EmbeddedMontiArc: A Implementation Case Study

Haller, Heithoff, Sezer

RWTH Aachen

Abstract

(Abstract by Philipp Haller) The magnitude and quantity of software projects rises constantly, as software development needs spread among scientific and technical disciplines. 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. 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 EmbeddedMontiArc for reactive systems by developing models for the games Pacman and Supermario. Games are highly reactive systems were entities controlled by the player react to a changing environment and try to reach goals. From the models presented it is concluded that EmbeddedMontiArc ... FINISH THIS SENTENCE AFTER CONCLUSION

Keywords:

☆

1. Introduction

Abstract-Sezer 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?

—-

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.

[1]Author: Sezer

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 (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 EmbeddedMontiArc. 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 (by Philipp Haller)

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 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.

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 .

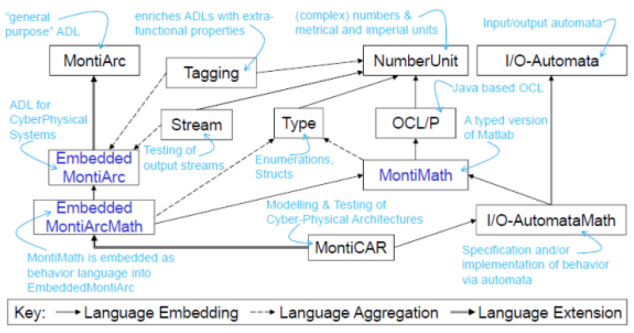

"general purpose" ADL
enriches ADLs with extra-functional properties
(complex) numbers & metrical and imperial units
Input/output automata
MontiArc
Tagging
NumberUnit
I/O-Automata
Java based OCL
ADL for CyberPhysical Systems
Stream
Type
OCL/P
A typed version of Matlab
Embedded MontiArc
Testing of output streams
Enumerations, Structs
MontiMath
Embedded MontiArcMath
Modelling & Testing of Cyber-Physical Architectures
I/O-AutomataMath
MontiCAR
MontiMath is embedded as behavior language into EmbeddedMontiArc
Specification and/or implementation of behavior via automata
Key: Language Embedding Language Aggregation Language Extension

Figure 1: Composition of MontiCAR language family[8]

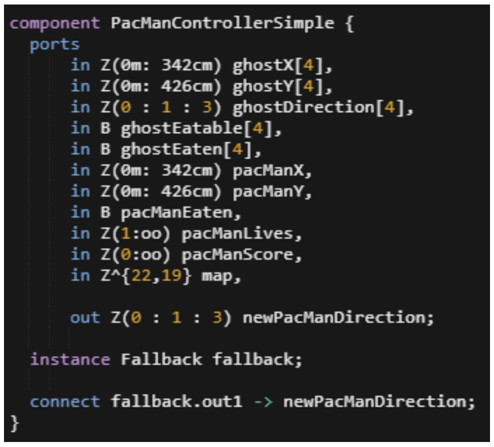
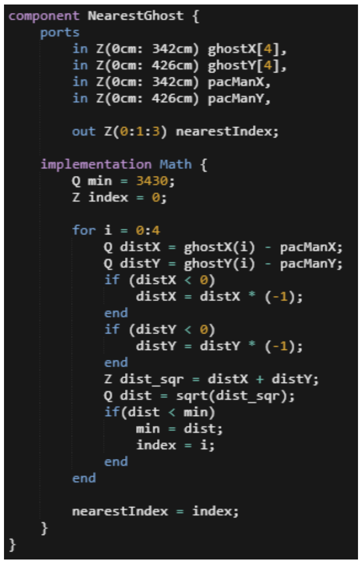

```
component PacManControllerSimple {
  ports
      in Z(0m: 342cm) ghostX[4],
      in Z(0m: 426cm) ghostY[4],
      in Z(0 : 1 : 3) ghostDirection[4],
      in B ghostEatable[4],
      in B ghostEaten[4],
      in Z(0m: 342cm) pacManX,
      in Z(0m: 426cm) pacManY,
      in B pacManEaten,
      in 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

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 test for a sum function.

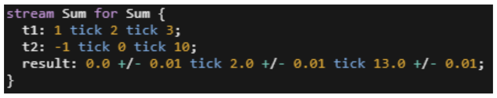
2.3. Performing a case study in Software Engineering

This study roughly follows the guidelines stated by Runeson and Hoest [9] by presenting the objective, the specific case, method and acquiring 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 analyzed by calculating the mean of each question, and the quantitative by summarizing the subject's writings.



```
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]

3. Approach

3.1. Stream Testing

To address **RQ1** and **RQ3** two groups were assigned the task to model a Controller for PacMan and SuperMario respectively and interview the results afterwards. In the future the groups will be referred to 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 scenarios. 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 fulfill the following deterministic tests for PacMan and SuperMario were defined.

3.1.1. PacMan

The tests are taken from use case scenarios as stated before. In this section the process of deriving the stream test from a scenario is presented once and then a few conditions are framed.

Deriving a Stream Test

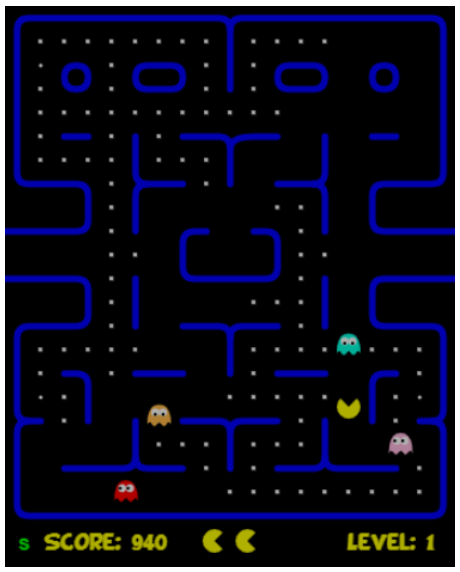
In fig. 5 a scenario is shown where the only option for PacMan is to flee to the left in order to not collide with the pink and blue ghost. The values of the ghosts and PacMan are partially listed in listing 1. Together with the other values this concludes to the stream test shown below 2.

Listing 1: Values for the stream test

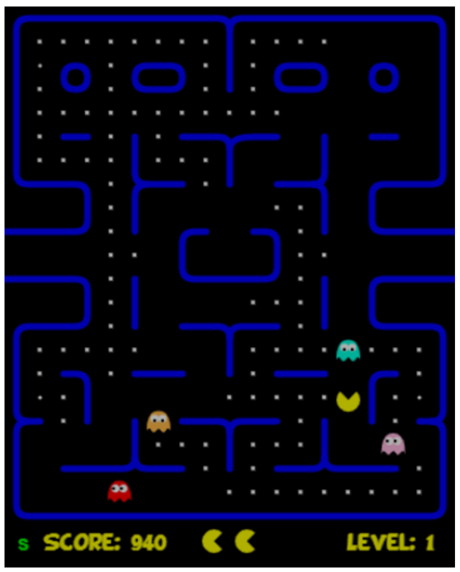
```
(a)
  PacMan: (15m, 17.2m)
  Pink Ghost: (17m, 19m)
  BlueGhost: (15m, 14.8m)
  newDir: 0
(b)
  PacMan: (15m, 17m)
  Pink Ghost: (16.8m, 19m)
  BlueGhost: (15m, 15m)
  newDir: 0
(c)
  PacMan: (14.8m, 17m)
  Pink Ghost: (16.6m, 19m)
  BlueGhost: (15m, 15.2m)
  newDir: 2
(d)
  PacMan: (14.6m, 17m)
  Pink Ghost: (16.4m, 19m)
  BlueGhost: (15m, 15.4m)
  newDir: 2
```

Listing 2: Stream test for the scenario above

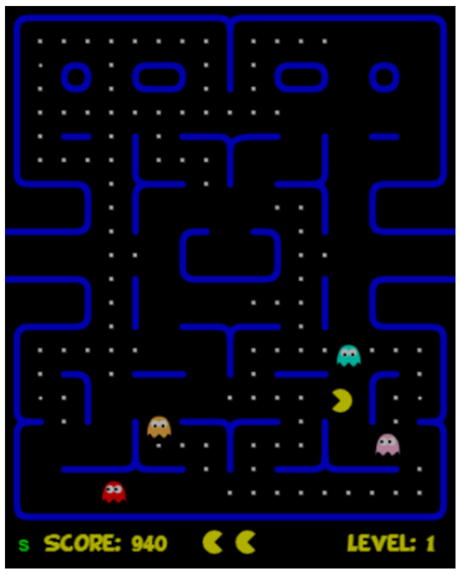
```
package de.rwth.pacman;
stream Test1 for PacManWrapper {
  ghostX: [5.4m,15m,17m,7m] tick [5.2m,15m, ...
  ghostY: [21m,14.8m,19m,17.2m] tick [21m,15m, ...
  ghostDirection: [2,1,2,1] tick [2,1,2,1] tick ...
  ghostEtable: [false, false, false, false] tick ...
  ghostEaten: [false, false, false, false] tick ...
  pacManX: 15m tick 15m tick 14.8m tick 14.6m;
  pacManY: 17.2m tick 17m tick 17m tick 17m;
  pacManEaten: false tick false tick false tick false;
  pacManLives: 3 tick 3 tick 3 tick 3;
```



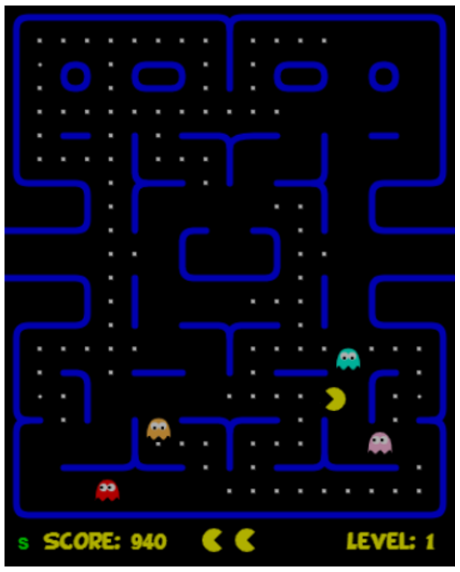
(a)



(b)



(c)



(d)

Figure 5: PacMan has to move left to avoid colliding with the ghosts

```
pacManScore: 0 tick 0 tick 0 tick 0;
map: [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0; ...
newPacManDirection: 0 tick 0 tick 2 tick 2;
}
```

Some other tests

- If PacMan is located at an intersection and ghosts are coming from two sides, PacMan should walk to a safe path.
- If PacMan is located at an intersection and ghosts are at the top path and are all eatable, PacMan should walk this path.
- If PacMan is located at an intersection

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

3.1.2. Supermario (by Philipp Haller)

The goal for the Supermario model is to solve a level successfully. The first level was chosen since it provides a diverse environment with different enemy types and obstacles, while not being too skill intensive to solve. Prior to modeling some assumptions were made to fulfill time and complexity constraints. Only a fixed number of enemies and obstacles in the path of the player are considered in order to ensure a static input size. For this number, five has proved to be sufficient for the first level and the implemented strategy. There are rarely more than 3 enemies in scene. For the same reason only the next hole in the ground is considered. In order to develop the model, different situations were assessed and according tests derived. Both the scenarios which a Supermario model has to master and the derived tests are listed below.

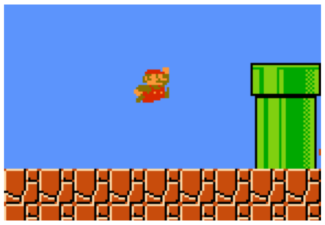


Figure 6: Mario has to jump and move right to overcome the obstacle

Figure 6 depicts the player next to an obstacle. In order to jump over it he has to move right and jump at the same time. He needs to keep jumping until he is higher than the obstacle.

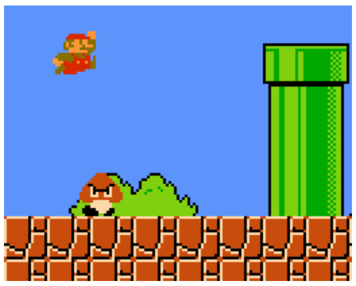
Figure 7 shows two situations. In the first one, mario jumps to evade an enemy. The second depicts him landing on top of enemies to kill them.

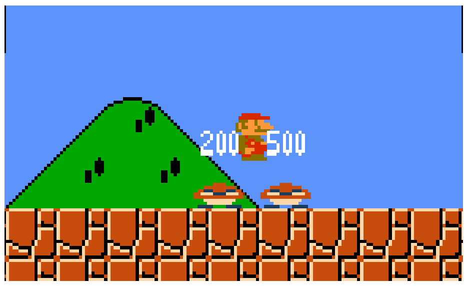
In Figure 8 the player is seen standing next to holes in the ground. In the first picture he is on the ground level, in the second he is standing on an obstacle.

The stream tests derived from the scenarios are introduced in the following.

Listing 3: Enemy watcher stream test

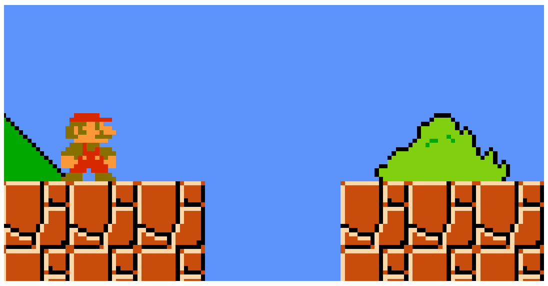
```
package de.rwth.supermario.haller.environment;
```


(a) Mario evades a enemy by jumping



(b) Mario defeats enemies by landing on them

Figure 7: Mario has to jump over/to enemies


(a) Mario and a hole in the ground

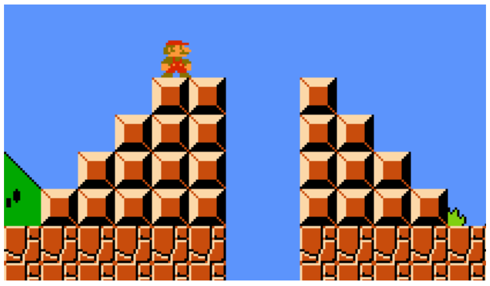

(b) Mario and a hole with obstacles

Figure 8: Mario has to jump over a hole

```
stream Env_EnemyWatcher_Evade for EnemyWatcher {
    EnemyDistX:          200  tick   100    tick     75;
    EnemyDistY:            0  tick     0    tick      0;
    EnemyVelocityX:      -10  tick   -10    tick    -10;
    EnemyVelocityY:        0  tick     0    tick      0;

    movesTowardsPlayer:    1  tick     1    tick      1;
    inJumpRange:           0  tick     0    tick      1;
    }
```

If a enemy gets closer than 80 pixels (two blocks) and is on the same height as the player, the player has to jump in order to evade the enemy (listing 3). The units for the EnemyDistX and EnemyDistY values are pixels, while the velocities are given in pixels per time frame. The output values are of type boolean.

Listing 4: Enemy watcher stream test

```
package de.rwth.supermario.haller.environment;

stream Env_EnemyWatcher_FromAbove for EnemyWatcher {

    EnemyDistX:          200  tick   100    tick      5;
    EnemyDistY:          128  tick   128    tick     32;
    EnemyVelocityX:      -10  tick   -10    tick    -10;
    EnemyVelocityY:        0  tick     0    tick      0;

    movesTowardsPlayer:    1  tick     1    tick      1;
    inJumpRange:           0  tick     0    tick      0;

    }
```

The stream in listing 4 covers the case when the player is above enemies and shall drop on them while he is above.

Listing 5: Enemy watcher stream test

```
package de.rwth.supermario.haller.environment;

stream Env_EnemyWatcher_FromAbove for EnemyWatcher {

    EnemyDistX:          -1  tick;
    EnemyDistY:          -1  tick;
    EnemyVelocityX:       0  tick;
    EnemyVelocityY:       0  tick;

    movesTowardsPlayer:   0  tick;
    inJumpRange:          0  tick;

    }
```

If there is no enemy near the player, the enemy watcher object shall give no jump advice (listing 5).

Listing 6: Obstacle watcher stream test

```
package de.rwth.supermario.haller.environment;
stream Env_ObstacleWatcher for ObstacleWatcher {
  ObstacleDistX: 200 tick 100 tick 75 tick 50 tick 25 tick  0;
  ObstacleDistX:   0 tick   0 tick  0 tick 25 tick 50 tick 75;

  inJumpRange:     0 tick   0 tick  1 tick  1 tick  1 tick  0;
    }
```

If a obstacle is in front of the player, he shall jump until he has passed it(listing 6). The distances are given in pixels, and the obstacle in this text is of 70px height.

Listing 7: Hole watcher stream test

```
package de.rwth.supermario.haller.environment;
stream Env_ObstacleWatcher for ObstacleWatcher {
    holeDistance: 200 tick 100 tick 10 tick 0 tick 1200;

    inJumpRange:    0 tick   0 tick  1 tick     1 tick    0;
    }
```

In listing 7 the stream test for jumping over holes is given. In this case, the player shall start jumping close to the hole and only stop once he is over.

3.1.3. Supermario (by Mustafa Sezer)

- 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.

3.2. Preparations (by Haller and Heithoff)

The Code of the PacMan emulator [10] and SuperMario emulator [11] we used was available in Html5 and JavaScript. C&C-Components in EmbeddedMontiArc can be translated to C++ code and then to a web assembly (using Emscripten [12]) 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 the inputs out of the emulator and then give the

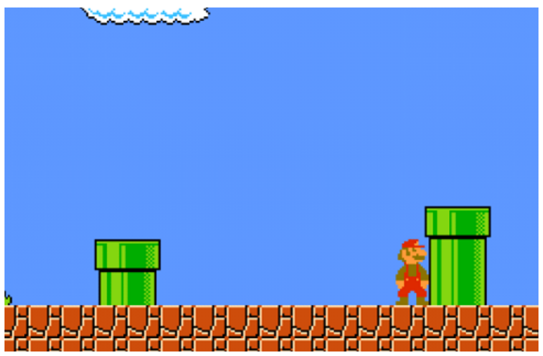


Figure 9: Mario has to go left in order jump over the obstacle

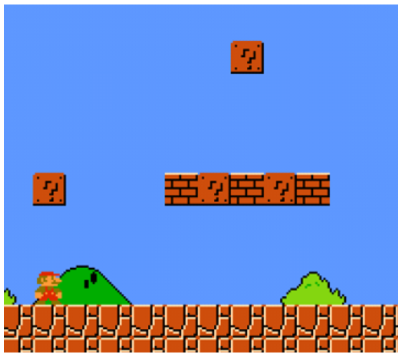


Figure 10: Mario with loot boxes over him

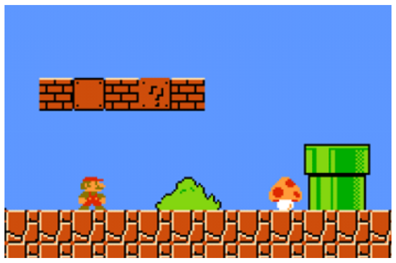


Figure 11: Mario with a mushroom

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 answer 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)

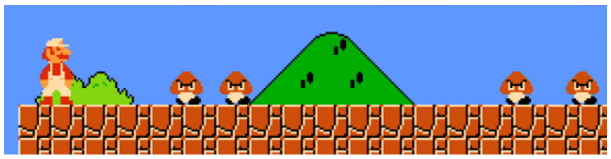


Figure 12: Mario with enemies

which files need modification and what to add. After that group A instructed group B the same way.

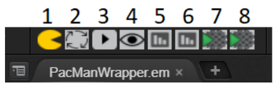


Figure 13: Main options for the PacMan project in the ide

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 modeling of the controllers for Pacman and Supermario are discussed.

4.1. IDE integration (Introduction by Philipp Haller)

As the participants of the use-case study were divided into two groups, the first group dealt with the IDE integration of the Pacman simulator after being instructed by an EMA professional. After successful integration this first group wrote a step-by-step instruction list. The second group used this list to integrate the Supermario simulator into the IDE. Details for both are given in the following.

4.1.1. Pacman (by Malte Heithoff)

To integrate a simulator into the EmbeddedMontiArcStudio several steps were necessary. In figure. 4.1.1 you can see the top view of the EmbeddedMontiArc's ide. The five added features here are as follows:

1. Open a new tab where you can play a normal game of PacMan
2. Generate the WebAssembly of the main component
3. Open 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
6. Generates the reporting of all components with stream test results and shows it in a new tab
7. Run all tests in the repository and show their results
8. Run a single test and show its result

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.

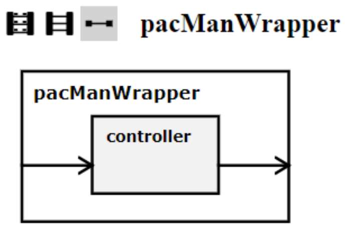


Figure 14: Visualization of the PacMan wrapper

4.1.2. Supermario (by Philipp Haller)

..maybe move this to context? *Execution*

The presented models are executed in

4.2. Modeling

This chapter introduces the models of Pacman and Supermario respectively. The models should always follow certain rules defined in the EmbeddedMontiArc documentation (see [13]). The math implementation of all atomic components should be short and have a short runtime. This way not only the clarity of the code is enhanced but also the runtime of the components is fixed. C&C models should, at some point, be runnable on microchips and due to the fact that those models are designed for real-time systems the runtime has to be fix. To achieve this a lot of functionality can be extracted into subcomponents. In general, loops should be avoided and split up into subcomponents if possible. Because while loops are not ensured to terminate, those should never be used.

4.3. PacMan (by Heithoff)

In the following the model for the PacMan controller is presented. The goal is to collect as many biscuits and coins as possible and to avoid the ghosts. After introducing the interface which is used here two controllers for PacMan are shown. There is a simple controller which was used in the early stages of the IDE integration to test everything and then a more complex controller that can actually survive a few levels.

4.3.1. Interface

Listing 8: 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 8 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 web assembly, 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 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.

4.3.2. C&C modeling - PacMan (simple)

In fig. 4.3.2 the design of a simple controller is shown. It has four subcomponents:

- nearestGhost: Is given 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: Is given all ghost informations as input as well as an index and gives back the ghost information of the ghost at this index.
- away: Is given 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.

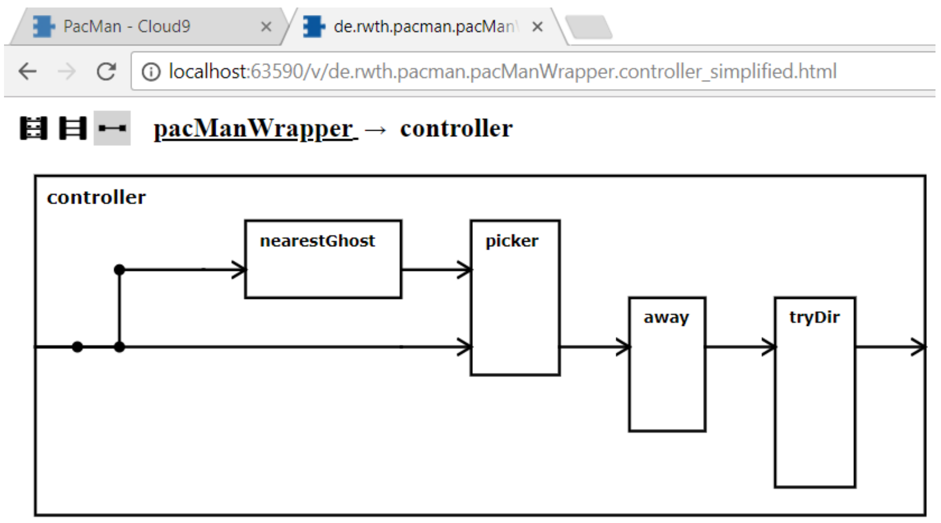

PacMan - Cloud9
de.rwth.pacman.pacMan
localhost:63590/v/de.rwth.pacman.pacManWrapper.controller_simplified.html
pacManWrapper → controller
controller
nearestGhost
picker
away
tryDir

Figure 15: Visualization of the PacMan controller (simple)

- 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 tries 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.
This design is very simple and not very successful. It shows the concept of C&C modeling in its basics and is therefore listed here. The next controller is a lot more complex and can easily beat up to 10 levels.

4.3.3. C&C modeling - PacMan (complex)

The more complex PacMan controller is shown in fig. 4.3.3. It has three main subcomponents:

- *safePaths*: This component is responsible for checking all the paths leading from PacMan into the labyrinth for safety. This is done by searching in each of the four possible directions until a wall or intersection is found.

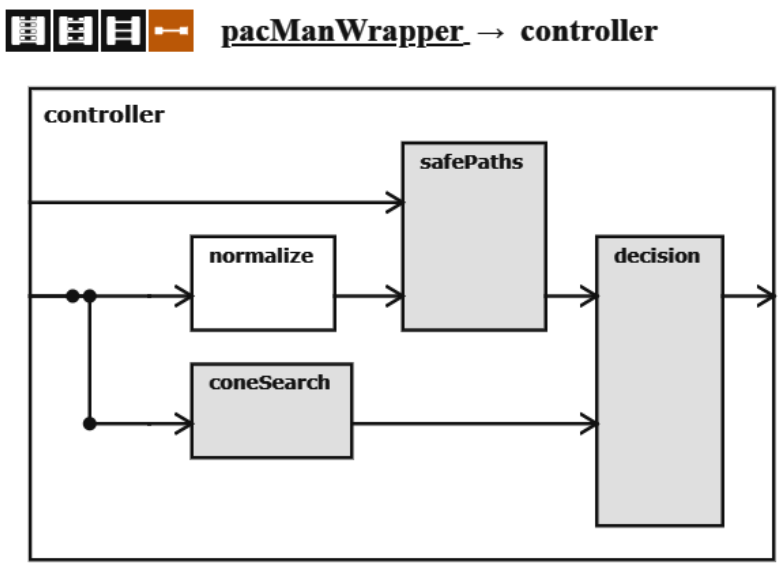


Figure 16: Visualization of the PacMan controller (complex)

- *coneSearch*: Searches in cones in each of the four directions for enemies and coins and gives back a score for each direction.
- *decision*: The decision component evaluates all data from the other two components. Based on those values it decides which direction the PacMan should go next.

The last component *normalize* not listed here is only responsible for increasing all position values from the ghosts and PacMan by 1 to fit the indexation from the math library. We will now go into detail for the three main component.

Safe Paths

In fig. 4.3.3 the *safePaths* component is shown. It contains a subcomponent for each direction and some starting values. It gives back whether the four directions are safe or not. A direction is safe if there is a wall blocking it (no path) or there is no enemy on its path until the next intersection. This is calculated by "going" the path. This could be done with a single component looping through the path to the next intersection. Due to the fact that this would contradict the conditions on C&C components stated before it is split up into subcomponents. Each path in this labyrinth has a length of at most 10. So the task is divided into 10 components as one can partially see in fig. 4.3.3. Each of those checks whether the current position is safe and then calculate the position to check for the next component. This way the runtime is fixed and the code is better parallelizable.

The task of one of the 10 subcomponents is again split up into 5 subtasks (see fig. 4.3.3):

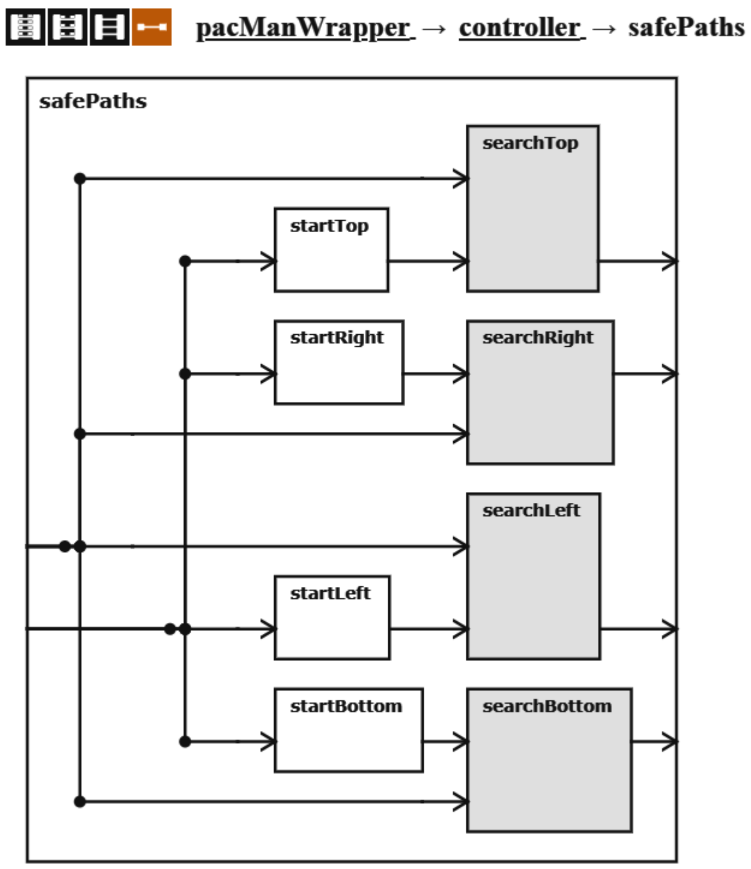


Figure 17: Visualization of Safe Paths

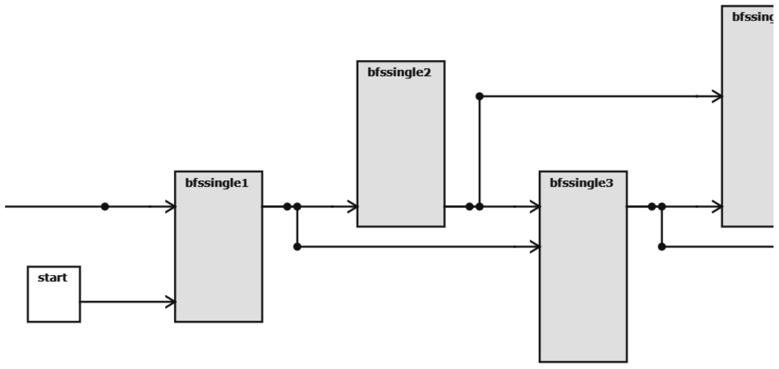


Figure 18: Visualization of one Search

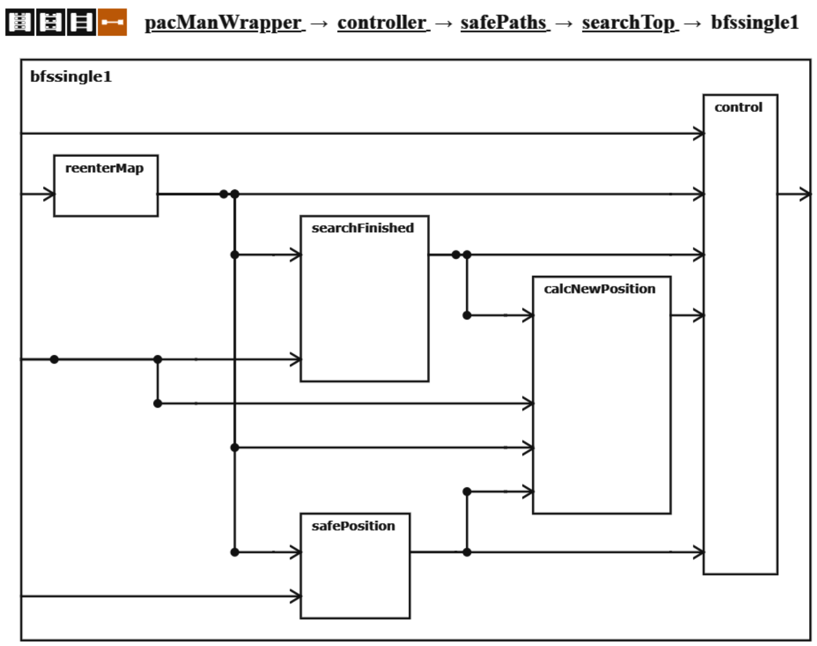


Figure 19: Visualization of one Single Search Component

- *reenterMap*: If the previous component calculated a position outside of the map (e.g. leaving the map on the right through the tunnel), reenter the map on the other side.
- *safeFinished*: The search is finished if it is marked as finished by a previous search component or a wall is found (only when there is no path).
- *safePosition*: Loops through the four ghosts and check whether their position matches the current position. If an unsafe tile is found, the search is marked as finished and not safe.
- *calcNewPosition*: Looks for free ways in the adjacent tiles. If there are more than two free tiles (no wall), an intersection is reached and the search can be marked finished. Otherwise this component gives back the next position which is different from the previous one.
- *control*: The control unit evaluates all data from the other components and gives back a corresponding new position and whether the search until now is safe or not.

Cone Search

The *ConeSearch* component searches through the map in cones (see fig. 4.3.3). This way each direction can be given a value which increases when biscuits and coins are found and decreases when ghosts are found. The following weights are used in the most current version:

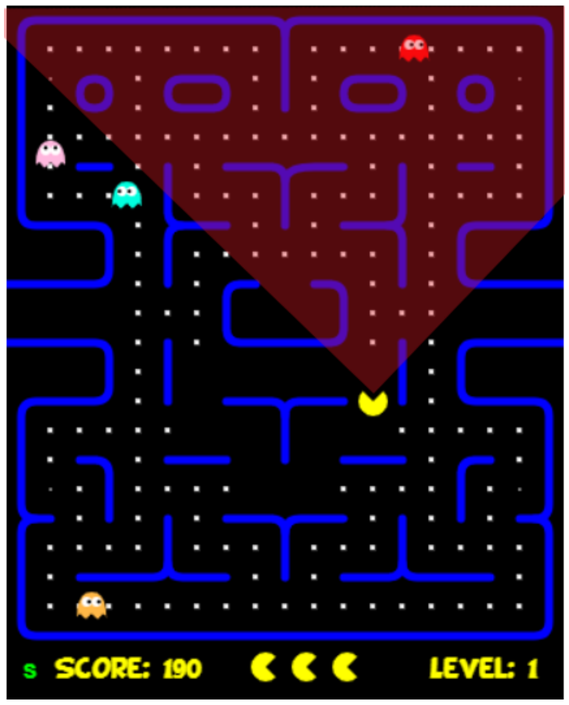


Figure 20: Visualization of Cone Search

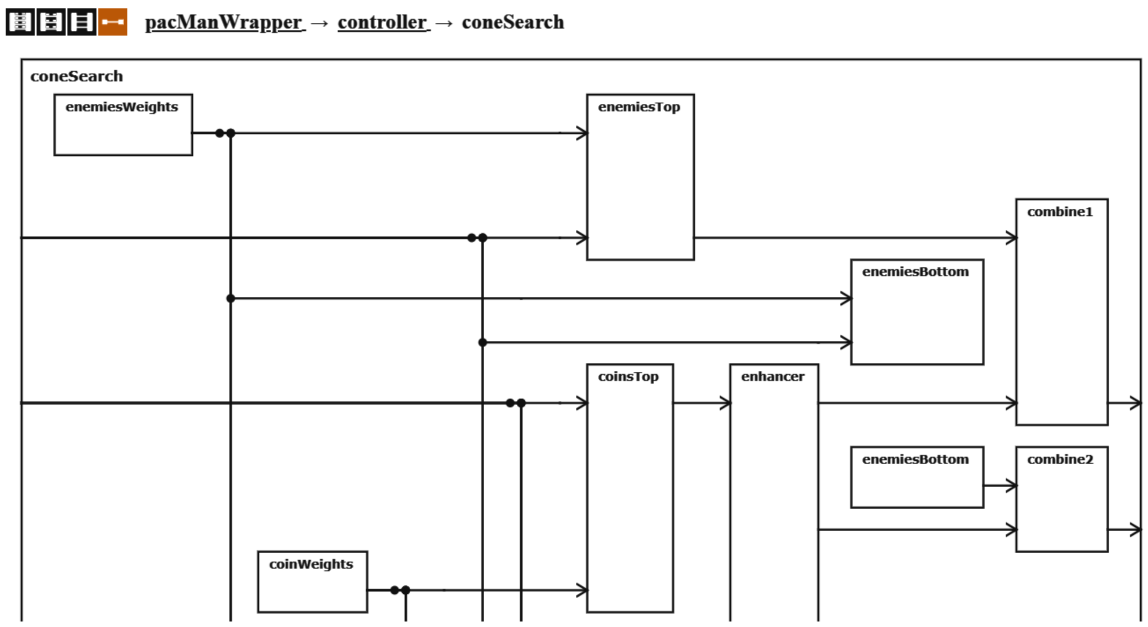


Figure 21: Visualization of Cone Search

- biscuit: 50
- coin: 200
- enemy (facing towards PacMan): -10
- enemy (facing a different direction): -4
- enemy (eatable): 1000

The values shrink with the distance to PacMan. The biscuit/coin value shrink squared and the enemy value linear with the distance. This way near objectives are valued more and PacMan does not go for only far away biscuits/coins if there already are nearby ones. But if all biscuit/coin values are small the maximum gets increased by a fix amount, so PacMan goes for far away biscuits/coins if there are no around. In the end for each direction a value is returned by combining the biscuits/coins value and the enemy value.

In the visualization of the component (see fig. 4.3.3) one can see the different kind of subcomponents:

- *enemiesWeights* and *coinWeights*: some constants for weighting biscuits, coins and ghost values. This design allows easy adjustments.
- *enemies(Top)*: searches for enemies in the (top) cone and gives back its value.

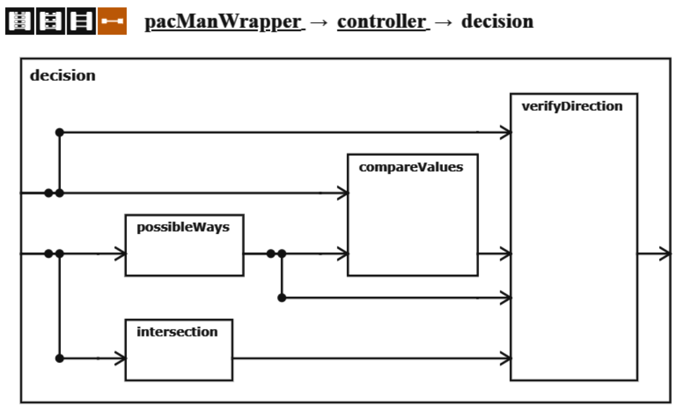


Figure 22: Visualization of Cone Search

- *coins(Top)*: searches for biscuits/coins in the (top) cone and gives back its value.
- *enhancer*: increases the maximum biscuits/coins value if it is small.
- *combine*: combines the values for biscuits/coins and enemies for a direction.

Decision

The *decision* component gets all data from *safePaths* and *coneSearch* and makes a final decision on where to go. Beside the maximum value for a direction and whether it is safe or not, the decision is based on a few additional information. E.g. the top direction has the maximum value from the cone searches but it is blocked by a wall or not safe. Then another direction has to be chosen. Here an orthogonal direction (left or right) is preferred to stay near to the desired one (top). In addition, to prevent stuttering a new path is only chosen if the current one is not safe anymore or an intersection is reached. In fig. 4.3.3 one can see the four subcomponents of *safePaths*:

- *intersection*: Gives back whether PacMan is located on a tile with more than 2 Paths leading from it.
- *possibleWays*: Gives back which directions are not blocked by a wall.
- *compareValues*: Calculates the safe direction with the maximum value. If this direction is blocked, a new direction has to be chosen.
- *verifyDirection*: Checks whether the chosen direction is opposing the previous one. This is only allowed if the previous direction is not safe anymore or an intersection is reached.

4.4. Modeling - Supermario (by Philipp Haller)

This part discusses the model used to solve a level of the Supermario game. First a general introduction on model types is given. Thereafter, the different models are discussed step by step, beginning at the most abstract.

4.4.1. Model Types

In this context the following model types used were:

Watcher
The watcher model type takes a position as input and returns a boolean value which indicates if it is in a certain range.

Selector
The selector model type uses a raw array and an index as input and returns the corresponding array entry.

Strategy
A strategy model type can take different inputs and performs a action decision based on its inputs.

Controller
The controller model type combines the other defined model types to refine the inputs of the simulation and executes a strategy.

Filter
The filter model type is intended to perform filtering like debouncing and plausibility checks.

4.4.2. Models

The presented model visualizations are generated from the EmbeddedMontiArc Studio. Therein, a grey component indicates that the component uses additional subcomponents, whereas a white component marks atomic components.

Split -¿ Reference

The first and most abstract entity modeled was the supermario wrapper which is closely related to the outputs and inputs of the simulator. Therefore it receives all necessary values as input with the aim to forward them to the actual controller and its corresponding sub-components. After computation the results of the controller are handed back into the wrapper, which forwards the data to the simulator. Figure 23 shows the graphical representation, while listing 9 shows the actual EMA interface definition.

Listing 9: Interface of the Supermario Wrapper

```
    ports
        in Z^{1,2} marioPosition ,
        in Z^{1,2} marioVelocity ,
```

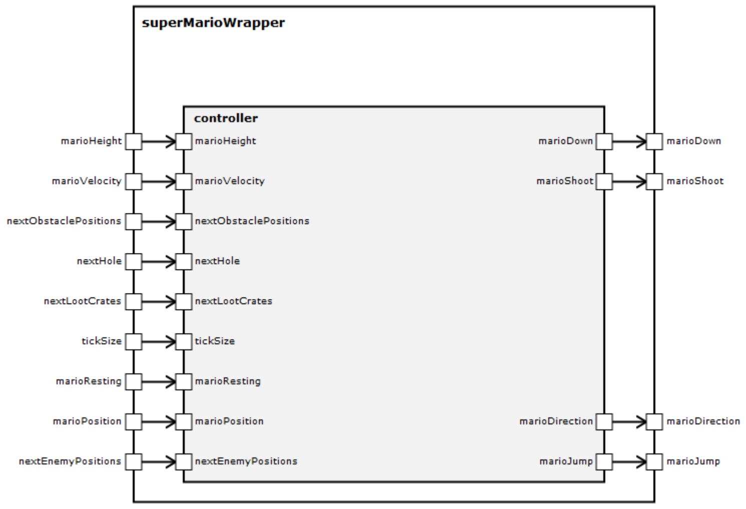


Figure 23: Visualisation of the Supermario wrapper model

```
        in Z marioHeight,
        in Z^{5,2} nextEnemyPositions,
        in Z^{5,2} nextObstaclePositions,
        in Z nextHole,
        in Z^{5,2} nextLootCrates,
        in Q tickSize,
        in Z marioResting,
        out Z(-1 : 1 : 1) marioDirection,
        out Z marioJump,
        out Z marioDown,
        out Z marioShoot;
```

The player figure's position, velocity and height were chosen as inputs, together with the positions of the next five enemies and obstacles. Furthermore, the position of the next hole in the ground, the position of the next five loot crates, the tick size (the time between model executions) and the information if the player is resting on a tile is given. The outputs consist of the direction the player shall go in combination with the action instructions jumping, crouching and shooting. The data type for most values is integer, indicated by a "Z" in the code. This is due to the circumstance that the simulator uses a number of pixels as a measure for distance. Only exception being the "tickSize" which can be fractions of a second.

The controller used (Figure 24) consists of five parts. There are sub-controllers tasked to cope with the evaluation of enemies and obstacles respectively, named enemyController and obstController. They return an advice to indicate if the player should jump or not. The genStrategy is an atomic component which is

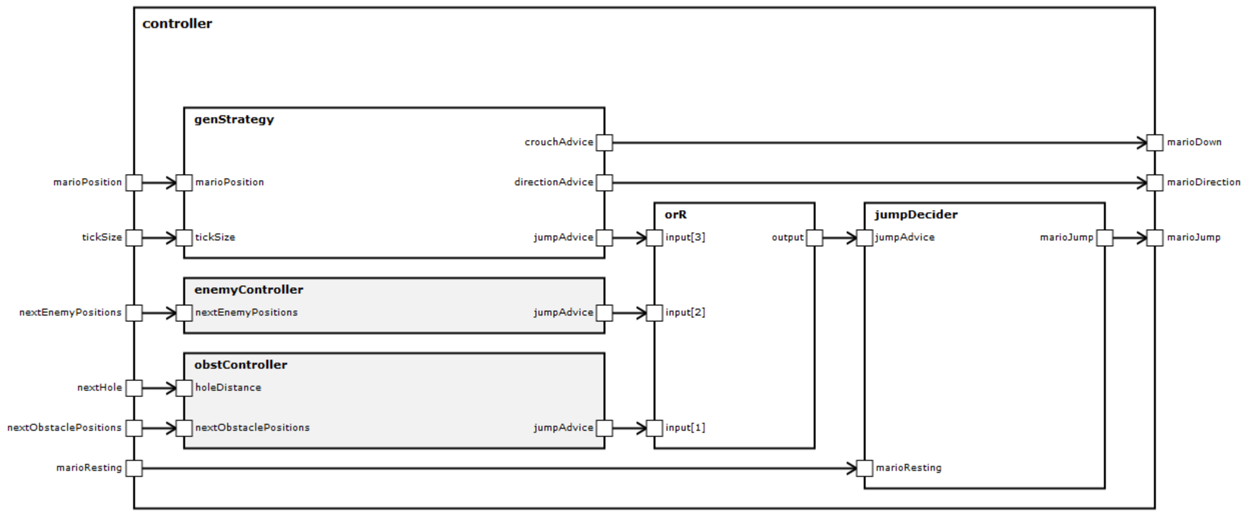


Figure 24: Visualization of the Supermario controller model

currently used to provide a general strategy like moving in another direction, jumping or crouching if the player is stuck.

The action advices of the controllers and the strategy are combined via a logical or-relation, as indicated by the ”orR” block. Additionally, the jumpDecider filters the output of the combined value and forwards it, if the player can jump in that time frame. This is necessary to prohibit side-effects like the player only jumping once because the jump key remains pressed constantly and the simulator only accepts distinct jump activations, opposed to continuous jumping.

The enemy controller (Figure 25) handles the enemy position evaluation and assesses if an action has to be initiated. As the input data from the simulator is a array with five positions, it contains a enemy selector component which returns the corresponding x and y values from a given index. For purposes of overview and readability of the EMA code a component ”enemyIndexes” was used to feed these indexes into the selectors.

The enemy component (Figure 26) is used to compute a velocity from the x and y positions by comparing the former positions with the current ones.

The enemy strategy (Figure 27) uses the distances and velocities from the enemy components to watch them for their distance to the player and wether they can get dangerous. If an enemy comes too close and is on the player’s plane, a jump advice is given. The jump advices are again combined via a logical or-relation and returned.

The obstacle controller is modeled very similar to the enemy controller, extracting positions from the raw input array and feeding them into a obstacle strategy. The main difference to the enemy controller is the presence of another

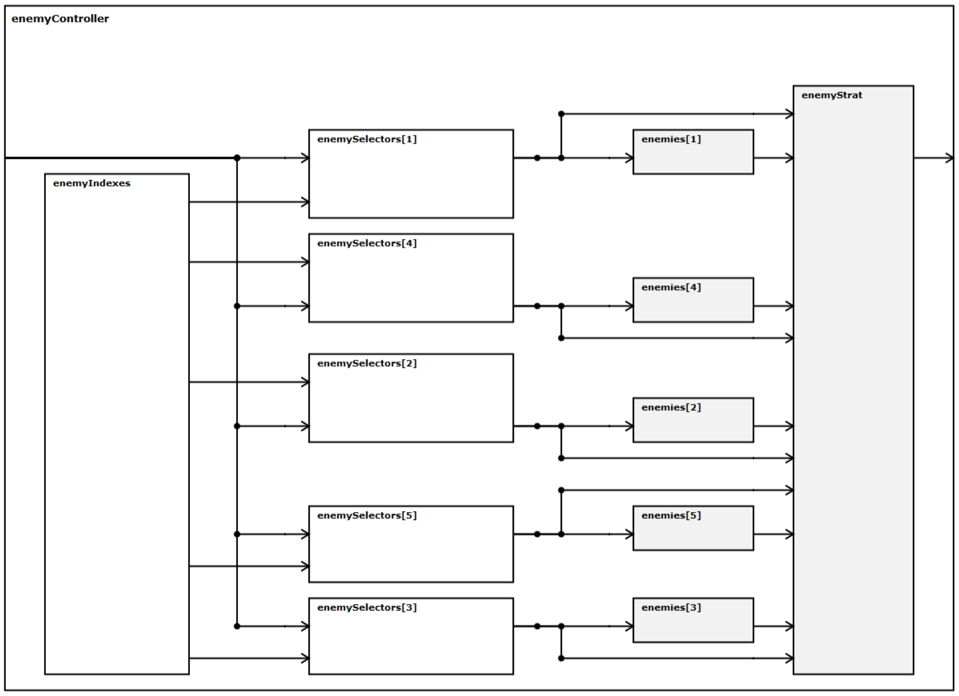


Figure 25: Visualization of the Supermario enemy controller model

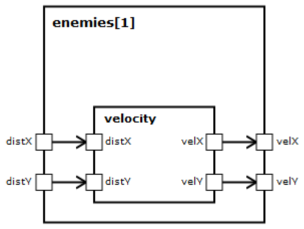


Figure 26: Visualization of the Supermario enemy model

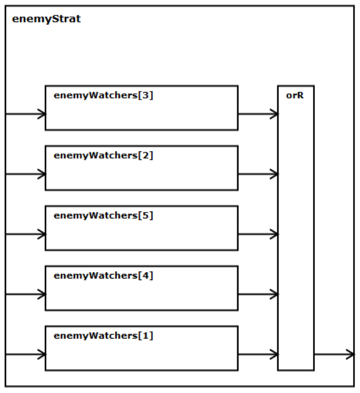


Figure 27: Visualization of the Supermario enemy strategy model

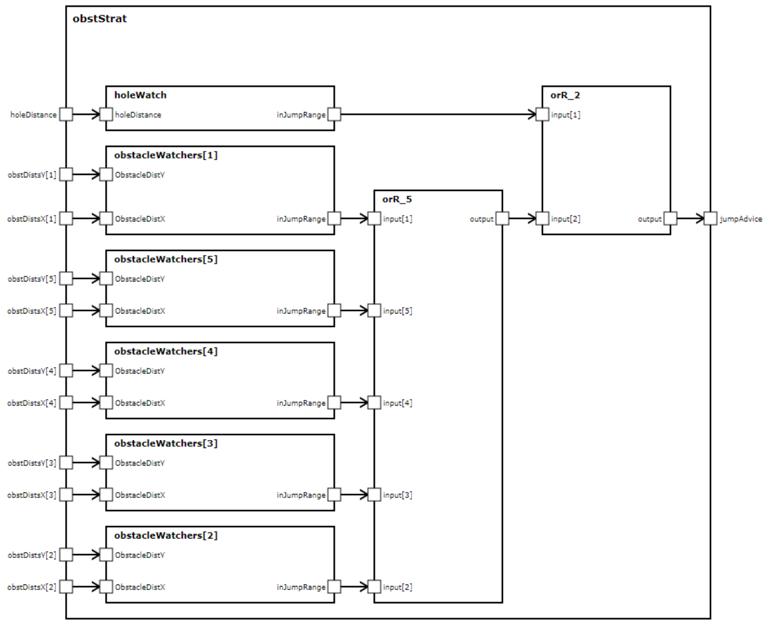


Figure 28: Visualization of the Supermario enemy strategy model

input. This additional input is the distance to the next hole in the ground plane of the level. It is forwarded into the obstacle strategy (Figure 28) where a watcher component checks the player's proximity to the hole and computes a jump advice. All advices are again combined by a or relation.

4.4.3. Future Modeling

The models presented in this chapter were developed with modularity and extensibility in mind, such that in future work more complex strategies can be used to solve more levels and to lay more attention to the score. The presented model utilizes that the player always runs into the right direction, thus it can't solve levels which require the player to move backwards. A future model should be able to solve those situations too. This behaviour could be modeled in the general strategy component or a "movement controller". Another issue could be, that currently all advices are combined via or relations. This can lead to side effects where the player jumps to early because of an enemy and drops into a hole he would have avoided without the enemy. To achieve a better model, the or relations could be swapped with a weighted decision making process.

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 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] D. Harvey, Html5 pacman.
URL `https://demo.embeddedmontiarc.com/pacman2/`

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

[12] A. Zakai, Emscripten.
URL `http://kripken.github.io/emscripten-site/`

[13] E. Team, Embeddedmontiarc documentation.
URL `https://github.com/EmbeddedMontiArc/Documentation`