SCCharts
For Game Development

Philip Raschkowski

Bachelor's Thesis

March 2021
Real-Time and Embedded Systems

Prof. Dr. Reinhard von Hanxleden
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Reinhard von Hanxleden

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

Bei der Spieleentwicklung gibt es eine Vielzahl verschiedener Anwendungen von Zustandsautomaten, aber die eingebauten Lösungen leiden unter einer Reihe von Einschränkungen. Diese Einschränkungen bestehen entweder in Form von Funktionalität oder dadurch, dass sie nur für bestimmte Aufgaben anwendbar sind. Eine fehlende Implementierung verlangsamt den Entwicklungsprozess, da der Entwickler nur einen begrenzten Werkzeugsatz verwenden kann oder das gewünschte Verhalten überkompliziert beschreiben muss. Das größere Problem ist jedoch, dass diese Implementierungen meist nur für einen Anwendungsfall konzipiert sind. Dies lässt die Entwickler im Wesentlichen auf sich allein gestellt, wenn sie Zustandsautomaten für andere als die vorgesehenen Anwendungsfälle nutzen wollen.

In dieser Arbeit soll dieses Problem durch die Einführung einer synchronen Sprache namens SCCharts behoben werden. Zudem wird untersucht, wie gut diese Sprache in den Arbeitsablauf der Spieleentwicklung integriert werden kann. Die Sprache wurde entwickelt, um deterministische endliche Zustandsautomaten für sicherheitskritische Anwendungen zu erstellen und kommt zudem mit vielen eingebauten Features, die die Entwicklung erleichtern. SCCharts ist geeignet, um komplexe Automaten zu modellieren, die nicht nur einfach zu lesen, sondern auch zu simulieren sind, noch bevor sie in ihre Umgebung integriert werden. Wenn dieser Ansatz übernommen wird, könnte dies zu Spielen führen, die stabiler sind, weniger Bugs aufweisen und einfacher zu warten sind.

Abstract

Game development makes use of a multitude of different applications of state machines, but the built-in solutions suffer from a couple of limitations. These limitations are either in form of functionality or by being only applicable to specific tasks. An implementation that lacks features slows down the development process, since the developer can only use a limited toolset or must describe the desired behavior in an overcomplicated fashion. However, the bigger issue is that, most of the time, these implementations are designed for only one specific use case in mind. Most of the time the engine specific implementations are designed with a single task in mind. This essentially leaves the developers on their own if they want to use state machines for other use cases than the ones provided.

This thesis aims to alleviate this issue by considering a synchronous language named SCCharts and tries to investigate, how well this language can be integrated into the game development workflow. The language is designed to create deterministic finite state machines for safety-critical applications and comes with many built-in features, which ease development. As such, the language is suitable to model complex automata, that are not only easy to read, but also to simulate, even before integration them into their environment. If adopted, this approach could lead to games that are more stable, include fewer bugs, and are easier to maintain.

Acknowledgements

I would like to thank my advisor Prof. Dr. Reinhard von Hanxleden who not only supported me by providing feedback about this thesis, but also introduced me to SCCharts in the first place. Also, I would like to say thank you to Alexander Schulz-Rosengarten, who provided me with additional information especially in regard to SCCharts while writing this thesis and long before. I am also grateful for Kay, a dear friend of mine, who vigorously proofread my writing.

Lastly, also place on record, my sense of gratitude to one and all, who directly or indirectly, have supported me in this venture.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

There are many aspects of game development that act according to specific states. Consider something simple like a door. A door can be open or shut or even locked. If it is locked, it cannot be opened and if it is open, it cannot be locked. This can easily be written in simple code since it is an uncomplicated example. However, with the help of a finite state machine, FSM for short, the structure is more obvious and the states can be well defined. A state machine can also be visualized to increase readability and therefore maintainability. Additionally, it is easier to come back later and add additional states or functionality to the FSM than it is to rewrite the code of a more or less hardcoded behavior not split into distinct states. The choice of whether or not to use a finite state machine becomes clearer as the implementations get more complex. That is why most game engines feature at least one implementation of a state machine. Unreal Engine 4 (further denoted as UE4), for example, features an excellent implementation of such an FSM, however, this is limited to 3D skeletal animations. Godot, on the other hand, offers a state machine implementation that can be used for anything, but whose feature set is relatively limited. It has only a very basic implementation of an FSM, which contrary to the UE4 implementation can be used for anything.

The goal of this thesis is to evaluate the integration of a domain-specific language for FSMs in game development. The considered language will be SCCharts, a synchronous language mainly used for embedded systems that can be compiled to C code, among other languages, which can be integrated into most engines as external libraries. The language is textual but features a graphical representation of the written code live with an automated layout as well as other features, such as the ability to build hierarchical FSMs and model verification.

1.2 Related Work

The need for a solid state machine implementation to use in games becomes apparent when looking at different game engines and their respective implementation. Firstly, there is Unity's implementation of state machines [Uni21]. Their approach is mainly meant for animating 2D/3D animations but is, however, not strictly limited to this application. The existence of numerous guides and wikis which describe how to implement an FSM besides the one delivered by a game engine implies that another kind of tooling is needed. For example, the guide [Sho20] and the wiki [Unk12] describe how to implement an FSM in Unity.

However, this is not unique to Unity. Developers encounter similar limitations with other engines. Godot for example also brings its own implementations of a state machine called *AnimationTree* [LMG20]. However, this again comes with its own limitations and restrictions, mainly defined by functionality. The *AnimationTree* should be able to *animate pretty much any property in any node or resource* [LMG20], which would be more flexible than the Unity implementation. That is of course if their claim holds that they have the *most flexible animation systems that you can find in any game engine* [LMG20]. Still, there are many websites, blogs and videos that can easily be found with a quick google search that highlights that this is simply not enough.

Lastly, there is the Unreal Engines implementation of state machines. Their system is very sophisticated, integrated and tailored for a specific use case, which limits its use to 3D animation [Epic]. They also feature another implementation, namely *Behavior Trees* [Epia], but they are also tailored to be used in the development of AI behavior. Epic, the developer of the Unreal Engine, also acknowledges the need for another kind of FSM by producing a live training tutorial that is available on YouTube [Epib]. There are also a couple of implementations available on the Unreal market place, e.g. [Rec19], [L16]. There also exists an FSM designed to be very close to the original UE4 state machine but for 2D animation [Stu], which is a recipient of an Epic MegaGrant, which is Epics program of funding projects of any kind developed with or for the Unreal Engine.

1.3 Finite State Machines

1.3.1 Introduction to finite state machines

Finite state machines, also called finite automata, belong to the field of automata theory. They were originally proposed to model brain functionality, according to Hopcroft et. al., however, they were found to be extremely useful for a wide range of use cases [ED79]. Since their introduction, FSMs are used to model all kinds of different behavior. Such an automaton consists of a finite set of *states* and a set of *transitions* between two states. The transitions have a *condition* and an *action*. If the condition on a specific transition is met, the transition is taken and the correlating action is executed. This is the most basic example of an FSM, but even with this simplification, complex systems can be built which can also be visualized. Consequently, FSMs are a useful tool for the expression of behavior that can be distilled in distinct states and corresponding actions, which are performed once a transition is taken.

To demonstrate this, the simple door example from the introduction will be elaborated. The door could have the state *open* iff it is open, and *closed* iff it is not open. Accordingly, one would define transitions from *open* to *closed* if the door opens and, in the context of a game, the engine could play the opening animation as the action of the transition. A more complex but still simple example, considering what is used in modern games, would be the sprite animation control of a moving 2D character. One possible implementation for this is shown in Figure 1.1, which was created using Unity.

Even though it is still rather elementary, it gets hard to read and therefore to build upon or maintain. Figure 1.2 shows the same behavior as in Figure 1.1, this time with SCCharts. At first glance, this could look a bit overwhelming. Since a hierarchical approach is used, the nested diagram in Figure 1.3 and the top-level diagram in Figure 1.4 can be viewed separately. This allows us to get a better understanding of what is going on in a more complex FSM compared to one big chart. Also, this illustrates the use of a divide and conquer approach, since first a smaller problem for one cardinal direction only is solved. In this specific case Figure 1.3 is, in essence, the core of the chart in Figure 1.2. It describes the state change for one direction and can be reused for all other cardinal directions. That means that the FSM one needs to focus on is smaller, since the underlying functionality is abstracted. This can lead to a significant increase in productivity that increases further as models get more and more complicated. Furthermore, the developer does not need to worry about the layout and spend valuable time arranging the FSM or click through a GUI. The reason for this is the automated layouting that makes SCCharts easy to read without the need for manual interference. For example, the automaton in Figure 1.2 is automatically created by compiling the lines of code in Listings 1.1 and 1.2, which both reside in the same file. This enables to reference the SCChart directly without the need to import an external SCChart, which can also be done and is described in the SCCharts syntax documentation [SS20].

Further, it is also possible to blend out the nested FSM to first get a more abstract view on the general behavior of this, as seen in Figure 1.4

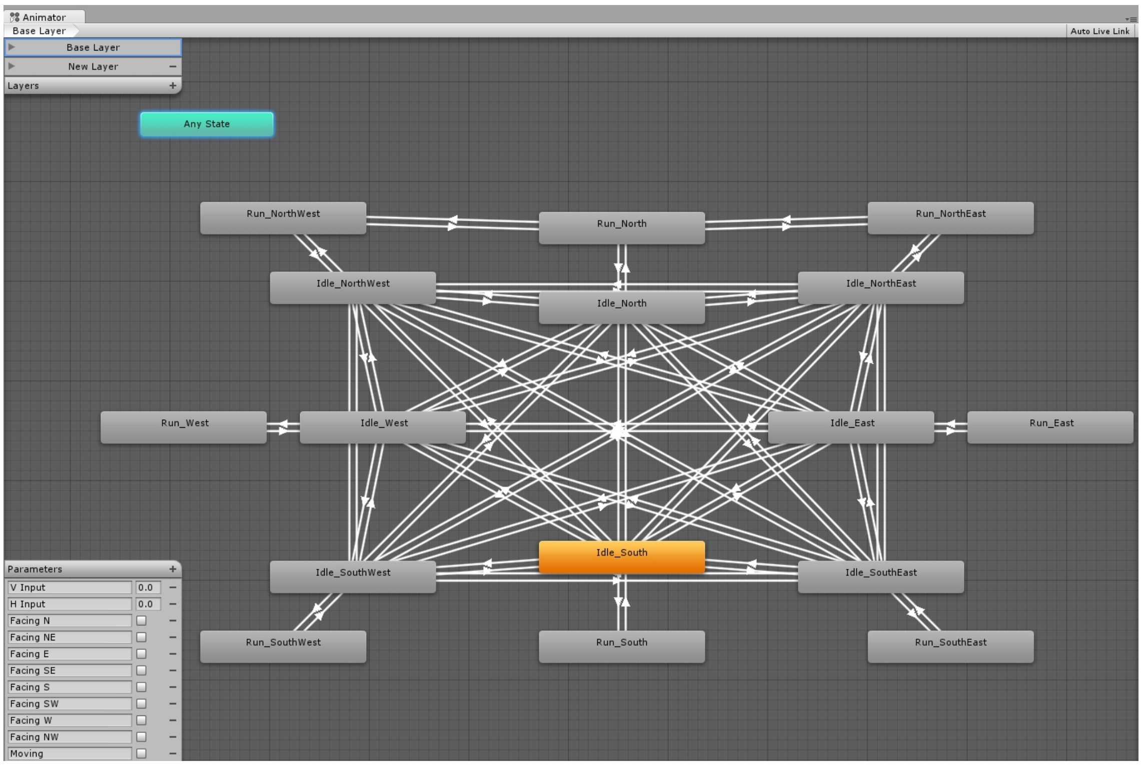
1.3.2 Use Cases

Typical use cases, as hinted in Section 1.2, include but are not limited to animations of any kind and AI behavior. Apart from that, basically anything that can be described using states can be designed with an FSM in a controlled and predictable way. Since most implementations of FSMs are deterministic, meaning that all events and actions are well-defined at any point in time, the stability of the whole system increases. Unwanted behavior could also be expressed with the help of model checking [Sta19] with linear temporal logic.

1.4 FSM inside Unreal Engine 4

The UE4, as well as most modern game engines, supports at least one type of state machine. However, these usually either lack features, as in Unity or Godot, or are limited to specific use cases like in UE4, or both. UE4 has a matured state machine with visual simulation and debugging well-integrated in the engine. These state machines are limited to skeletal mesh animation [Epic]. The engine also has an implementation for AI, as previously mentioned, namely the *Behavior Tree*. The *Behavior Tree* can in essence also accomplish similar tasks as an FSM in some manner but is much better suited for the use in an AI.

This leaves the developer essentially with no built-in solution for an FSM outside of the typical use cases. This means the developer has to either create one themselves, buy one from



V Input	0.0
H Input	0.0
Facing N	☐
Facing NE	☐
Facing E	☐
Facing SE	☐
Facing S	☐
Facing SW	☐
Facing W	☐
Facing NW	☐
Moving	☐

Figure 1.1. 2D character FSM to handle cardinal sprite direction made in Unity

the market or abstain from the use of an FSM altogether. The UE4 team at Epic acknowledges this in some way by providing an example of how to write an FSM in C++ for UE4 in a live training session published on YouTube [Epib].

1.5 Hybrid systems

Besides state machines, SCCharts can also handle hybrid systems like PID controllers. These are control loop mechanism, which are used in the real world for example in cruise mode for cars for acceleration and deacceleration control or to supervise the regulation of heater. In games, these can be used to control similar matter, but there are many other implementation possibilities, like *Quadrotor Simulator* [San17], which is also implemented in Unreal.

1.6 SCCharts

1.6.1 Introduction to SCCharts

SCCharts is a synchronous language developed at the Christian-Albrecht University of Kiel by the Real-Time and Embedded Systems group. The name stands for *Sequentially Constructive*

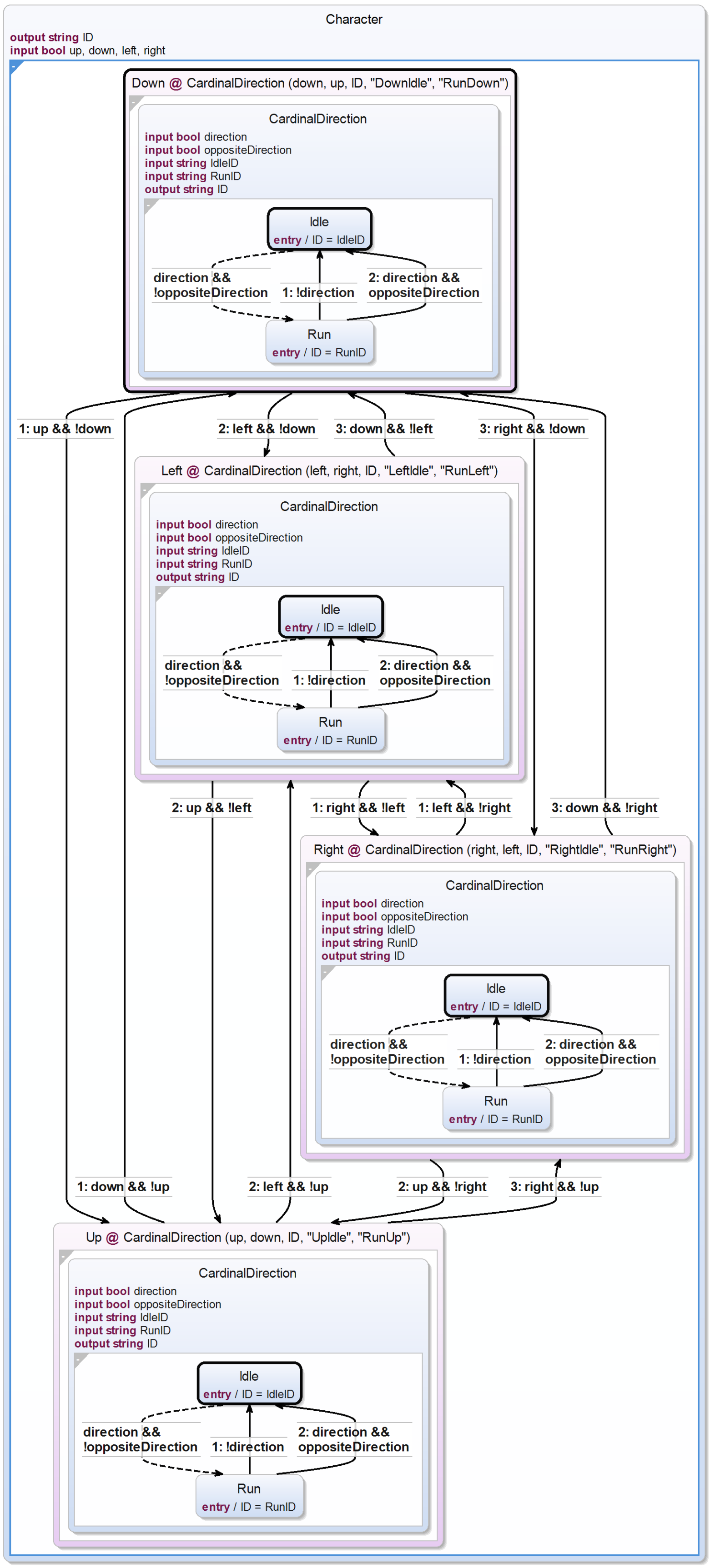

Character
output string ID
input bool up, down, left, right
Down @ CardinalDirection (down, up, ID, "DownIdle", "RunDown")
Left @ CardinalDirection (left, right, ID, "LeftIdle", "RunLeft")
Right @ CardinalDirection (right, left, ID, "RightIdle", "RunRight")
Up @ CardinalDirection (up, down, ID, "UpIdle", "RunUp")
CardinalDirection
input bool direction
input bool oppositeDirection
input string IdleID
input string RunID
output string ID
Idle
entry / ID = IdleID
Run
entry / ID = RunID
direction && !oppositeDirection
1: !direction
2: direction && oppositeDirection
1: up && !down
2: left && !down
3: down && !left
3: right && !down
2: up && !left
1: right && !left
1: left && !right
3: down && !right
1: down && !up
2: left && !up
2: up && !right
3: right && !up

Figure 1.2. The same FSM as in Figure 1.1 built with SCCharts

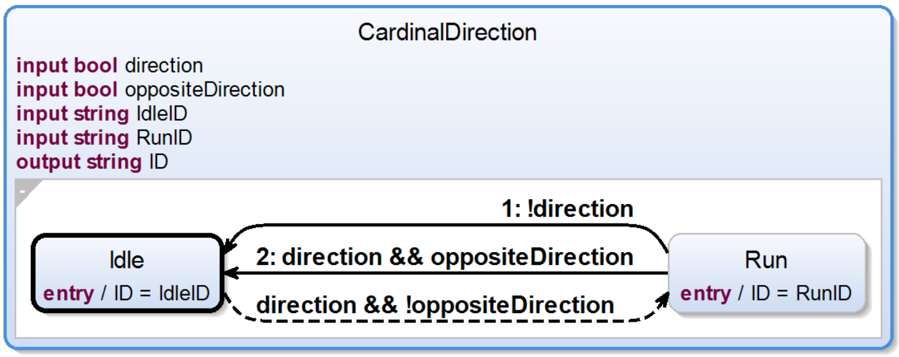


Figure 1.3. One cardinal direction of Figure 1.2

Statecharts and has its main footing in the world of embedded real-time systems. According to von Hanxleden et. al, the language is designed with safety-critical applications as well as easy adaptation in mind [HDM+14]. With this language, it is possible to build deterministic automata, which even operate concurrently, without introducing any race conditions.

Furthermore, due to the rich features as well as the graphical visualization, automated graph drawing, and the ability to compile to different languages like Java or C. As such, SCCharts is not only limited to embedded real-time systems, but can effectively be used wherever an FSM or hybrid system is needed. Since game development has many areas where FSMs are extremely useful, SCCharts can increase maintainability, as well as improve the debugging process and the overall workflow. SCCharts comes with many features like automated graph drawing, different types of transitions as well as conflict evaluation at compile-time and a visual simulation. Especially the simulation becomes increasingly handy while developing or debugging more complex systems. The interested reader who wants to learn more about SCCharts is referred to [SL20].

1.6.2 Advantages of SCCharts

As stated above SCCharts comes with many useful features that ease development, reduce the risks of errors and can therefore increase productivity. In addition to those already mentioned, these functions include various types of transitions that help build sophisticated functionality with just a few keywords. It is also possible to build nested SCCharts by referencing other SCCharts and therefore to build a hierarchy, which follows the Write-Things-Once (WTO) principle, which further increases maintainability and productivity.

Furthermore, the KIELER IDE, designed for the use with SCCharts, automatically generates graphs that facilitate understanding of the functionality of the design. This opens up the possibility even for developers who are not used to code to understand the automata and what is happening by simply looking at the diagrams instead of the code. Another very convenient tool is the simulation in conjunction with the highlighting of current and previous states to test the design even before integrating it into the engine or any other environment. More about SCCharts' syntax can be found in the respective documentation [SS20].

```
scchart Character {

    output string ID
    input bool up,down,left,right

    region {
        initial state Down is CardinalDirection(    down to direction,
                                                    up to oppositeDirection,
                                                    ID to ID,
                                                    "DownIdle" to IdleID,
                                                    "RunDown" to RunID)
        if up && !down go to Up
        if left && !down go to Left
        if right && !down go to Right

        state Left is CardinalDirection(    left to direction,
                                            right to oppositeDirection,
                                            ID to ID,
                                            "LeftIdle" to IdleID,
                                            "RunLeft" to RunID)
        if right && !left go to Right
        if up && !left go to Up
        if down && !left go to Down


        state Right is CardinalDirection(   right to direction,
                                            left to oppositeDirection,
                                            ID to ID, "RightIdle" to IdleID,
                                            "RunRight" to RunID)
        if left && !right go to Left
        if up && !right go to Up
        if down && !right go to Down

        state Up is CardinalDirection(  up to direction,
                                        down to oppositeDirection,
                                        ID to ID,
                                        "UpIdle" to IdleID,
                                        "RunUp" to RunID)
        if down && !up go to Down
        if left && !up go to Left
        if right && !up go to Right
    }
}
```

Listing 1.1. The 2D character SCCharts source code of the graph in Figure 1.2

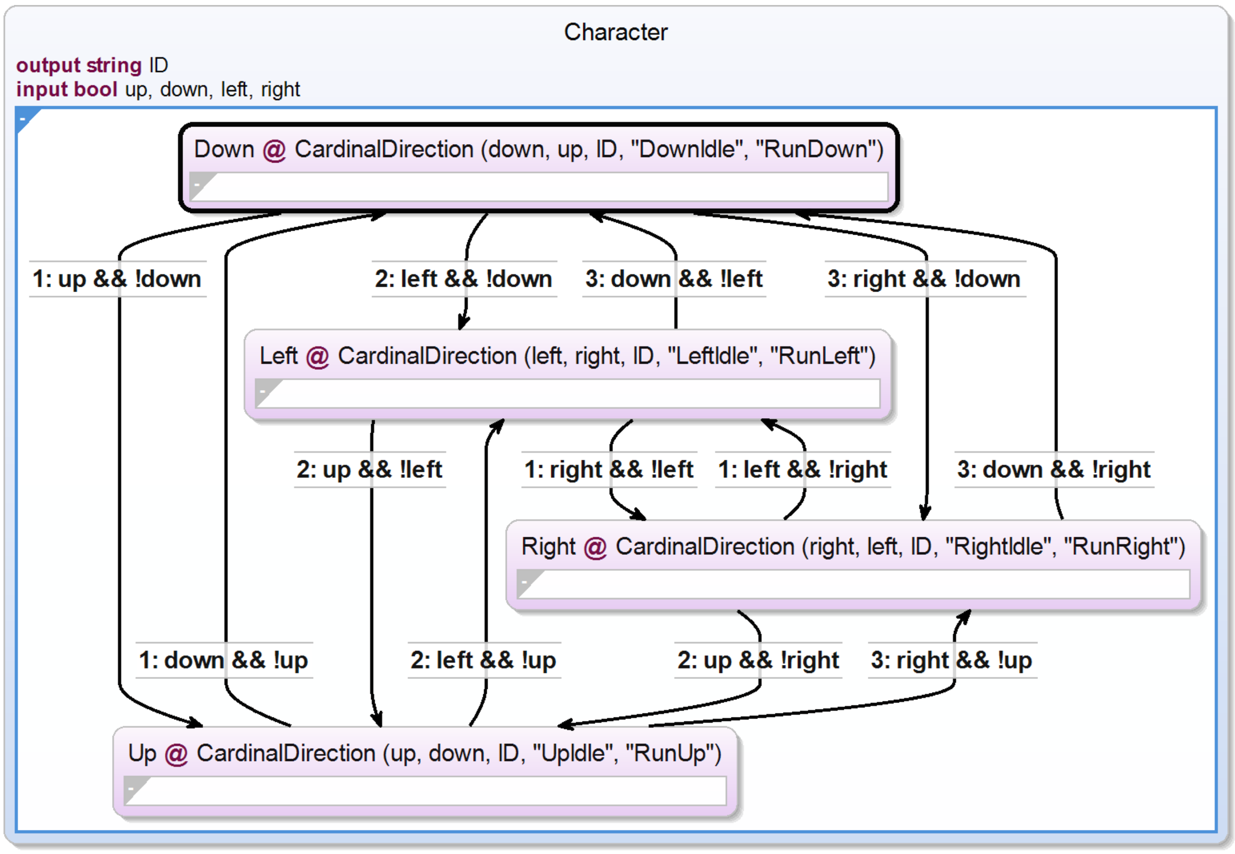

Character
output string ID
input bool up, down, left, right
Down @ CardinalDirection (down, up, ID, "DownIdle", "RunDown")
1: up && !down
2: left && !down
3: down && !left
3: right && !down
Left @ CardinalDirection (left, right, ID, "LeftIdle", "RunLeft")
2: up && !left
1: right && !left
1: left && !right
3: down && !right
Right @ CardinalDirection (right, left, ID, "RightIdle", "RunRight")
1: down && !up
2: left && !up
2: up && !right
3: right && !up
Up @ CardinalDirection (up, down, ID, "UpIdle", "RunUp")

Figure 1.4. The collapsed version of the chart in Figure 1.2

Additionally, it is also possible to build *timed automata* in SCCharts. According to Schulz-Rosengarten et al., *timed automata* are used to model the behavior of real-time systems over time [**lzRosengartenvHMdS+18a**]. By using these automata, it is possible to model, e.g., cruise control for a car or stabilization control for a helicopter in dependence of real-time. In SCCharts this can be accomplished by using *clocks*. These create an internal variable *deltaT*, which can be written from the environment. In the case of a game environment, one can pass the time of the last frame, also called delta time, to the SCChart *deltaT* variable. This also takes any time dilation effects which may happen in a game into account, and the automaton act accordingly.

1.6.3 External Library

For the SCCharts implementation approach, external libraries have been shown to be the most flexible solution. As stated in the Microsoft documentation: *The use of DLLs helps promote modularization of code, code reuse, efficient memory usage, and reduced disk space. So, the operating system and the programs load faster, run faster, and take less disk space on the computer.* [Mic20] This

```
scchart CardinalDirection {

    input bool direction
    input bool oppositeDirection
    input string IdleID
    input string RunID
    output string ID

    region {
        initial state Idle {
            entry do ID = IdleID
        }
        immediate if direction && !oppositeDirection go to Run

        state Run {
            entry do ID = RunID
        }
        if !direction go to Idle
        if direction && oppositeDirection go to Idle
    }
}
```

Listing 1.2. The CardinalDirection source code referenced in Listing 1.1

holds true not only for Microsoft's dynamic-link libraries (DLLs) but also for the Linux or iOS libraries.

This means, even though this example is demonstrated with UE4 in mind, the SCCharts libraries are essentially engine independent and could not only be used in different UE4 projects but also in different game engines or applications altogether.

Furthermore, most game engines support the use of linked libraries. In fact, they suggest it as the general approach to include third-party code.

1.6.4 Integration into Unreal Engine

For this thesis, the Unreal Engine 4.26 has been chosen as an example engine, mainly it comes with a build tool that automatically compiles the C code generated by SCCharts to external libraries for the most common architectures, which eased development. However, this example can be implemented in any engine that features the use of external libraries like Unity or Godot.

To be compliant with the UE4 specifications the external libraries are implemented via plugins as described.

In UE4 external libraries are implemented via plugins, which follow the UE4 specifications. A template for the UE4 Plugin with a minimal SCCharts for personal or commercial can be found in the provided repository [Ras21b].

Further documentation about plugins in UE4 can be found in the documentations provided by Epic [Epid].

Chapter 2

Limitations

Since we make use of external libraries such as DLLs, which are only used for Windows, the code must be compiled for all desired operating systems unless this is handled by the engine itself, which is the case for UE4. Moreover, the setup can create some overhead and should be used with its own IDE. Furthermore, it could be possible that the developer needs to manage the external libraries themselves. This means not only to compile the libraries but also to manage instances since by design an instance of an external library shares its memory space. In other words for a state machine built like described in this thesis a separate external library instance for each separate Actor in the game is needed. Fortunately, this is handled automatically by most modern game engines like Unity, Godot and Unreal.
It is also possible that the engine, or more specifically the programming language that is used by it, is not capable of handling the debugging with an attached debugger of external libraries. This could increase difficulties when trying to find problems which is a general issue when using external libraries. The aforementioned obstacle is further compounded by the fact that the C code created by the SCCharts compiler is generated and therefore most of the times not as easy to read as code written by humans. The readability in the case of the generated code depends on the chosen compilation system for the SCCharts compiler. The *state-based compilation* creates the code that is easiest to read for humans compared to other code generation options.

It is also not possible unless with extensive overhead and C and C++ knowledge to call functions on objects passed directly to the SCChart. This means in most cases a wrapper is needed to integrate the SCCharts with the host code base. The wrapper not only needs to call the functions on the desired objects but also needs to manage the state returned in one way or another from the SCCharts compiled code.

Chapter 3

Implementation

3.1 Requirements

Before one can start with the development of SCCharts a couple of things are needed in order to allow a smooth workflow. To develop SCCharts it is possible to download either the KIELER IDE or KEITH. KEITH is an IDE for Model-Driven Development which is based on KIELER [SD]. However, KEITH does not yet come with all the features present in the KIELER IDE yet. The Kiel Integrated Environment for Layout Eclipse Rich Client, or short KIELER, is a research project about enhancing the graphical model-based design of complex systems, according to the website [SSa]. As stated, KIELER supports more features at the time of writing, which is why it was chosen for this thesis instead of KEITH. It is also of advantage to download the Kieler Command Line Compiler [**Kico**], or KiCo CLI for short, since this allows a more comfortable compilation process. Lastly, some kind of game development environment is needed, preferably an IDE. For simple games this could of course be ignored. As a game engine as stated above UE4 was chosen for this task, with version 4.26.

3.2 Plugin Template (Unreal)

To get SCCharts inside UE4 working, one needs to take a look into UE4's approach for including non-native code. Non-native code can be any code that is itself not written inside the Unreal environment. This code can for example be C or C++ code. As it turns out, SCCharts can compile down to C code, which can easily be integrated inside UE4 since the engine is based on C++ itself. This is not a hard requirement, however, since external libraries are used which should work just as fine in any game engine.

3.3 General Workflow

The general workflow for using SCCharts inside UE4 starts by finding an applicable problem where the use of FSMs or hybrid systems makes sense. Thereafter, a plugin is needed for the SCCharts integration. After the plugin has been properly created the next step in line is to write the SCChart inside KIELER. Since SCCharts uses automated graph drawing the layout is handled automatically and can be calibrated to one's liking. The automated layout offers an accessible overview of the flow inside a chart without the need to position the states and transitions by hand. This also reduces development time especially if changes are in order later in the process. After the chart is completed, one can start the KIELER SCCharts simulation to verify that the implementation satisfies the requirements and make necessary adjustments. It is also possible to further visualize the simulation [SSb] besides the visualization in KIELER using Scalable Vector Graphic (SVG) images. This, however, is something that is not further discussed in this thesis.

Subsequently, the SCChart needs to be compiled to one of the supported target languages. To accomplish the compilation in a way that is convenient for the developer in case further changes are in order the *KIELER Compiler*, for short KiCo, can be used. With KiCo it is possible to compile the charts with a command-line interface (CLI) or via a script and hooks, to automate this process. As of the time of writing the KiCo does not come with its own documentation, besides the `--help` command inside the CLI. However, the KiCo CLI should be simple enough to be used without any further need for documentation besides that.

After the successful compilation, a wrapper class is needed inside UE4 to interact with the code in a way that compiles to C++.

From here on it is possible to go back and forth between the UE4 development environment and the SCCharts development. The code gets automatically compiled into the UE4 Plugin folder. Only if significant changes are made to the chart an update to the UE4 wrapper class is needed. Significant changes that force a change to the wrapper class include but are not limited to adding or removing input and/or output variables inside the SCChart that UE4 should interact with. Nested SCCharts that are not directly accessed via the wrapper class should not press for changes to the wrapper.

3.4 Plugin Creation (UE4)

To start using SCCharts as discussed in Section 3.3, a project needs to be opened and a plugin added to it. For this thesis the choose the *Blank Plugin* has been chosen for the implementation in this thesis. Documentation about plugins and how to create them can be found inside the Unreal Engine 4 documentation [Epid].

For this thesis, a repository with a variety of project files including a blank project with a blank plugin created inside it containing the basic files linked to the open project is provided.

To be able to call functions from the compiled SCCharts, Unreal needs to know what to expose so it can be accessed the way UE4 intends it. For this, the build file needs to be modified as shown in Listing 3.1.

With this, the classes and functions created in the plugin can be accessed in the main project. Normally the SCCharts-generated C code must first be compiled to a library file like dynamic-linked libraries for Windows. Fortunately, UE4 has a convenient build tool that takes care of that for us. If this should be integrated inside another engine, the library files may need to be compiled manually. After this is done, the C code that is generated by the SCCharts compiler simply needs to be added to the plugin's source code. To call for example the `tick()` function, the generated header file needs to be included and the include keyword needs to be surrounded by the *extern* block. This is because otherwise the included C code

```
        PublicIncludePaths.AddRange(new string[] {
            "../Plugins/<PluginName>/Source/<PluginName>//Public"
        });
        PrivateIncludePaths.AddRange(new string[] {
            "../Plugins/<PluginName>/Source/<PluginName>/Private",
            "../Plugins/<PluginName>/Source/<PluginName>/Classes"
        });
        PrivateDependencyModuleNames.AddRange(new string[] {
            "<PluginName>"
        });
```

Listing 3.1. Modified *<myProject>.build.cs* to enable function calls

```
    extern "C" {
        #include "/<path_to>/<header_file>.h"
    }
```

Listing 3.2. Include wrapper to prevent name mangling

function names get mangled so that linkage errors can occur on compilation. An example is shown in Listing 3.2.

Additionally, a C++ class named `FirstSCChart` from inside the engine for the plugin has been created. This class acts as a wrapper that wraps the needed function nicely inside the plugin source code. That way one only needs to call the created C++ functions outside of the SCCharts plugin.

3.5 The first SCChart inside Unreal

For the sake of getting started with SCCharts inside UE4, a demonstration is given on how to implement a small SCChart shown in Figure 3.1. A possible SCCharts implementation for this state machine is shown in 3.3.

This introductory FSM has only two states `A` and `B`. Every tick the automaton switches between these states and sets the `text` to `Hello`, iff it is in state `A` or to `Unreal World`, iff `B` is the current state of the machine. For initialization purposes, the start `text` is set to an empty

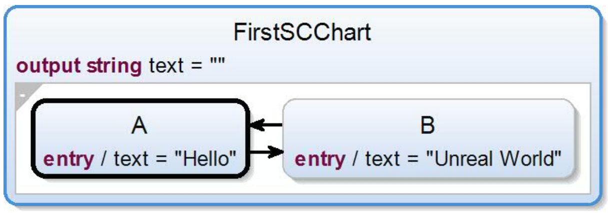


Figure 3.1. First simple SCCharts to bring to UE4

```
scchart FirstSCChart {
    output string text = ""

    region {
        initial state A {
        entry do text = "Hello"

        } go to B

        state B {
        entry do text = "Unreal␣World"
        } go to A
    }
}
```

Listing 3.3. FirstSCCharts source code

string.

After successfully setting up the project as described in Section 3.4, it is now possible to include the compiled SCCharts and call its functionality inside the engine. To compile the SCCharts to the desired plugin folder the KiCo CLI is used with a batch file as displayed in Listing 3.4.

```
call kico-win.bat Test.sctx -o <Drive>:\path\To\Plugin\Folder ^
                            -s de.cau.cs.kieler.sccharts.netlist
```

Listing 3.4. Batchfile content for compilation with KiCo

The `-o` argument specifies the output destination folder. While working on the thesis it is redirecting roughly to *path\to\project\SCChartsForGames\Plugins\FirstSCChart*. Additionally, the `-s` argument specifies the desired system that the compiler uses to compile the SCChart. The *de.cau.cs.kieler.sccharts.netlist* system has been chosen because it suffices for the needs of this thesis. A list for possible compilation systems can be shown within the KicO CLI with the command `--list-systems` for common systems only or `--list-all-systems` to list all available systems including internal systems.

For the first SCChart, an *Actor* is created from inside the engine named `AFirstSCChart`. On compilation, UE4 creates the necessary C++ files. Inside `AFirstSCChart.cpp` there is the template created by the engine. For initialization the lines in Listing 3.5 are added to the `void AFirstSCChart::BeginPlay()` function in Listing 3.5, which is called as soon the game starts and therefore before the `void AFirstSCChart::Tick(float DeltaTime)` function to reset the values of the SCChart.

`FistSCChart` is the wrapper class mentioned before. In order for the FSM to run properly the `reset()` function has to be called before any invocation of the `tick()` function. This is due to internal SCCharts specification and if it is not called future invocations of the `tick(&TickData)` function are not executed as expected.

```
// Called when the game starts or when spawned
void AFirstSCChart::BeginPlay() {
    // generated by UE for inheritance
    Super::BeginPlay();
    // The SCCharts FSM needs to be ressetted before it can receive the first tick
    SCChartsFSM.Reset();
}
```

Listing 3.5. BeginPlay function of AFirstSCChart

```
void FirstSCChartFSM::Reset() {
    reset(&tickData);
}
```

Listing 3.6. Reset function wrapper of FistSCChart

This is taken care of by the function `FistSCChart.Reset()` that initializes or resets the SCCharts inside Unreal, which is the wrapper needed to communicate with the SCCharts generated C function in Listing 3.6.

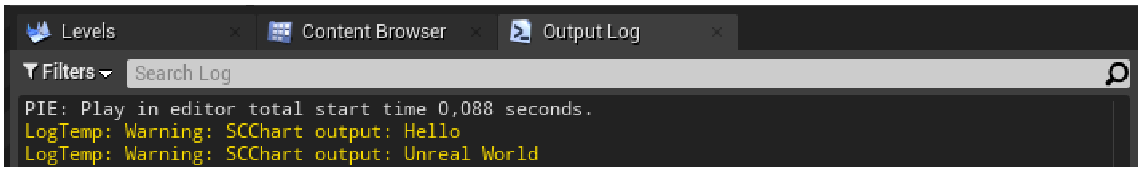
The argument `TickData` is the construct generated by the SCCharts compiler that holds the necessary values like input, output as well as all internal variables used by the chart.

To test the FSM, the function `Tick(float DeltaTime)` must be called which can be done inside the aforementioned `void AFirstSCChart::Tick(float DeltaTime)` like described in Listing 3.7

where `FistSCChart.Tick()` is a wrapper around the SCChart generated tick function in the same sense as the `FistSCChart.Reset()` wrapper was. The `UE_LLOG` macro enables us to print something to the logs so one can see if the code is working as intended. The *Warning* enum marks this text as a warning, so it is written in yellow in the logs so it can be distinguished more easily from the rest.

```
// Called every frame
void AFirstSCChart::Tick(float DeltaTime){
    Super::Tick(DeltaTime); // generated by UE for inheritance
    SCChartsFSM.Tick();
    // this prints a yellow Warning with the string inside TEXT(..) and the text received
    // from the SCCharts which is located under tickdata as are any other input or output
    // variable
    UE_LOG(
        LogTemp, Warning, TEXT("SCChart␣output:␣%s"), *FString(SCChartsFSM.tickData.text)
    );
}
```

Listing 3.7. Tick function of AFirstSCChart *Actor*



```
PIE: Play in editor total start time 0,088 seconds.
LogTemp: Warning: SCChart output: Hello
LogTemp: Warning: SCChart output: Unreal World
```

Figure 3.2. UE4 log output of the AFirstSCChart

After adding the *Actor* to the open scene and running the game the functions are executed in the aforementioned order and as a result, the output in Figure 3.2 is produced.

3.6 Elevator

3.6.1 The idea

An elevator is a good example for the use of an FSM because it has distinct states in which it should behave a certain way and is also used inside games. It is also possible to build a simple elevator implementation to get started and test the results and then later build upon this to further refine or improve the earlier functionality. Especially the latter conveyed itself to be very convenient while implementing the first more complex SCCharts for this thesis.

3.6.2 FSM modeling in SCCharts

For the elevator, a relatively simple design approach was chosen. The elevator waits for input, can move to a desired floor and can open or close doors. This defines the main behavior for the elevator and illustrates the use of SCCharts for this example and the corresponding workflow. The elevator's implementation graph is shown in Figure 3.3. Furthermore, the elevator has three main `states` as well as 4 `substates`. If the elevator's FSM is in the superstate `Idle` it does not move and waits for input. If it received input in form of the integer input variable `targetFloor` it takes one of the two outgoing transitions to the `MovingUp` or `MovingDown` state iff it resides in the `RdyForTakeoff` state. This requirement is because the used outgoing transitions are `join` transitions. This special kind of transition can only be taken if the exiting superstate, in this case, the `Idle` state, dwells inside its `final state` in the tick the conditions of the transition are met. This behavior is depicted by the little green arrows at the start of the transitions which indicate the `join` transitions as well as the double outline for the `final` state. Besides the `final` state, the `Idle` state has three more substates. These substates signal to wait or either open or close the doors.

In the states `MovingUp` and `MovingDown` the FSM outputs the signals `moveUp` or `moveDown` respectively as well as the integer variable `numFloorsToMove`. The variable `numFloorToMove` exposes to the outside how many floors the elevator must move to reach the target floor. The FSM resides inside one of the moving states until the input variable `currentFloor` is equal to `targetFloor`. If the condition is satisfied the transition back to the `Idle` state is taken.

To communicate with its environment, the FSM outputs an integer variable called `stateID`. This variable represents the state in which the FSM resides in the current `tick`. This variable needs to be evaluated inside the plugin's source code to determine in which state the SCCharts is currently. This is a common way for synchronous languages to communicate with their environment and comes with the advantage that the environment can easily be changed or adopted without the need for change to the model itself. The complete source code for the elevator SCCharts shown in Figure 3.3 is presented in Listing 3.8.

3.6.3 Unreal Engine integration

To make use of the compiled SCCharts code, a wrapper is needed as described in Section 3.3. The wrapper code is shown in Listing 3.12 as well as the header file in Listing 3.13. The functionality like pressing a button or moving the elevator is modeled with the help of UE4's *Blueprints*. They are contained in the GitHub repository since they are not part of this thesis. Nevertheless, to serve as an example the event that moves the elevator is laid out in Figure 3.4.

The handling of the *stateID* received from the SCChart FSM is done in C++ code and shown in Listings 3.9 and 3.10. The header file is displayed in Listing 3.11, which declares the functions for the Blueprint and the FSM interactions. Any function with the macro `UFUNCTION (BlueprintCallable)` is marked as being callable from Blueprints and the macro `UFUNCTION( BlueprintImplementableEvent)` declares a function that is implemented in Blueprints instead of C++ code. These functions allow communication with the Blueprints and execution of specific actions in accordance with the state ID received from the FSM.

3.6.4 Demonstration of the elevator

The source code and the project files for this example can be found in the GitHub repository [Ras21a]. A visual representation of the result is uploaded to YouTube and can be found under [Rasa]. Also, Figures 3.5, 3.6 and 3.7 show screenshots of the elevator and the SCCharts with the current state highlighted. In Figure 3.5, the elevator can be seen waiting for user input. The Figure 3.6 shows the elevator closing the doors after the passenger pressed a button of another floor. And lastly, the elevator moves down, which is shown in Figure 3.7

3.7 Rolling ball

3.7.1 The idea

The rolling ball implementation demonstrates one possibility of how a hybrid system could be used in a game environment. The ball has a target speed and a Proportional Integral Derivative (PID) controller to control said speed in dependence of the current speed as input. This is quite similar to how cruise control in cars work but simplified. It is also notable that the rolling ball can climb steep slopes as well as stairs since the acceleration received from the

```
scchart Elevator {
    input signal doorOpen
    input int currentFloor, targetFloor
    output signal moveUp, moveDown
    output int stateID, numFloorsToMove

    region {
        // The initial State where this FSM initially starts
        initial state Idle {
            // here the elevator doors would open
            initial state OpenDoors {
                entry do stateID = 1
            }
            if doorOpen go to WaitForInput
            // this state is essentially an Idle state where the elevator
            // waits for its input
            state WaitForInput {
                entry do stateID = 2
            }
            if  targetFloor != currentFloor go to CloseDoors
            // this state handles the door closing
            state CloseDoors {
                entry do stateID = 3
            }
            if !doorOpen go to RdyForTakeoff
            // final state signaling that the elevator can move
            final state RdyForTakeoff
        }
        // here is a join transition which can only be taken
        // if the state it is coming from resides in a final state
        if targetFloor > currentFloor join to MovingUp
        if targetFloor < currentFloor join to MovingDown
        // if this elevator is in this state it is moving upwards
        state MovingUp {
            entry do stateID = 4
            entry do numFloorsToMove = targetFloor - currentFloor
            entry do moveUp
        }
        // if the elevator stopped moving it returns to the Idle state
        if targetFloor == currentFloor go to Idle
        // if this elevator is in this state it is moving downwards
        state MovingDown {
            entry do stateID = 5
            entry do numFloorsToMove = currentFloor - targetFloor
            entry do moveDown
        }
        // if the elevator stopped moving it returns to the Idle state
        if targetFloor == currentFloor go to Idle
    }
}
```

Listing 3.8. The SCCharts source code of the graph in Figure 3.3 for the Elevator

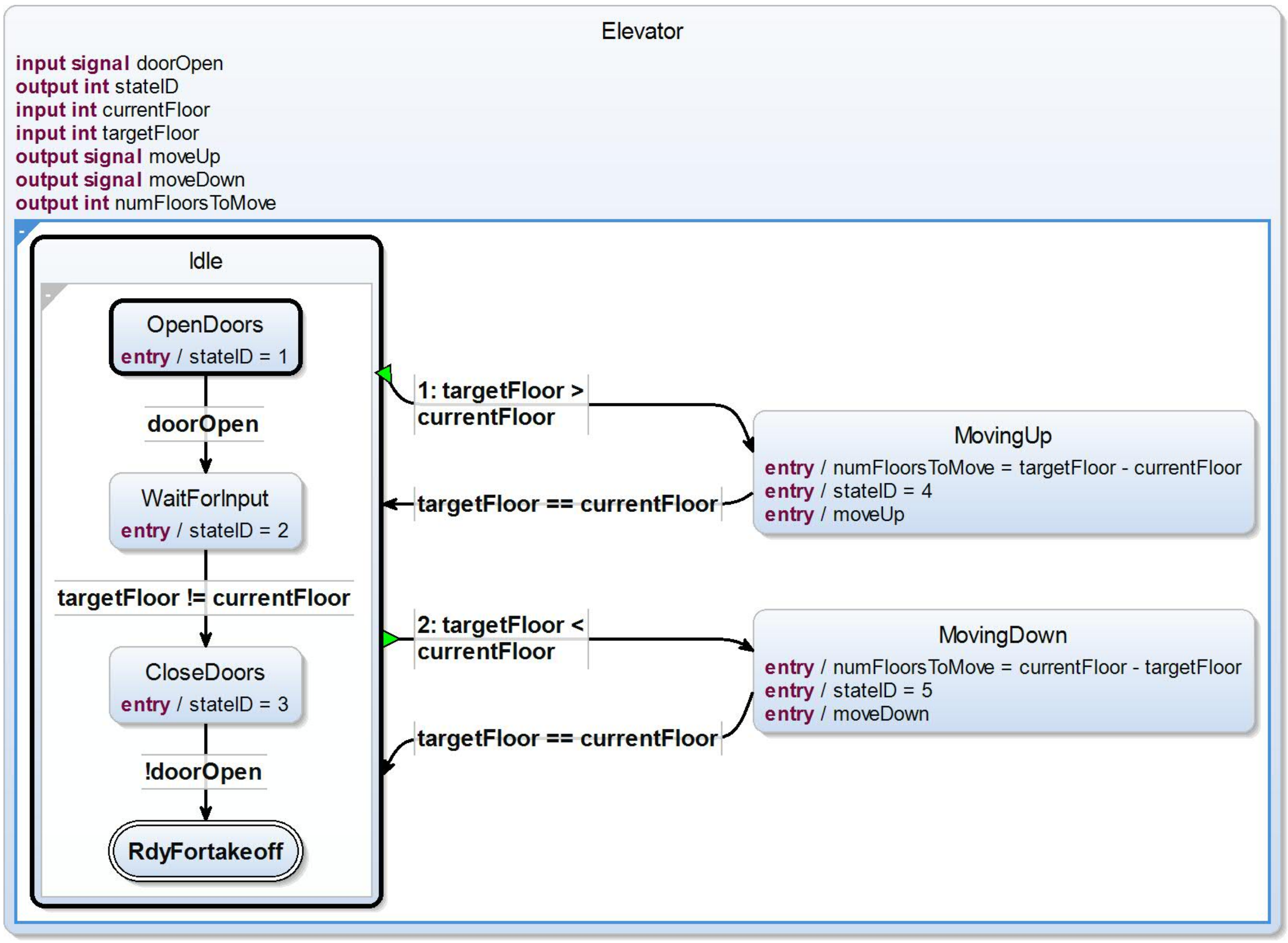


Figure 3.3. The SCCharts of the Elevator

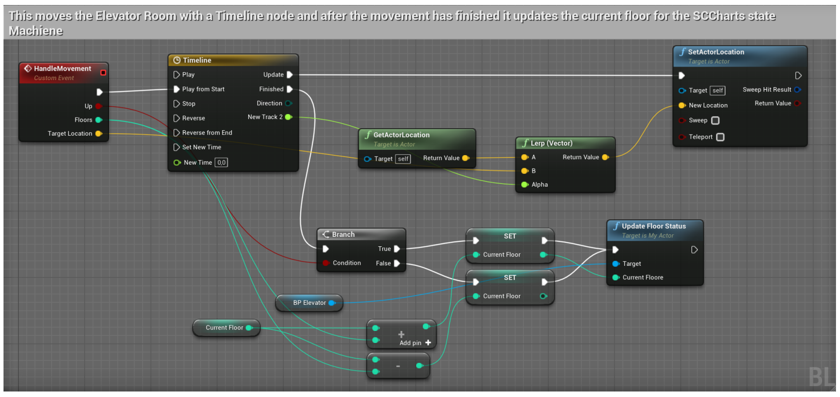

This moves the Elevator Room with a Timeline node and after the movement has finished it updates the current floor for the SCCharts state Machiene

Figure 3.4. The *Blueprint* that moves the elevator

```
#include "Elevator.h"

// Sets default values
AElevator::AElevator() {
    // Set this actor to call Tick() every frame.  You can turn this off to
     improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;
}
// Wrapper to relay the press of the button up to the FSM
void AElevator::PressBtn(int Floor) {
    Queue.Enqueue(Floor);
}

// Updates the doorOpen value inside the SCCharts model to reflect the same value
  inside the game
void AElevator::UpdateDoorStatus(bool isOpen) {
    Elevator.InputDoorOpen(isOpen);
}
// Updates the currentFloor value inside the SCCharts model to reflect the same
 value inside the game
void AElevator::UpdateFloorStatus(int Floor) {
    CurrentFloor = Floor;
    Elevator.tickData.currentFloor = Floor;
}
// Calls the elevator to the given floor
void AElevator::CallElevator(int Floor)  {
    // this Queue holds the requested number of floors in order of the requests
     given
    Queue.Enqueue(Floor);
}

// Called when the game starts or when spawned
void AElevator::BeginPlay() {
    Super::BeginPlay();
    // The SCCharts FSM needs to be ressetted before it can recive the first tick
    Elevator.Reset();
}
```

Listing 3.9. Utility functions of the elevator Actor

```
// Called every frame
void AElevator::Tick(float DeltaTime) {
    Super::Tick(DeltaTime);
    Elevator.Tick();
    DebugStateID = Elevator.tickData.stateID;
    int NextFloor = -1;
    // Check if the state changed. This is necessary to prevent opening the
    // doors repeatedly if they are already open
    if (Elevator.LastStateID != DebugStateID)   {
        switch (DebugStateID)       {
            case 1: // State: OpenDoors
                // Open the doors at the floor given by the SCCharts value
                  currentFloor
                OpenOutsideDoors(Elevator.tickData.currentFloor);
                OpenElevatorRoomDoors(); // also opens the Elevator doors
                break;
            case 3: // State: Wait
                // Close the doors at the floor given by the SCCharts value
                  currentFloor
                CloseOutsideDoors(Elevator.tickData.currentFloor);
                CloseElevatorRoomDoors(); // also closes the Elevator doors
                break;
            default:
                break;
            }
    } else if (Elevator.LastStateID == 2) {
        // the Queue that holds the requested floor destinations
        if ( Queue.Dequeue(NextFloor)) {
            // if Dequeue was successful at dequeuing,
            // the value is written to the SCCharts targetFloor value
            Elevator.tickData.targetFloor = NextFloor;
        }
    }
    // handles the output signals of the SCCharts
    if (Elevator.tickData.moveUp) {// if moveUp is present
        // move the Elevator up by the given number of floors
        MoveElevator(true, Elevator.tickData.numFloorsToMove);
    }
    if (Elevator.tickData.moveDown) {// if moveDown is present
        // move the Elevator down by the given number of floors
        MoveElevator(false, Elevator.tickData.numFloorsToMove);
    }
}
```

Listing 3.10. The tick function of the elevator Actor

```
class SCCHARTSFORGAMES_API AElevator : public AActor {
    GENERATED_BODY()

public:
    // Sets default values for this Actor's properties
    AElevator();
    UFUNCTION(BlueprintCallable)
    void PressBtn(int Floor);
    UFUNCTION(BlueprintCallable)
    void UpdateDoorStatus(bool isOpen);
    UFUNCTION(BlueprintCallable)
    void UpdateFloorStatus(int Floor);
    UFUNCTION(BlueprintImplementableEvent,BlueprintCallable)
    void OpenOutsideDoors(int Floor);
    UFUNCTION(BlueprintImplementableEvent,BlueprintCallable)
    void CloseOutsideDoors(int Floor);
    UFUNCTION(BlueprintImplementableEvent,BlueprintCallable)
    void OpenElevatorRoomDoors();
    UFUNCTION(BlueprintImplementableEvent,BlueprintCallable)
    void CloseElevatorRoomDoors();
    UFUNCTION(BlueprintImplementableEvent,BlueprintCallable)
    void MoveElevator(bool Up, int NumOfFloors);
    UFUNCTION(BlueprintCallable)
    void CallElevator(int Floor);

    // Called every frame
    virtual void Tick(float DeltaTime) override;
    UPROPERTY(BlueprintReadOnly)
    int DebugStateID = 0; // to display the SCCharts
    TQueue<int> Queue; // to queue button input
protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;
    ElevatorFSM Elevator = ElevatorFSM();
    bool bDoorOpen;
    UPROPERTY(BlueprintReadWrite)
    bool bMoving;
    UPROPERTY(BlueprintReadWrite)
    int CurrentFloor;
};
```

Listing 3.11. Header file of the elevator Actor

```
#include "ElevatorFSM.h"
extern "C" {
#include "../kieler-gen/Elevator.h"
}

// Wrapper for the SCCharts tick function. Also saves the last StateID
void ElevatorFSM::Tick() {
    LastStateID = tickData.iface.stateID;
    tick(&tickData);
}

// this increments the target floor
void ElevatorFSM::inputUp() {
    tickData.iface.targetFloor = tickData.iface.currentFloor + 1;
}

// this decrements the target floor
void ElevatorFSM::inputDown() {
    tickData.iface.targetFloor = tickData.iface.currentFloor - 1;
}

// updates the doorOpen value in the FSM
void ElevatorFSM::inputDoorOpen(bool isOpen) {
    tickData.iface.doorOpen = isOpen;
}

// wrapper to reset the FSM
void ElevatorFSM::Reset() {
    reset(&tickData);
}
```

Listing 3.12. The C++ source code of the wrapper for the SCCharts in Figure 3.3

FSM reaches high enough levels to enable this behavior. There is, however, an implemented accelerations limit to prevent too erratic behavior.

3.7.2 FSM modeling in SCCharts

The rolling ball is a simplification of a cruise control problem. The aim is to hold a constant speed level by adjusting the acceleration as needed. For this to work, the SCChart needs to know the current speed as well as the time. The time is internally accumulated via a `clock` construct and an input variable `deltaT`. To calculate the appropriate acceleration change, a PID controller is used as stated before. The PID controller is a control loop mechanism that continuously calculates an `error` by the means of proportional, integral and derivation action, as the name suggests. The proportional control takes the current error multiplied by a calibration variable `KP`.

```
#pragma once

#include "CoreMinimal.h"

extern "C" {
    #include "../kieler-gen/Elevator.h"
}

class SCCHARTSFSM_API ElevatorFSM {
public:
    ElevatorFSM();
    ~ElevatorFSM();
    void Tick();
    void Reset();
    void inputUp();
    void inputDown();
    void inputDoorOpen(bool isOpen);
    TickData tickData = TickData();
    int LastStateID;
};
```

Listing 3.13. The C++ header code of the wrapper

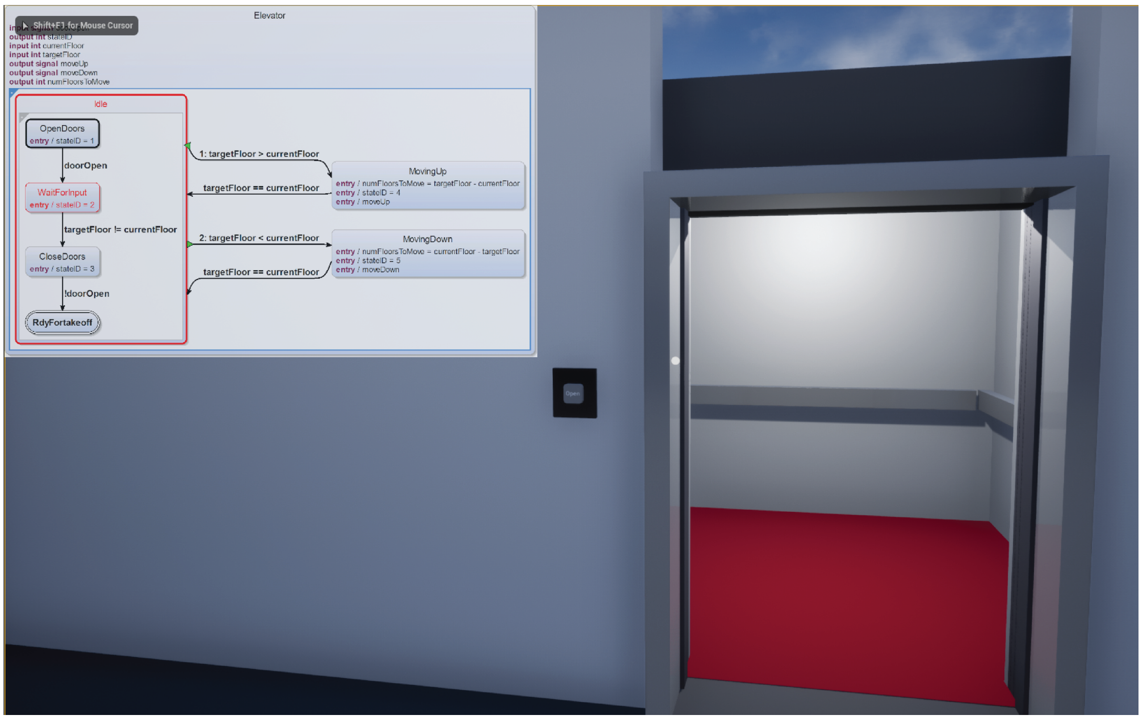


Figure 3.5. The elevator waiting for input

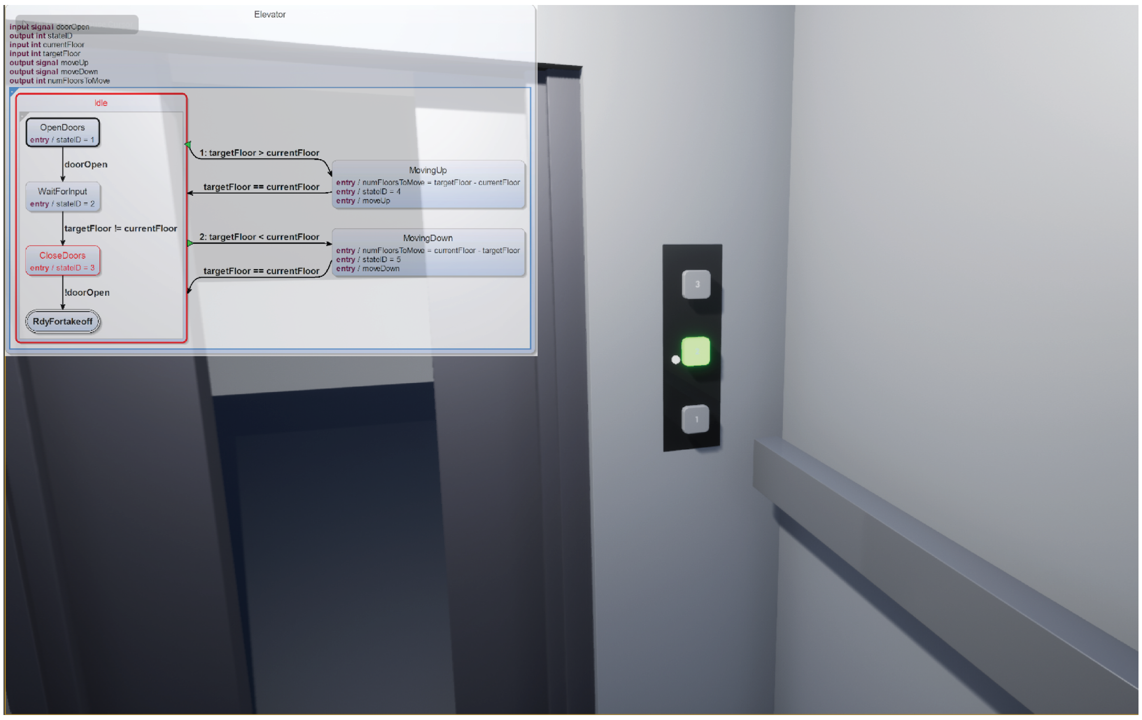


Figure 3.6. The elevator closing the doors

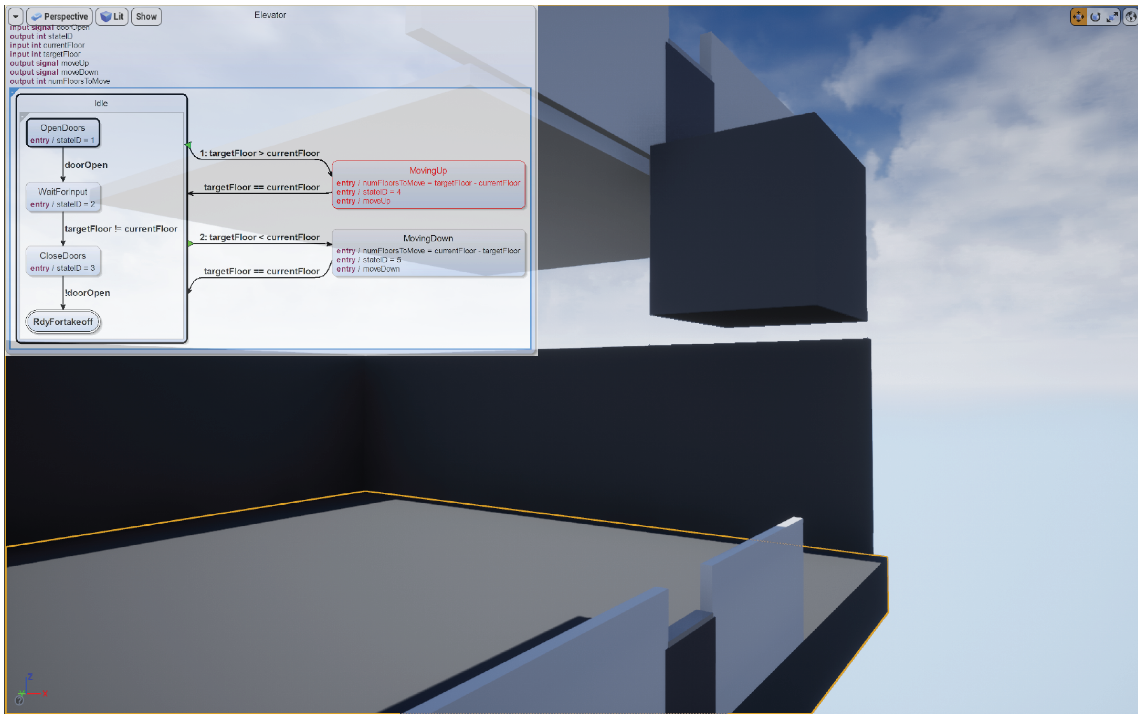


Figure 3.7. The elevator moving to the third floor

However, this alone would not be very accurate since the proportional action can only react to the error margin directly with no regards to factors like time. This is where the integral and deviation part comes into play. The integral part of the controller accumulates the data over time and takes action not only to the error itself like the proportional action but also in consideration of the time since the system is running. Lastly, the derivative action takes the rate of change into account and can increase precision by dampening the output to prevent overshooting. Altogether, the PID controller can accurately control a model if calibrated well enough. Since this is only an example of how to use SCCharts inside a game environment the calibration is only loosely configured to get a running system and is not part of this thesis.

To model the PID controller in SCCharts the dataflow syntax is used. Dataflow is a programming paradigm that represents applications as a directed graph similar to a dataflow diagram [Sou12]. This paradigm is also known from other languages like Lustre and VHDL where the flow of data is one of the main aspects. With the use of the dataflow syntax inside SCCharts, one can conveniently model PID controller calculations. The resulting SCCharts is shown in Figure 3.8 and the correlated source code in Listing 3.14. More about SCCharts implementation of the dataflow paradigm as well as the use of real-time inside timed automata can be found in the SCCharts' syntax documentation [SS20] and the documentation about timed automata [Sch20] respectively.

3.7.3 Unreal Engine integration

As described in Subsection 3.6.3, after compiling the finished SCCharts, a wrapper is needed to interact with UE4. The header file shown in Listing 3.15 declares the needed functions in order to interact with the SCCharts C code. The `Tick` function takes two arguments, `Velocity` and `DeltaTime`. The `Velocity` is simply the current velocity of the ball and the `DeltaTime` again is the time between the current and the last game frame. The source file containing the implementations is shown in Listing 3.16. Besides the `Tick` function and the `Reset` function known from the previous example in Section 3.6.3 a `Setup(...)` function was implemented. This function is used to calibrate the PID controller in the `Actor` class that uses this wrapper. The *Actor's* header in Listing 3.17 declares the needed functions and variables which get implemented in Listing 3.18. In this case, in contrast to the Elevators implementation, all of the functionality needed for the rolling ball is contained in Listing 3.18 with no additional Blueprints. Furthermore, the calibration used for the PID controller contains empirical values only that suffice for demonstration purposes.

3.7.4 Demonstration of the rolling ball

As with the demonstration of the Elevator, only a live or video presentation would be appropriate to accuratly demonstrate this case. The demo for the rolling ball is available as a UE4 project in the SCCharts for UE4 Demo repository [Ras21a] and a video showing the rolling ball on YouTube [Rasb]. Nevertheless, Figure 3.9 shows how the rolling ball traverses over the obstacle course at different points in time. The rolling ball is able to maintain the

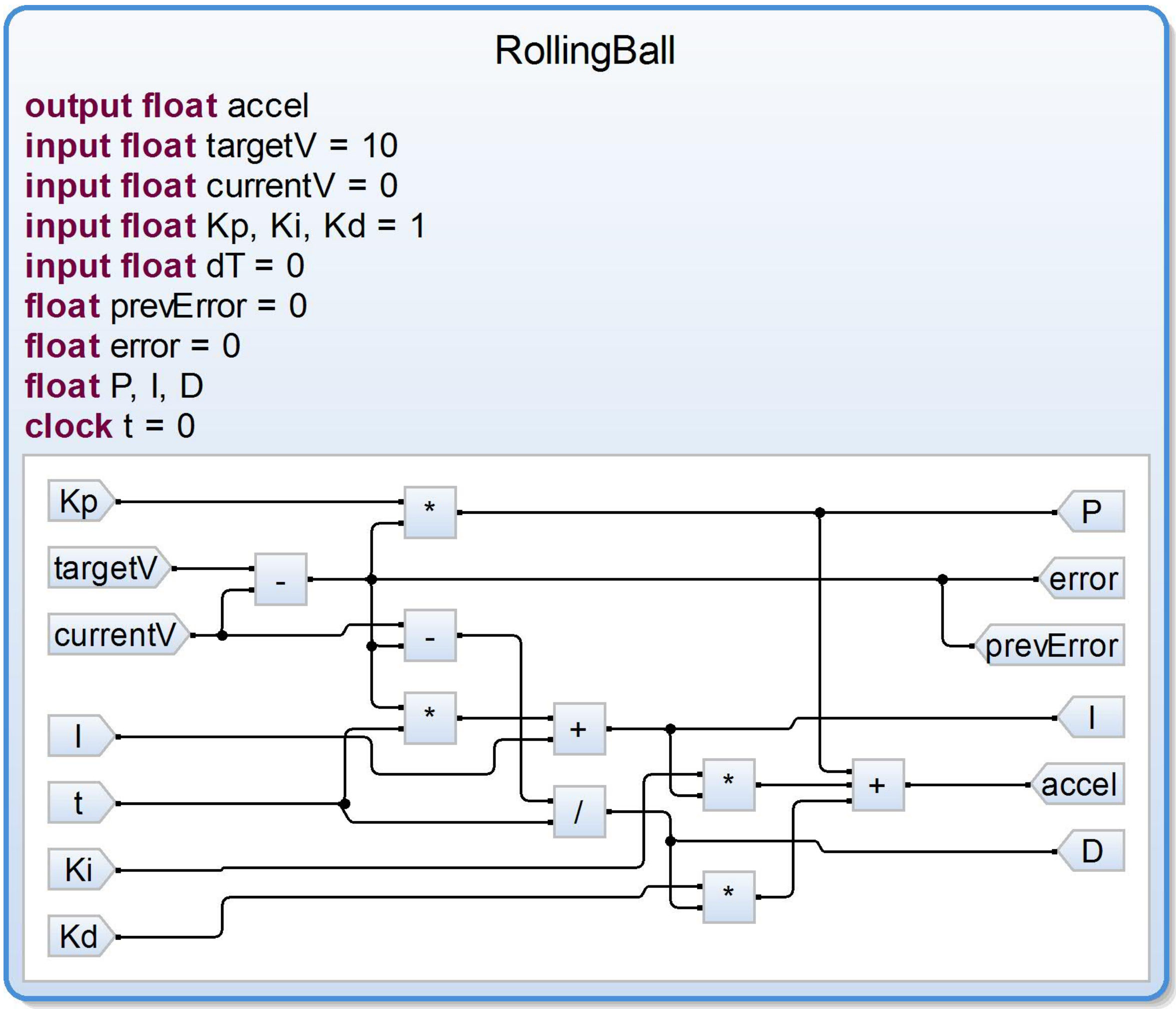


Figure 3.8. The SCChart of the rolling ball

desired speed relatively accurate, even when climbing stairs.

```
@ClocksUseSD
scchart RollingBall {

  output float accel // output acceleration change

  input float targetV = 10 // target velocity input
  input float currentV = 0 // current velocity input
  input float Kp, Ki, Kd // PID calibration variables input
  input float dT = 0 // delta Time input
  float prevError = 0 // error in the previous tick

  float error = 0// divergence from the target velocity
  float P, I, D // internal variables for PID calculation
  clock t = 0 // clock for real time calculation

  dataflow:
  prevError = error
  error = targetV - currentV
  P = Kp * error
  I += error * dT
  D = (currentV - prevError)/dT
  accel = P + Ki * I + Kd* D
}
```

Listing 3.14. The C++ header code of the SCCharts wrapper class

```
#pragma once

#include "CoreMinimal.h"
extern "C" {
#include "../kieler-gen/RollingBall.h"
}

class ROLLINGBALLHS_API RollingBallHybridSystem {
public:
    float Tick(float Velocity, float DeltaTime);
    void Reset();
    void Setup(float TargetSpeed = 50, float Kp = 1 , float Ki = 1, float Kd = 1)
     ;
    TickData TickData;
};
```

Listing 3.15. The C++ header code of the wrapper

Modified version with corrected formular in the rolling ball SCCharts calculation.

```
#include "RollingBallHybridSystem.h"
float RollingBallHybridSystem::Tick(const float Velocity, const float DeltaTime)
 {
    TickData.deltaT = DeltaTime;
    TickData.currentV = Velocity;
    tick(&TickData);
    return TickData.accel;
}

void RollingBallHybridSystem::Reset() {
    reset(&TickData);
    tick(&TickData);
}

void RollingBallHybridSystem::Setup(const float TargetSpeed, const float Kp,
                                    const float Ki, const float Kd) {
    TickData.Kp = Kp;
    TickData.Ki = Ki;
    TickData.Kd = Kd;
    TickData.targetV = TargetSpeed;
}
```

Listing 3.16. The C++ source code of the wrapper class RollingBallHybridSystem

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Pawn.h"
#include "RollingBallHybridSystem.h"
#include "RollingBall.generated.h"


UCLASS()
class SCCHARTSFORGAMES_API ARollingBall : public APawn {
    GENERATED_BODY()

public:
    // Sets default values for this pawn's properties
    ARollingBall();
    virtual void Tick(float DeltaTime) override;
    virtual void SetupPlayerInputComponent(
        class UInputComponent* PlayerInputComponent
    ) override;
protected:
    virtual void BeginPlay() override;
    UPROPERTY(BlueprintReadWrite)
    // the 3D Ball object in the world which is also the root of this Actor
    UPrimitiveComponent* Component;
    // The SCChart wrapper class
    RollingBallHybridSystem RollingBallHS;
};
```

Listing 3.17. The C++ header file of the RollingBall *Actor*

```
#include "RollingBall.h"

ARollingBall::ARollingBall() {
    // Set this pawn to call Tick() every frame.  You can turn this off to
     improve performance if you don’t need it.
    PrimaryActorTick.bCanEverTick = true;
}

void ARollingBall::BeginPlay() {
    Super::BeginPlay();
    Init();
}
// function to initialize the Actor
void ARollingBall::Init() {
    RollingBallHS.Reset();
    RollingBallHS.Setup(100, 25, 10, 0.25);
}

void ARollingBall::Tick(float DeltaTime) {
    Super::Tick(DeltaTime);
    // retrieve the current velocity as a vector
    const FVector V = GetVelocity();

    // the new acceleration for this tick returned from the SCCharts wrapper
    // the Tick function receives the current velocity vector V’s magnitude to
    // so the total direction independent speed is used
    float Accel =  RollingBallHS.Tick(V.Size(), DeltaTime);
    GEngine->AddOnScreenDebugMessage(-1, 15.0f, FColor::Yellow, FString::Printf(
     TEXT("Accel␣%f"), Accel));
    // This limits the Acceleration to |Accel| <= 10000 to prevent to erratic
     behavior
    Accel = FMath::Sign(Accel) * FMath::Min<float>(FMath::Abs(Accel), 5000);
    if (IsValid(component)) { // checks if the component is exists
        // sets the newly received acceleration as the new force for the ball in
         x direction
        Component->AddForce(FVector(Accel,0,0), NAME_None, true);
    }
}

// created by UE4
void ARollingBall::SetupPlayerInputComponent(UInputComponent*
 PlayerInputComponent) {
    Super::SetupPlayerInputComponent(PlayerInputComponent);
}
```

Listing 3.18. The C++ source code of the RollingBall *Actor*

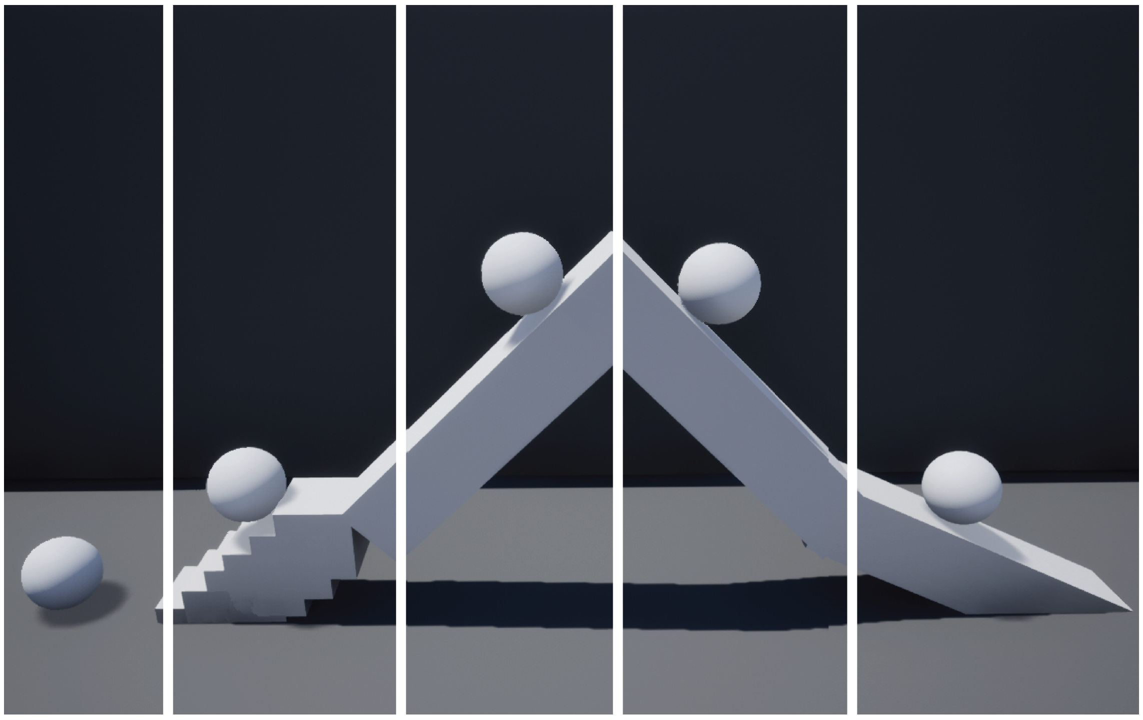


Figure 3.9. The rolling ball at different points in time from left to right

Chapter 4

Conclusion

The idea to use SCCharts as a means to improve the design and development workflow for games has shown to be very useful. The overhead is relatively small compared to the advantages of using a pre-built sophisticated language for FSM in comparison to writing the alternatives. With its many features, it is possible to create robust FSMs that can also be visualized. The different kinds of languages features like advanced transitions or nested state machines allow for defining complex behavior in a simplified fashion. The simulation provided by the KIELER IDE aids development and increases the iterability of the design by testing specific behavior without the need to run it in a complete game environment. To reduce errors one can make use of the integrated simulation in SCCharts IDE as well as use model checking to describe unwanted behavior even before building the FSM. The automated layout generated from the written code offers an accessible overview of the FSM and allows to find mistakes by reading the chart instead of understanding the logic inside the code.

Since it is not possible in most engines to include the generated SCCharts code directly, one has to first compile the SCCharts to e.g. C code. After the code generation, a wrapper is needed in most cases which then has to be compiled as a linked library. Furthermore, the ability to call functions on objects directly can only be achieved with even more overhead unless SCCharts can compile to the engines natively supported language. Therefore, the states must be evaluated in the environment it is implemented in. This means that another possibility for errors exists, because not only can the SCCharts implementations be erroneous but also in the place where the states are being evaluated and the correlation functions are called.

All in all, SCCharts is a versatile language that aids in the development process of games at any scale and increases readability by a well designed automated layout that reduces time and effort previously put into manual layouting.

If this approach is adopted in future game projects, it could lead to more robust games with fewer bugs which would be beneficial not only to developers but also to the company images as well as for the players.

Chapter 5

Future Work

After testing how well SCCharts can be integrated into a game development environment, it is interesting to see where else it would be applicable. One of the main development issues though is the need to create separate wrappers and to compile libraries instead of being able to use the engine's host code. In the future, however, this requirement could be alleviated for engines that support native C++. This is because the C++ compilation is in the backlog of the SCCharts development team at the Christian-Albrecht University of Kiel. This opens up a simpler, more powerful workflow for supporting engines since the functions could be called directly on the object instances. This would reduce the amount of work required to communicate with the game environment and eliminate a potential source of error by containing all the logic needed in one place instead of spreading it across a SCChart, a wrapper and a class.

Furthermore, the model verification, which is not further elaborated in this thesis, could be a very useful tool in building more robust games. One could state the behavior in linear temporal logic that is known to be problematic or reported by players to cause issues. The model verification would then check whether or not this issue could arise and the developer can fix the problematic area by using the feedback provided by the model checker.

Bibliography

[ED79] Hopcroft John E. and Ullman Jeffrey D. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[Epia] Epic. *Behavior Trees*. URL: https://docs.unrealengine.com/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/index.html.

[Epib] Epic. *C++ Plugin Based State Machine*. URL: https://www.youtube.com/watch?v=hr9ybCCPw9Y&t=1323s.

[Epic] Epic. *State Machines*. URL: https://docs.unrealengine.com/en-US/AnimatingObjects/SkeletalMeshAnimation/StateMachines/index.html.

[Epid] Epic. *UE4 Plugin Documentation*. URL: https://docs.unrealengine.com/en-US/ProductionPipelines/Plugins/index.html.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "SCCharts: Sequentially Constructive Statecharts for safety-critical applications". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Edinburgh, UK: ACM, June 2014, pp. 372–383.

[L16] Bruno Xavier L. *UFSM State-Machine*. 2016. URL: https://www.unrealengine.com/marketplace/en-US/product/ufsm-plugin.

[LMG20] Juan Linietsky, Ariel Manzur, and the Godot community. *AnimationTree*. 2020. URL: https://docs.godotengine.org/en/stable/tutorials/animation/animation_tree.html.

[Mic20] Microsoft. *What is a DLL*. 2020. URL: https://docs.microsoft.com/en-us/troubleshoot/windows-client/deployment/dynamic-link-library.

[Rasa] Philip Raschkowski. *Sccharts for game development - elevator demo*. URL: https://www.youtube.com/watch?v=cwwZo2Cen5c.

[Rasb] Philip Raschkowski. *Sccharts for game development - rolling ball demo*. URL: https://www.youtube.com/watch?v=B8uSIVJ4MWg.

[Ras21a] Philip Raschkowski. *UE4 SCCharts demo*. 2021. URL: https://github.com/diocore/SCChartsForGamesDemoProject.

[Ras21b] Philip Raschkowski. *UE4 SCCharts plugin template*. 2021. URL: https://github.com/diocore/SCChartsForUEPlugin.

[Rec19] Recursoft. *Logic driver pro - blueprint editor*. 2019. URL: https://www.unrealengine.com/marketplace/en-US/product/logic-driver-state-machine-blueprint-editor.

[San17] Ricardo dos Santos Moreira. “Quadrotor Simulator for Control Development – Application to Autonomous Landing”. Dissertation. Universidade Nova de Lisboa, Faculty of Science and Technology, Sept. 2017.

[Sch20] Alexander Schulz-Rosengarten. *Timed Automata*. 2020. URL: https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Timed+Automata.

[SD] Alexander Schulz-Rosengartena and Soeren Domroes. *Keith*. URL: https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KEITH.

[Sho20] Najmm Shora. *State Pattern using Unity*. 2020. URL: https://www.raywenderlich.com/6034380-state-pattern-using-unity.

[SL20] Steven Smyth and Daniel Lucas. *SCCharts*. 2020. URL: https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/SCCharts.

[Sou12] Tiago Sousa. “Dataflow programming: concept, languages and applications”. In: Doctoral Symposium on Informatics Engineering, Jan. 2012.

[SSa] Steven Smyth and Alexander Schulz-Rosengarten. *KIELER: The Kiel Integrated Environment for Layout (Eclipse Rich Client)*. URL: https://rtsys.informatik.uni-kiel.de/confluence/pages/viewpage.action?pageId=57802970.

[SSb] Steven Smyth and Alexander Schulz-Rosengarten. *Simulation visualization (kivis)*. URL: https://rtsys.informatik.uni-kiel.de/confluence/pages/viewpage.action?pageId=57802970.

[SS20] Steven Smyth and Alexander Schulz-Rosengarten. *Syntax*. 2020. URL: https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Syntax (visited on 05/03/2010).

[Sta19] Andreas Achim Stange. “Model checking for SCCharts”. MA thesis. Kiel University, Department of Computer Scienece, May 2019. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aas-mt.pdf.

[Stu] Polyart Studio. *Pixel 2D - Complete 2D Engine for Unreal*. URL: https://www.unrealengine.com/marketplace/en-US/product/pixel-2d-complete-2d-engine-for-unreal.

[Uni21] Unity. *Animation State Machines*. 2021. URL: https://docs.unity3d.com/Manual/AnimationStateMachines.html.

[Unk12] Unknown. *Finite State Machine*. 2012. URL: http://wiki.unity3d.com/index.php?title=Finite_State_Machine.