Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2018) 1–17

Procedia Computer Science

This paper is a seminar paper of two students; the seminar teaches students to write scientific papers.

This paper has NOT been submitted!

Extending the EmbeddedMontiArc Language Family by an Application Language

Fabian Kahlert[1], Sascha Schneiders[1]

Supervised by Michael Wenckstern and Bernhard Rumpe, Software Engineering, RWTH Aachen University

Abstract

Component and Connector models are commonly used when developing cyber-physical systems. As cyber-physical systems manage safety-critical functions like the control of a reactor, or an autonomous vehicle, ensuring correctness and real-time responsiveness is required. While these are no hard-requirements in most software projects, fast execution times and correct behaviour are still important common requirements. This paper presents how the `EmbeddedMontiArc` language family can be extended with interactive simulations support 2D visualization. A collection of simple requirements needed to fulfillthis, as well as the adaption and reusage of already existing tooling is explained. This also includes the current EmbeddedMontiArc language family, as well as the full approach, and the presentation of the most important requirements. Furthermore, a C++ backend for handling all the visualization related functionality was created. Additionally, the creation of a simple Pac-Man simulation is discussed. Pac-Man simulations serve as a test-bed for artificial intelligence.

Keywords: EmbeddedMontiArc, MontiCore, Code Generation

1. Introduction (S)

The development of new software is still an art that requires a precise and thoughtful structured approach. If software is not developed carefully, it results in faults that either increase development costs during the late phases of the development process, or in the worst case after the project is completed. The end of the development phase is reached once a project is called complete. Once a customer has received an application there usually is still support requirement and if necessary further modifications that are to be applied to the software. Development costs usually increase the later an issue is detected. To make early error detection available as soon as possible there exist different approaches. One of these approaches that is also extensively used in the industry is the model-driven development approach. When developing software model-driven several different tools can be used. One of the most popular software tools for embedded systems that exist is Simulink [1]. It was one of the first available tools that aided a model-driven development process. However, it provided no native SI-Unit support until version R2016a [2] and also no standard method of testing created models in a very simple way. The EmbeddedMontiArc [3] language family was created as a research project to examine the benefits of developing a complete toolchain that considered model-driven and

Email addresses: `fabian.kahlert@rwth-aachen.de` (Fabian Kahlert), `sascha.schneiders@rwth-aachen.de` (Sascha Schneiders)

test-driven development as one of its most important requirements from the beginning. It enables the description of Component and Connector models in a textual way. Additionally, a behaviour description can be added by using the MontiMath language that is embedded into the EmbeddedMontiArc language by using the language embedding feature of MontiCore [4]. MontiCore is a language development workbench that internally uses ANTLR [5] for automatic parser generation that are based on MontiCore grammars. There also exist different middleware extensions for supporting existing frameworks like the Robot Operating System (ROS) [6] and OpenDavinci [7]. To reuse and extend the already existing EmbeddedMontiArc language family it was extended by an `Application` language. The performed requirements analysis is discussed in Section 3. An explanation of the whole approach also serves as a means of understanding how the EmbeddedMontiArc language family can be extended. This can also be useful for future projects, which might aim to extend the EmbeddedMontiArc language family, as it is explained in Section 4. The approach is presented in a general and specific matter, so it can also be used as a guidance or aid when developing a domain-specific language in general.

2. Preliminaries (F)

2.1. Component and Connector Modeling

Component and Connector [8] modeling, also referred to as *C&C* modeling, is a modeling technique that is heavily employed in the industry when developing cyber-physical systems. Furthermore, it is employed in Simulink [1], which is part of the Matlab [9] software suite. In *C&C* models, components are used to represent logical functionality. The data exchange between different components is depicted by connectors. Components may also contain subcomponents. Therefore, certain levels of abstraction can be hidden and only be examined once required.

2.2. EmbeddedMontiArc

The Architecture Description Language (ADL) EmbeddedMontiArc is a textual *C&C* modeling language that extends the ADL MontiArc. EmbeddedMontiArc provides new features as for example port arrays, component instance arrays, generic parameter binding and units. Furthermore, a strictly typed math language – EmbdeddedMontiArcMath (EMAM) – is part of the language family and is used to describe behavioural properties of components. The strict typing can be leveraged to enable significant algebraic optimizations, especially pertaining matrix computations, but also allows to catch type errors at compile-time like misfitting units or wrong matrix dimensions. Threading optimizations can be applied additionally (see [10]). The language family aims to facilitate model-driven development for embedded and cyber-physical systems, especially in the context of the automotive domain, and generates highly optimized code.

Since EmbeddedMontiArc incorporates the *C&C* modeling principle, components are used to model embedded and cyber-physical systems, including their interactions. Each component has an interface, consisting of in- and output ports which are used to connect components using connectors. Furthermore, components may include instances of other components as subcomponents. In the following all important concepts are explained in more detail together with textual examples of EmbeddedMontiArc code.

2.2.1. Components

Components have a name and optionally a package they belong to and usually are separately stored in `.ema` files. Components generally – but not necessarily – consist of ports, subcomponents and connectors. They are textually defined with the keyword `component` followed by its name and curved parentheses `{ }`. Component's names are by convention capitalized. The parentheses mark the body of the component, everything that is contained by a component needs to be defined within these.
Note that the component's name and the file name should be equal.

```
component ComponentName {
}
```

Listing 1. Example EmbeddedMontiArc code for an empty component.

2.2.2. Subcomponents

A component is defined as a subcomponent of another component by instantiating it within the body of the parent component. An instantiation is accomplished by using the keyword `instance` followed by the respective subcomponent and the instance's name. It is convention to give component instances non-capitalized names.

```
component ParentComponent {
  instance SubComponent subcomponentname;
}
```

Listing 2. Example EmbeddedMontiArc code for subcomponent instantiation.

2.2.3. Ports

Ports define the interface of a component. Each port consists of a direction (`in` or `out`), a type and a name. Ports of a component are textually defined by the keyword `port` or `ports` followed by the port's direction, type and name – per port separated with commas (`,`) and finished with a semicolon (`;`). The type may be a Boolean `B`, a range or a struct. Ranges may either be given as `(start:step:end)` or `(start:end)`, where `start`, `step` and `end` are given as numbers (e.g. `1` or `-0.5`) and optionally contain a unit declaration like `m/s`. Furthermore, `start` and `end` can be replaced by `oo` or `-oo`, denoting infinity or minus infinity. A missing `step` declaration assumes it to be as small as possible. The ranges of integers, natural numbers and rationals are predefined and can be used with the following abbreviations:

```
N0:   (0:1:oo)
N1:   (1:1:oo)
Z:    (-oo:1:oo)
Q:    (-oo:oo)
```

To denote arrays of ports append `[size]` to the ports name where `size` describes the size of the array. Note that the indexing of port arrays in EmbeddedMontiArc starts at `1`.

```
component ExampleComponent1 {
  ports
    in (0m:0.05m:1m) in1,
    out N1 out1[4];
}
```

Listing 3. Example EmbeddedMontiArc code for a component with two ports.

2.2.4. Connectors

Connectors directly connect two ports and represent the flow of data between these. As connectors are directional, connectors have a *source port*, the port where the connector starts, and a *target port*, the port where the connector ends. It is important to notice that exactly four cases of connections occur:

- `IN` → `IN` An incoming port is connected to another incoming port when incoming data is passed to a subcomponent.
- `OUT` → `OUT` An outgoing port is connected to another outgoing port when outgoing data is passed from a subcomponent to its parent's outgoing port.

- `IN` → `OUT` An incoming port is connected to an outgoing port if data directly passes a component without being influenced.
- `OUT` → `IN` An outgoing port is connected to an incoming port whenever a component's data is passed to another component on the same hierarchy level.

In other words: A component cannot influence the data flow inside of its subcomponents.

Connectors are textually defined using the keyword `connect` followed by the source port, an arrow (`->`) and finally the target port. Optionally, multiple target ports can be specified by separating each one with a comma, similar to ports.

```
1  component ExampleComponent {
2    ports
3      in B in1,
4      out B out1;
5
6    instance AtomicComponent atomicCmp;
7
8    connect in1 -> atomicCmp.in1;
9    connect atomicCmp.out1 -> out1;
10 }
```

Listing 4. Example component with intercomponent connections.

```
1  component AtomicComponent {
2    ports
3      in B in1,
4      out B out1;
5  }
```

Listing 5. Simple atomic component.

2.2.5. *Component Generics*

Listing 6 shows three examples of possible component generics declarations. The generics can be divided into two categories, type and type instance generics.

The first example presented in line 1 shows a component called `TypeGenerics`. This component uses type generics. Ports inside of this component can use `T` as their type. The component has to be instantiated with a concrete type, like `Boolean`, that replaces `T`.

The second example depicted in line 4 shows a component called `NumberGenerics`. `NumberGenerics` shows an application of type instance generics. In this case a natural number that is greater or equal to one is expected when instancing this component. Inside of the component `n` can be used to denote the size of a port or component array.

The third example shown in line 7 displays a component called `MultipleGenerics`. This component uses two generics definitions. A type and a type instance generic. Similar to the `TypeGenerics` and the `NumberGenerics` component, `T` can be used to denote a port type and `n` can be used to set the size of a port array.

```
component TypeGenerics<T> {                 //type generics
  //...                                     //body
}
component NumberGenerics<N1 n> {            //number generics
  //...                                     //body
}
component MultipleGenerics<T, N1 n> {       //multiple generics
  //...                                     //body
}
```

Listing 6. An example of three components which use component generics.

2.2.6. Component Parameters

A component parameter is configuration data that is passed to a component. It can be utilized in many different ways. Two different use cases of component parameters are listed in the following.

- It can be used to provide an initial configuration. Component `Delay` which can be examined in Listing 7 shows the EmbeddedMontiArc implementation of a component that delays its input to one execution tick later, and supports setting an initial value by using the component parameter `value`.

```
component Delay<T>(T value) {
  ports
    in  T in1,
    out T out1;
}
```

Listing 7. EmbeddedMontiArc implementation of a delay component.

- It can be used to implement a `LookUp` component as seen in Listing 8. It is initialized with an array of data. This array of data is then used to look up parts of the data, once it is required.

```
component LookUp(Q^{1,n} lookupTable) {
  ports
    in  T in1,
    out Q out1;
}
```

Listing 8. EmbeddedMontiArc implementation of a lookup component.

2.2.7. MontiMath

The language MontiMath supports two different types, namely `Matrix` and `Boolean`. A n-dimensional vector, is a $n \times 1$-matrix and a number is a 1-dimensional vector. All types can be furthermore separated into integer and rational number using either `Q` or `Z` in the declaration. It is possible to directly assign and declare variables in the same line.

```
Q a = 5.2;                  //Rational number
Z b = 5;                    //Integer number
Z^{2,1} c;                  //Integer column vector
Q^{2,2} d = [1 0; 0 1];     //Rational 2x2 matrix
```

Listing 9. Example MontiMath variable declarations.

2.2.8. Math For-Loops

The only loop control structure which is currently present in the MontiMath language, is the for-loop construct. The following example shows the usage of a for-loop in the MontiMath language. A loop variable is used, which changes its value during the runs of the loop. The expressions inside the loop body define what happens during each execution step of the loop.

```
for i = 1:1:3 //i will take the values 1, 2 and 3
  //...
end
```

Listing 10. Example of a for-loop structure.

2.2.9. Math Conditional Statements

The MontiMath language also offers conditional statements. A conditional statement starts with the keyword `if` and is then followed by a condition. If the condition is true, the body of the `if` statement is being executed. A condition has to be an expression that returns either true or false. An `if` statement can also be followed by one or more `elseif` statements and it is also possible to end a conditional statement with an `else` statement. A conditional statement ends with the keyword `end`. The following example shows the general syntax of a conditional expression where an `elseif` and `else` statement are present.

```
if condition
  //...
elseif condition
  //...
else condition
  //...
end
```

Listing 11. Example of a conditional statement.

2.3. OpenGL

Open Graphics Library (OpenGL) [11] is a software interface which provides access to graphics processing units (GPUs) and is available on all major desktop computer platforms like Linux, Windows and Mac OS. We use GLEW (OpenGL Extension Wrangler Library) as extension wrapper for handling the on the hardware available OpenGL capabilities. It is used with OpenGL 3.3 for compatibility reasons.

2.4. Simple DirectMedia Layer

The Simple DirectMedia Layer (SDL) [12] is a cross-platform framework that provides a common abstraction over low-level functionality. This includes audio, graphics, mouse and keyboard abstraction. The major supported platforms include Linux, Mac OS X, Windows, Android and iOS. It is developed in the C programming language [13], but there are also wrappers for different languages available, so that SDL can be utilized in other languages too. Therefore, many applications which either require input, graphics hardware or audio support, employ SDL internally. This includes emulators like DOSBox [14] and Visual Boy Advance [15].

3. Requirement Analysis (F, S)

In the following a list of requirements is presented, which we want our language to fulfill. The requirements are explained in more details later on.

R1 rendering textures

R2 handling sprites

R3 animations

R4 handling input

R5 simple language

R6 interactive controls

R7 management of stages

R8 world management

R9 description of logical interactions

R10 description of mathematical properties (e.g. collision detection)

R1: Rendering textures is required to display two-dimensional graphical assets that were created by external software like Paint.NET [16], or GIMP [17].

R2: Separating details like actual position and size from the actual texture is a necessity if textures shall be reused for different entities. To accomplish this, the combination of these attributes and an actual texture are accomplished in so called sprites.

R3: Animations are usually rapidly switching textures, each displaying a part of the desired effect. This can be a moving character, or a door that is opened, or closed. They serve as a means of increasing immersiveness.

R4: As interactive simulations require user input, an interface that exposes input device information like keyboard key, or mouse button presses is needed.

R5: In contrast to general purpose languages, we only want to describe relevant concepts of our domain with our application language.

R6: Interactivity plays a crucial role in many applications. To achieve this, the user must be able to control the application.

R7: Hiding and displaying several sprites which are grouped together in a scene, enables better management when the visibility of a lot of sprites changes. This includes the showcase of menus or overlays that are only intended to be visible during certain situations.

R8: To create, manage and modify data that is the basis of a simulation or video-game application like Pac-Man [18] or Super Mario Bros. [19], means to represent and modify that data are required.

R9: To handle behavioural changes that depend on certain conditions, logical instructions are required. This includes boolean-like types and if cases.

R10: For handling collision detection, positional updates, etc.: Mathematical instructions are required.

Depending on the context additional requirements can be gathered, like showing text by specifying the strings directly. However, this can also be achieved by using textures directly, as text on computers is usually rendered by using fonts, and fonts themselves are textures that contain the needed letters for display.

3.1. Pac-Man

Pac-Man is a classical arcade game that was released in 1980 by the company Namco Limited. It is also used in scientific research as a platform to test artificial intelligence [20, 21]. Therefore, we choose Pac-Man as the evaluation example for an environment that is modelled with the EmbeddedMontiArc language familiy and our newly added extension, which provides application related functionality. The required functionality in the Pac-Man example is mainly graphical rendering and the description of the world rules. As we also want to be able to give application users the possibility to interact with the application, user input is also a requirement.

The graphical rendering functionality and the input handling functionality are provided by our application language. The description of the Pac-Man world and its rules can be done using the EmbeddedMontiArcMath language.

4. Approach (S)

The general approach when designing a new DSL usually includes several different phases. At the beginning, the requirements analysis phase takes place. The design phase follows next. Similar to other software projects, the implementation phase comes afterward. After the implementation is completed, a short evaluation of the results takes place, which rates the fitness of our language for different purposes. This usually starts by checking whether the requirements are truly fulfilled. This approach is similar to the waterfall-model [22] that is an established software development methodology. However, similar to the V-model, there are some issues with this approach that can be avoided by refining the approach to be more agile. If the requirements analysis has not been performed very carefully, some important requirements might have been overlooked. To enable early detection of this issue, a prototype should be developed as early as possible. Going back to the previously described model, this results in an interaction between the described phases. Therefore, once it is possible to describe some examples, which were examined for the requirements analysis, like Pac-Man, are defined during the design phase while the language is developing. Once the description is complete, or the realization that it is not possible to describe the example arose, the requirements are examined again. If a needed feature is not part of the requirements, it needs to be added. If this feature is not part of the already gathered requirements, this shows the incompleteness of the requirements and that a refinement of these is needed. This continues until the language is complete and there are no further evaluation examples that suggest the language to be incomplete. So it all starts with a certain problem.

In our case the problem in particular is the desire to create simulations and interactive applications with EmbeddedMontiArc without the need of adding handwritten target language code. As EmbeddedMontiArc already contains a rather extensive toolchain, the idea of reusing as much of it as possible during the implementation phase quickly arose. The most important requirements are presented in Section 3. As the EmbeddedMontiArcMath language is already a member of the EmbeddedMontiArc language family, it became obvious from looking at the capabilities described in Section 2.2 that the EmbeddedMontiArcMath language provides enough features to fulfill the requirements **R9** and **R10**. To fulfill the requirements that are presented in Section 3 the application language was developed. It is described in Section 5. To handle input, graphics and audio SDL was chosen as a framework, as it provides a common interface that works on Linux, Mac OS X and Windows.

5. Application Language (S)

To fulfill the extracted requirements we extended EmbeddedMontiArc with an additional minimalistic behaviour language that is called `Application`. The `Application` language consists mainly of two different parts. One is the input interaction, and the other the render interaction part. As the names suggest, the purpose of the input interaction part is to provide access to input information like keyboard keystrokes, or mouse clicks. The render interaction part enables the management of four distinct elements that are part of our graphical abstraction and match the requirements presented in Section 3. The `Application` language needs no additional description capabilities to fulfill requirement **R9** and **R10** due to EmbeddedMontiArcMath being able to fulfill these requirements.

Our developed generator integrates the already existing EmbeddedMontiArc toolchain which enables us to generate C/C++ code usable on the operating systems Linux, Mac OS X and Windows. All input and render instructions are written inside of the `implementation Application` body, as shown in Listing 12. It works similar to the `implementation Math` behaviour description in EmbeddedMontiArcMath. However, as EmbeddedMontiArcApplication features side-effects due to how OpenGL works it does not use *C&C* components but *Actuators*. The syntax is kept similar to the syntax of components, but the keyword `actuator` is used instead of `component`. Components are side-effect free and thus have always one input port, when the output port is not a constant value all the time [23]. Variables are also similarly defined to component's port definitions, they differ in the keyword `variable(s)`. They also can be connected with connectors the same way ports are connected in EmbeddedMontiArc.

```
actuator MainApplication {
    implementation Application {
        //Render and Input Instructions inside here
    }
}
```

Listing 12. Simple EmbeddedMontiArcApplication example.

5.1. Input Interaction

Listing 13 shows an exemplary implementation of an actuator handling the user input for Pac-Man. Four boolean output variables are sufficient since Pac-Man is not capable of any actions except moving vertically or horizontally at constant speed. The following `Application` body controls the output of the actuator. The shown syntax binds the output to the boolean keystate of the given key. Despite keyboard key inputs like `"w"`, special keys like `"MB0"` indicating a left click, or `"MB1"` for a right click can be used. These boolean values indicate whether a press has occurred or not.

```
actuator PacmanInputControl {
    variables out B moveDown,
          out B moveUp,
          out B moveLeft,
          out B moveRight;

    implementation Application {
       set variable moveUp to w;
       set variable moveLeft to a;
       set variable moveDown to s;
       set variable moveRight to d;
    }
}
```

Listing 13. Simple PacmanInputControl actuator used in the Pac-Man evaluation example.

5.2. Render Interaction

Listing 14 presents the `PacmanInit` actuator which configures the application's rendering. A boolean output variable `initDone` shows other actuators whether the initiation is finished or not. It is set to the constant value `true` during initiation.

The `Application` implementation follows: At first the size of the render context of the window is set to 816×816 pixel, first x- then y-coordinates. The window is created just slightly bigger to add a window title bar to the top of it and a border. No separation sign is needed between the coordinates. The animated Pac-Man texture (called sprite) is created and configured in the following. At first four textures showing parts of Pac-Man's animation – closed mouth, half-opened mouth and open mouth – are loaded using the `load texture` command with three different identifiers to be referenced with later-on. The sprite is then created and the textures are added to the animation in order of apparel.

At first the closed Pac-Man texture, then the half-opened texture and the closed texture, and finally the half-opened texture again. So that when the Pac-Man is animated it creates the illusion of smooth movement. Finally, the `texture delay` is set to four ticks, meaning that the current texture of the sprite is replaced by the next one after four ticks, leading to a seemingly smooth animation. The number after delay is the amount of ticks that are waited before the next texture is displayed. This is quite a simple concept but it allows to create animations of different length, while also maintaining the ability to reuse the same texture in the animation.

Possible other render interaction statements are:

- deleting of textures
- setting sprites in-/visible
- adding sprites to rendering stages
- deleting sprites
- creating/deleting stages
- de-/activating rendering of stages
- hiding all stages

```
actuator PacmanInit {
  variable out B initDone;

  connect true -> initDone;

  implementation Application {
    set window size to 816 816;
    load texture pacmanClosed : "textures/pacmanclosed";
    load texture pacmanMiddle : "textures/pacmanmiddle";
    load texture pacmanOpen : "textures/pacmanopen";
    create sprite pacman;
    add texture pacmanClosed to pacman;
    add texture pacmanMiddle to pacman;
    add texture pacmanOpen to pacman;
    add texture pacmanMiddle to pacman;
    set pacman texture delay to 4; //4 ticks delay between animations
  }
}
```

Listing 14. Simple PacmanInit actuator used in the Pac-Man evaluation example.

6. Implementation (F, S)

Implementation details are discussed in this section. This includes specific parts like the created model but also the created generator and C++ backend.

6.1. EmbeddedMontiArc Model

Figure 6.1 shows an excerpt of a simple Pac-Man like simulation created using the languages `EmbeddedMontiArc`, `EmbeddedMontiArcMath` and `EmbeddedMontiArcApplication`. It depicts the parts of the main model that handle the initialization, updating and control of Pac-Man in its own simulated world. Orange components are

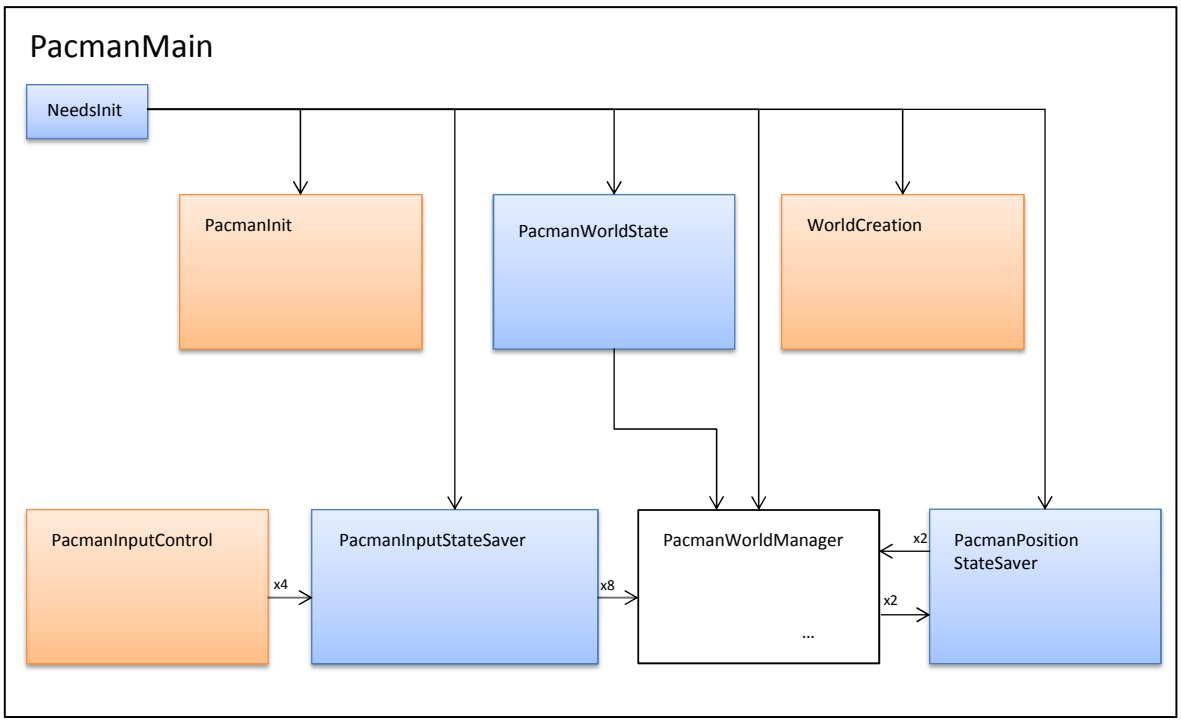

Figure 1. High level view of simple Pac-Man simulation model. Orange components are EmbeddedMontiArcApplication actuators and blue are EmbeddedMontiArcMath components.

`EmbeddedMontiArcApplication` actuators and blue are `EmbeddedMontiArcMath` components. The depicted components will be explained in the following:

NeedsInit: This component is used to signal all other components that are connected to its output port whether an initialization should take place. In the first step of the execution of the simulation model all components might need to do one time initialization behaviour. It can also be reset so that the simulation can start from initialization again while running.

PacmanInit: This actuator is an EmbeddedMontiArcApplication component and it is depicted in Listing 14. It is only executed during the first time initialization and creates the window with the given size, textures and to be used Pac-Man sprites.

PacmanWorldState: This component contains the logical Pac-Man world that is stored in a matrix. It is used to determine if the tile at a certain position is a pacdot, a wall or just empty space.

WorldCreation: Similar to the PacmanInit component, this actuator only is used during initialization to load the textures and create the sprites required to render the Pac-Man world. This includes the wall segments, and the pacdots.

PacmanInputControl: This actuator sets its variables like `moveUp`, `moveDown`, `moveLeft`, `moveRight` to true if the corresponding keys are pressed.

PacmanInputStateSaver: This component stores what input actions are currently active, like the key for up, down, left or right movement. It does also store the last pressed input states. This is required to support pressing the left button before a left movement is possible, but the left movement should occur once it is possible, i.e. Pac-Man is able to change direction to the right as there is no blocking wall anymore.

PacmanPositionStateSaver: In contrast to the `PacmanInputStateSaver` component this component does not store the current pressed input and the last input, but the current position of Pac-Man . It is updated by the `PacmanWorldManager`

component if Pac-Man is able to move to a new position. If this is not the case the position does not change between two executions.

PacmanWorldManager: As the `PacmanWorldManager` component is larger and consists of several different important components which themselves contain other non-atomic components it is separately depicted in Figure 6.1 and will be explained in the following:

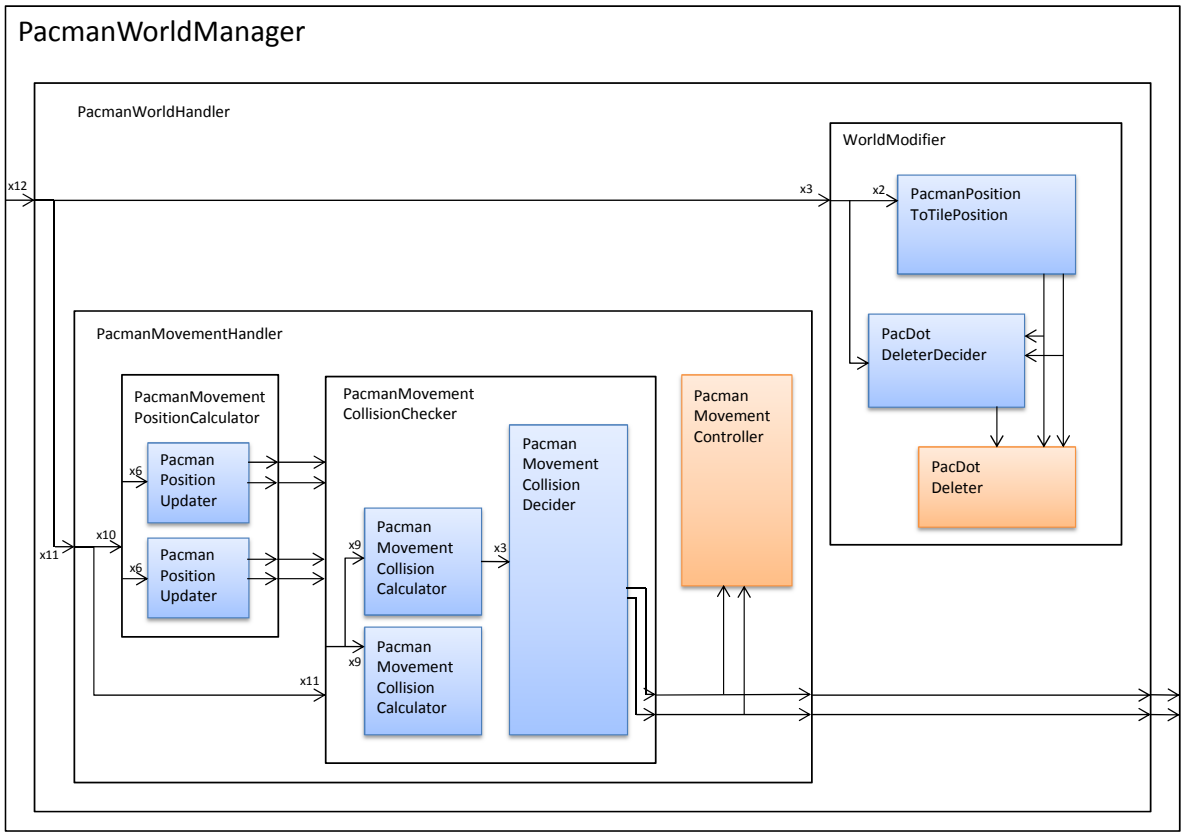

Figure 2. PacmanWorldManager view of simple Pac-Man simulation model. Orange components are EmbeddedMontiArcApplication actuators and blue are EmbeddedMontiArcMath components.

PacmanPositionToTilePosition: This component handles the conversion of Pac-Man pixel positions to tile positions. A tile is 48x48 pixels. But if this should be changed, only the `tileSize` parameter of this component needs to be changed and all logic which utilizes it will work with a different tile size then.

PacDotDeleterDecider: This component decides whether a pacdot at a certain position shall be deleted. It is supplied with information like Pac-Man entity position and whether there is a pacdot at this position.

PacDotDeleter: This actuator is responsible for handling the deletion of pacdot sprites. It is able to delete a pacdot at the position that is connected to its corresponding ports.

PacmanMovementController: This actuator changes the position of the Pac-Man entity sprite. If the position is the same as before no visible change can be observed.

PacmanMovementCollisionDecider: This component decides whether the next calculated Pac-Man position is a position that Pac-Man can enter. It is provided with the current, the next position in the current direction, and the position if it would change direction to the last pressed key where the direction change was not possible as it was blocked by a wall then.

PacmanMovementCollisionCalculator: This component is there twice because of calculating collision on current path, and collision on path that we would take if the last input was used to move. This enables pressing the left key when left is not possible yet, but is possible some time later.

PacmanPositionUpdater: Calculates the new position of a Pac-Man entity based on the given current position and the intended direction on which it should move.

6.2. EmbeddedMontiArc Language Adaption (S)

As MontiCore enables the reuse of already created MontiCore grammars similar to the extension of classes in an object-oriented programming language, the grammar of the EmbeddedMontiArc language was extended. However, the `EmbeddedMontiArcApplication` does not allow to add behaviour to components but instead to actuators as the application behaviour is not side-effect free. The reason for this is that components must express side-effect free behaviour per definition. Therefore, the starting rule of the EmbeddedMontiArc grammar which expects the keyword `component` for the definition of a component was overwritten and changed to expect the keyword `actuator`.
As variable input and output in EmbeddedMontiArcApplication is also similar to ports in components, the rule of the EmbeddedMontiArc grammar for handling expressions like `port in B isActive` was changed. Note that the keyword `port` was changed to the keyword `variable` as only components communicate via ports. The created application language that is discussed in Section 5 was then embedded into the `EmbeddedMontiArcApplication` language using the language embedding feature of MontiCore, which enables embedding one or more languages into another language. The only requirement is that the embedding language has a rule inside of its MontiCore grammar that provides an access point for extension, like an `interface` rule. For more detail regarding the usage of MontiCore consulting [4] is recommended.

6.3. EmbeddedMontiArc Tool Adaption (S)

The EmbeddedMontiarcMath2CPP generator was reused to create a new project. The class `GeneratorCPP` of the already existing generator project was extended to create a new class called `EMAApplication2CPPGenerator`. This class adds additional functionality which handles the conversion of `EmbeddedMontiArcApplication` files, while still maintaining the ability to generate C++ code out of `EmbeddedMontiArcMath` models. This means that only functionality for handling the conversion of `EmbeddedMontiArcApplication` models needs to be added to the generator. As the application language offers a simple language, no extensive usage of the symbol table mechanisms of MontiCore was required. Instead, the generated AST visitor of the MontiCore grammar was used. Each ASTNode was handled in a similar fashion to a pretty printer by converting each command into related C++ methods to achieve the desired behaviour. Thus the generator does not need to be explained in greater detail. Information about the created C++ backend is given in the following Section 6.4.

6.4. C++ Backend (F)

This section gives a brief overview of the `C++`-backend's source files with their functions and relations:

main.cpp: The `main.cpp` file is the entrance point of the application. It deals with the initialization of `SDL` and `GLEW` (*OpenGL Extension Wrangler Library*) that are used by the system for window creation and rendering images and sets default values for window size, title, etc. The file also handles the program loop using the `Renderer`, `InputSystem` and `HelperAPP` classes.

HelperAPP.cpp/.hpp: The `HelperAPP` class provides static access to many functions, especially to simplify the code generation of the model, for example getting/setting the positions of sprites, dis-/enabling rendering of specific sprites, managing the stage system but also changing window attributes or requesting keyboard key states. It is thereby intended to be used as global access point for manipulating the backend. Specific stages, sprites and textures can be statically accessed by adding them to the internal lists, for example with `HelperAPP::addSprite(Square2D* sprite)`, and later-on referring to the respective given names.

TextureLoader.cpp/.hpp: The `TextureLoader` class provides the static method `Texture* TextureLoader::LoadTexture(std::string path)`. It requires a path to an image file on the disk as argument and returns a `Texture` pointer that, for example, can be added to a sprite using the provided methods of the `HelperAPP` class.

Texture.cpp/.hpp: The `Texture` class is used to manage textures on the graphics card as objects in `C++`. The provided methods for uploading image data to the graphics card and binding the textures in `OpenGL` are used by the `TextureLoader` and rendering methods.

Entity.cpp/.hpp: Render entities are used to manage render objects. The class provides all methods necessary for rendering, positioning and scaling. Each entity needs to have a name, a model (in form of vertex data) and one or more textures to be able to be rendered. The adding of textures can either be done directly using the given methods or using the static `HelperAPP` methods using the name, in case the entity is added to the static object `HelperAPP`. It is also necessary to assign a shader that is going to be used to render the specific entity. Visibility of each entity can be managed with a boolean attribute. The position coordinates are given as pixels with their origin in the upper-left border of the screen. Additionally, the texture switching of sprites is implemented in the `Entity` class. As the class is implemented as abstract class it is intended to be extended (see `Square2D`) and provides virtual methods to be adapted to the used shaders. A possible extension of the backend to 3-dimensional graphics would require adapted entities and shaders, while all necessary rendering methods are already given.

Square2D.cpp/.hpp: `Square2D` extends the class `Entity`. The class overwrites the rendering methods and adds an attribute called `UISize` to control the size of a rendering element in pixels. The scaling method is adapted to adjust the scale depending on the `UISize`. When using a `Square2D` object, no vertex model needs to be uploaded as it provides a hard-coded square model by itself.

Stage.cpp/.hpp: Each render stage bundles the entities added to it together to enable simple grouping. Stages can be identified with a name and also individually be activated and deactivated using a boolean attribute. The staging system can completely be managed using the `HelperAPP`'s functions.

Renderer.cpp/.hpp: The `Renderer` class is used in the program loop of `main.cpp` and provides methods that have to be used before and after the rendering steps.

Shader.cpp/.hpp: The `Shader` class enables object-oriented dealing with shaders on the graphics card. It features a few standard OpenGL shader implementations as static string variables but also allows loading shader code from text files. Methods to upload shader-relevant data while rendering are also given but only for currently used features of the standard implementations.

InputSystem.cpp/.hpp: The `InputSystem` wraps SDL's API for keyboard and mouse input into simple functions. The methods return boolean values whether specific keys, given as argument, are currently pressed.

MathHelper.cpp/.hpp: At last, the `MathHelper` class features interpolation and normalization methods that are used to calculate the correct rendering positions of the pixel-based positions given by `Square2D`-objects.

7. Results (F, S)

Using the EmbeddedMontiArc language family and its newly introduced member the EmbeddedMontiArcApplication language a simple Pac-Man simulation was created. This simulation as well as possible improvements for the Application language are explained in this section.

7.1. Application Output

Figure 3 presents an execution step of the Pac-Man simulation. Contrary to an ordinary Pac-Man game not one yellow Pac-Man is controlled by a player playing against four ghosts but against one ghost now. That ghost is also controlled

by another player and not autonomously. You can also see the walls and pacdots which are typical for a Pac-Man level. The control of both entities could also be changed to be not handled by a user but an autonomous controller if desired [24].

In Figure 4 the positions of the ghost and Pac-Man changed. Based on the missing pacdots the taken path of the entities can be seen. Note that the player which controls the ghost is also able to collect pacdots. As this work focused on creating a simple visualization and application language suitable for creating simple two-dimensional simulations no advanced Pac-Man features like pacpallets which enable the Pac-Man to eat ghosts were added. However, the model can be modified to add these features if desired, as this is just logical behaviour. This logical behaviour, the rendering and removal of sprites are the only important features required to accomplish this task. Thus, it has been shown that EmbeddedMontiArc language family is now in possession of these capabilities.

Download executable: `http://www.se-rwth.de/~vonwenckstern/PacMan_EMA_Application.zip`

Figure 3. The application is shown shortly after Pac-Man and the ghost have spawned.

7.2. Possible Improvements

After using the created language to implement a simple Pac-Man like world, it became obvious that different additional requirements would facilitate the usage of the application like using audio cues in addition to visual cues. Additionally, we noticed that rotating of textures is missing in our requirements as it might be useful to simplify code and textures. When executing the simulation one notices that the Pac-Man is animated but does not turn into the direction it is moving to. This can be simulated in the logics by adding rotated textures for each direction but is linked to more effort in both, texturing and logics.

When looking at the Pac-Man world that was encoded into an actuator it bothered us that this method of representing the visual world is redundant to the representation of the logical world. If advanced control structures like loops were also available in the application language, the logical representation could have been directly fed into the application actuator as variables and reused. Alternatively, from a different perspective, the whole world data could have been stored in a different file and loaded from the hard drive. For this, the reading and writing of files in the application would have been needed. In combination with advanced control structures, the data could have been loaded from that file and in the next step outputted to another actuator that deals with the rendering.

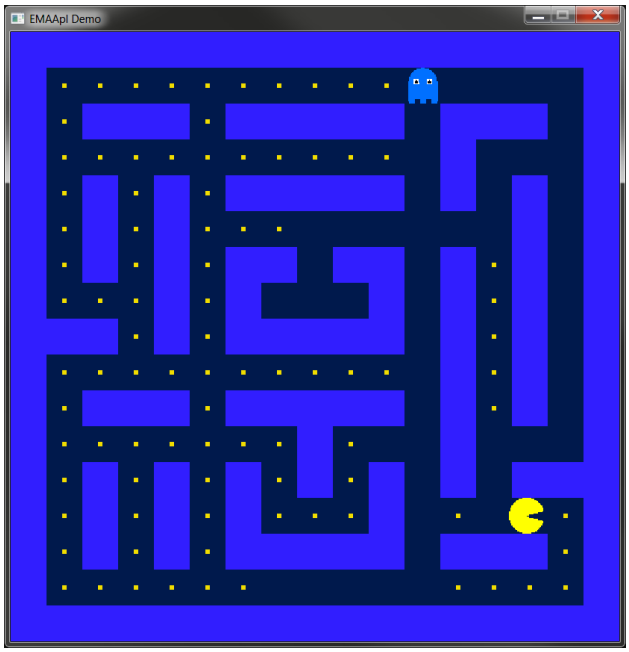

Figure 4. The application is shown after Pac-Man and the ghost have moved for some time around on the map.

8. Related Work (S)

As the EmbeddedMontiArc language family is a relatively new *C&C* modelling language family, there exist no publications yet, which describe the extension of it. However, similar modelling languages and their toolchains in regard to visualization and interactiveness can be used for comparison. There is support for the open standards file format Xtensible 3D (X3D)[25] that is available for the visual modelling language and tool Simulink, which is the successor to the Virtual Reality Modeling Language (VRML)[26]. X3D, as well as VRML are used to describe two- or three-dimensional scenes that can be displayed. This allows the visualization of running simulations in Simulink. However, this approach does not provide a standard solution for modifying the configuration of the running simulation based on user input that is directed to the visualization window of the application. Therefore, X3D or VRML are not sufficient to create interactive simulations. [27] presents an approach on how interactive simulations can be used to increase the users' understanding and interest in exploring scientific or non-scientific interactive simulations. This underlines the benefits and therefore also the importance of supporting the visualization of simulations that can be interacted with in modelling language environments, while developing concrete *C&C* models.

9. Conclusion (F)

In this paper we presented `EmbeddedMontiArcApplication` as extension of the `EmbeddedMontiArc` language family for support of creating simple 2-dimensional visual applications and stated requirements necessary for creating these. We extended `EmbeddedMontiArcMath` by a language for describing behaviour of actuators, called `Application`, that satisfies given requirements. Furthermore we presented our implementation of a simple Pac-Man variant as evaluation example together with the necessary adaptions of the language family's code backend. Additionally, the C++ code backend, that reuses OpenGL and SDL for visualization, that is incorporated by the code generator is briefly explained. Finally, we presented the executable application that resulted from utilizing the presented model and the code generator. We concluded that all gathered requirements are fulfilled but not sufficient to create a Pac-Man-like simulation easily as rotation is still missing. Nevertheless, as this is normal in agile development, the development of the next iteration of the Application language will start with adding the rotation requirement.

However, we discovered the drawback that the simple application language without advanced control mechanisms like loops does not facilitate reuse of existing structures like the logical representation of the Pac-Man world in EmbeddedMontiArcMath. Furthermore, additional cues like sound effects might improve immersiveness of possible simulations.

References

[1] Mathworks, Simulink User's Guide, Tech. Rep. R2016b, MATLAB & SIMULINK (2016).
[2] Mathworks, Simulink unitconversion, accessed: 2018-08-23 (2018).
URL https://de.mathworks.com/help/simulink/slref/unitconversion.html
[3] E. Kusmenko, A. Roth, B. Rumpe, M. von Wenckstern, Modeling architectures of cyber-physical systems, in: European Conference on Modelling Foundations and Applications, Springer, 2017, pp. 34–50.
[4] B. Rumpe, K. Hölldobler, MontiCore 5 Language Workbench, RWTH Aachen University Software Engineering Group, edition 2017 (2017).
[5] T. Parr, The definitive ANTLR 4 reference, Pragmatic Bookshelf, 2013.
[6] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, Ros: an open-source robot operating system, in: ICRA workshop on open source software, Vol. 3, Kobe, Japan, 2009, p. 5.
[7] C. Berger, An open continuous deployment infrastructure for a self-driving vehicle ecosystem, in: IFIP International Conference on Open Source Systems, Springer, 2016, pp. 177–183.
[8] J. O. Ringert, Analysis and synthesis of interactive component and connector systems, Ph.D. thesis, RWTH Aachen University (2014).
[9] MATLAB, version 7.10.0 (R2010a), The MathWorks Inc., Natick, Massachusetts, 2010.
[10] E. Kusmenko, B. Rumpe, S. Schneiders, M. von Wenckstern, Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc, in: Conference on Model Driven Engineering Languages and Systems (MODELS'18), IEEE, 2018, pp. 167–177.
[11] K. GROUP, Opengl core specification, accessed: 2018-05-24 (2018).
URL https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf
[12] S. Lantinga, et al., libsdl: Simple directmedia layer, U RL http://www. libsdl. org.
[13] B. Kernighan, D. M. Ritchie, The C programming language, Prentice hall, 2017.
[14] D. Team, Dosbox, accessed: 2018-05-25 (2018).
URL http://www.dosbox.com/
[15] V. Team, Visual boy advance, accessed: 2018-05-25 (2018).
URL https://sourceforge.net/projects/vba/
[16] R. Brewster, Paint .net-free software for digital photo editing, Getpaint. net. Ultimo acceso 15.
[17] G. Gimp, Image manipulation program, User Manual, Edge-Detect Filters, Sobel, The GIMP Documentation Team 8 (2) (2008) 8–7.
[18] N. Limited, Pac-man, accessed: 2018-08-23 (1980).
URL pacman.com
[19] E. Nintendo, Super mario bros. (1985), accessed: 2018-08-23.
URL http://mario.nintendo.com/history/
[20] B. Wu, The computational intelligence of the game pac-man, in: Internet of Things, Springer, 2012, pp. 646–651.
[21] M. Gallagher, A. Ryan, Learning to play pac-man: An evolutionary, rule-based approach, in: Evolutionary Computation, 2003. CEC'03. The 2003 Congress on, Vol. 4, IEEE, 2003, pp. 2462–2469.
[22] K. Petersen, C. Wohlin, D. Baca, The waterfall model in large-scale development, in: International Conference on Product-Focused Software Process Improvement, Springer, 2009, pp. 386–400.
[23] B. Rumpe, Formale Methodik des Entwurfs verteilter objektorientierter Systeme, Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.
URL http://www.se-rwth.de/~rumpe/publications/Formale-Methodik-des-Entwurfs-verteilter-objektorientierter-Systeme-Disser
pdf
[24] J. P. Haller, M. Heithoff, A case study of the component and connector modeling language embeddedmontiarc (2018).
[25] L. Daly, D. Brutzman, X3d: Extensible 3d graphics standard [standards in a nutshell], IEEE Signal Processing Magazine 24 (6) (2007) 130–135.
[26] R. Carey, G. Bell, The annotated VRML 2.0 reference manual, Vol. 15, Addison-Wesley Reading, MA, 1997.
[27] A. D. Baldwin, H. Elmqvist, S. Dahlberg, 3d schematics of modelica models and gamification, in: Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015, no. 118, Linköping University Electronic Press, 2015, pp. 527–536.