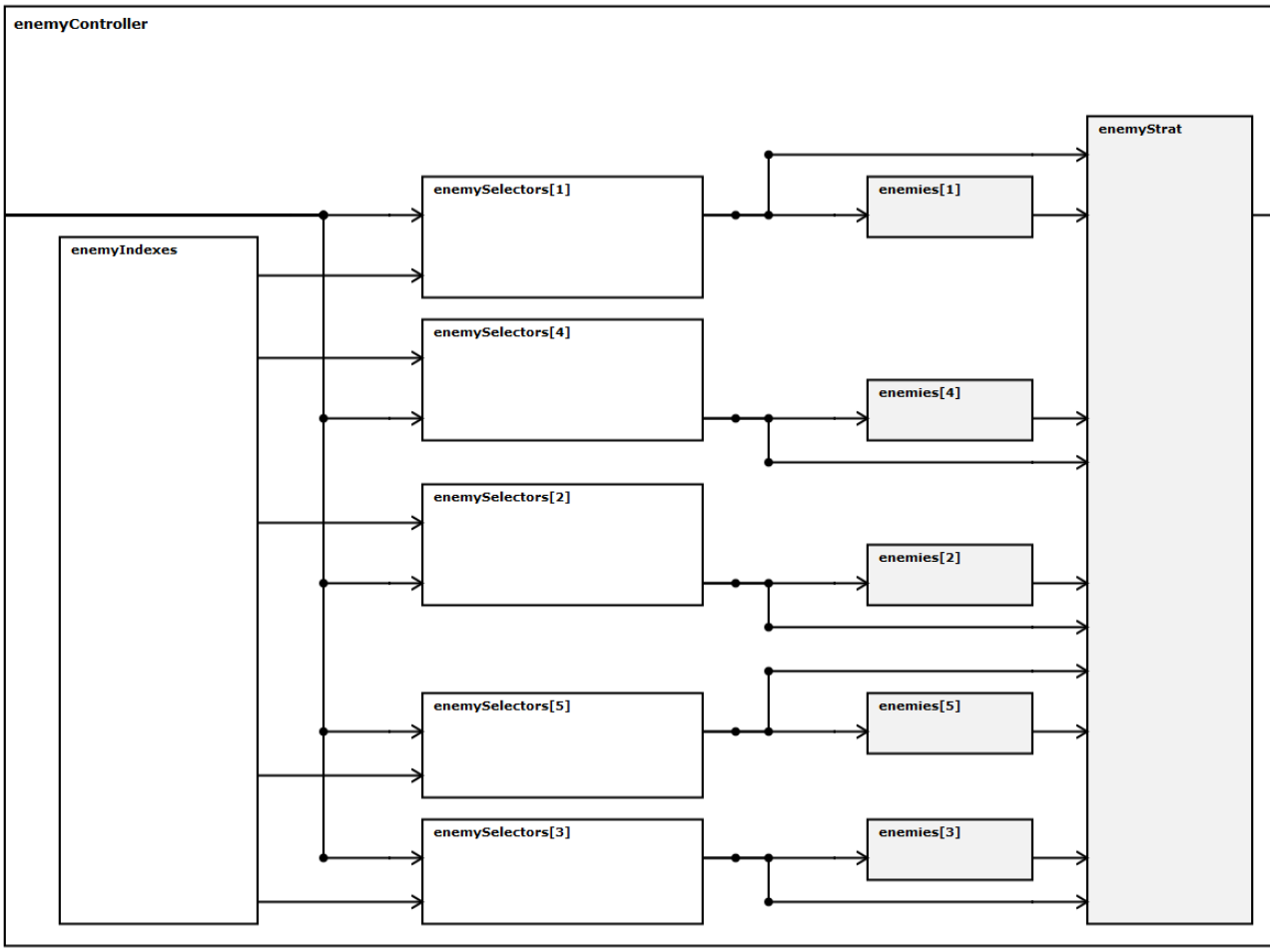

Figure 18: Visualisation of the Supermario enemy controller model

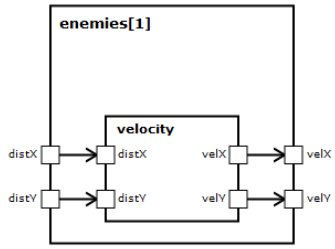

Figure 19: Visualisation of the Supermario enemy model

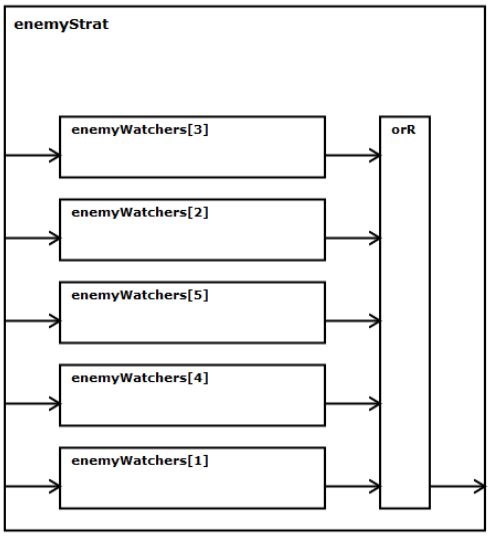

Figure 20: Visualisation of the Supermario enemy strategy model

the syntax of the IDE which is quite easy to handle. Moreover, the IDE offers a good overview on the used components and how components are connected with each other. Personally, I also had to get to know what a "wrapper" and a "controller" are. For someone like me who is relocated in web development, it was difficult to get into the project. Besides there are other more intuitive datatypes in EmbeddedMontiArc Studio. These are for example **B** (Boolean), **Z** (Integer), **Q** (Rational number) and **N** (Natural number). It is also possible to assign a domain or an interval to the above-mentioned datatypes which is not always possible with other programming languages. You can also "define" units which makes it easier to comprehend your datatypes and which is not always usual for other programming languages. Apart from this, we can also create and handle with matrices in an easier way. The creation of matrices in the EmbeddedMontiArc studio is more intuitive. In my opinion, Michael's and Armin's video are good for beginners since the interaction with the components and how to program them are explained in a good manner. Besides, the relation between the single "MontiArc" programming languages are described in the tutorial videos. What I liked in our project is that we could use techniques

from the software engineering domain (e.g. representation of the components) but nevertheless we faced some problems which I will mention below. What I also liked was that it is shown in the IDE whether the used packages are correct or not. What I didn't like was that there is no possibility of debugging as it is possible in other IDEs. It is very difficult to find a semantical error and the only stuff that was available is the start console of the IDE where errors are shown. But there are no methods for debugging like variable watcher or breakpoints which made the debugging for us very difficult. Honestly, I had problems to get an idea how to start with the implementation of the PacMan-game. Therefore, I had a look on my partners' code examples in order to get an inspiration.

5.2.2. Summary of writeups

6. Conclusion

This section is going to be written in the near future

References

[1] V. Bertram, S. Maoz, J. O. Ringert, B. Rumpe, M. von Wenckstern, Component and connector views in practice: An experience report, ACM/IEEE International Conference on Model Driven Engineering Languages and Systems 20.

[2] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, Michael von Wenckstern, Simulation framework for executing component and connector models of self-driving vehicles, Proceedings of MODELS 2017. Workshop EXE, Austin, CEUR 2019, Sept. 2017.

[3] OMG, Sysml.
URL http://www.omgsysml.org

[4] SAE, Architecture analysis and design language.
URL http://www.aadl.info/

[5] Mathworks, Simulink.
URL https://de.mathworks.com/products/simulink.html

[6] N. Instruments, Labview.
URL http://www.ni.com/de-de/shop/labview.html

[7] C. of Software Engineering RWTH Aachen University, Homepage of the chair of software engineering at rwth aachen university.
URL http://se-rwth.de

[8] A. Mokhtarian, Monticar: 3d modeling using embeddedmontiarcmath (2018).

[9] P. Runeson, M. Hst, Guidelines for conducting and reporting case study research in software engineering.

[10] J. Goldberg, Fullscreenmario, html5 browser game.
URL http://www.joshuakgoldberg.com/FullScreenMario/Source/