Fakultät für Informatik
Technische Universität München

TUM

Bachelor’s Thesis in Informatics: Games Engineering

Development of a High-Performance Engine to Execute Statecharts for Games

Matthias Ellerbeck

Fakultät für Informatik
Technische Universität München

Bachelor’s Thesis in Informatics: Games Engineering

Development of a High-Performance Engine to Execute Statecharts for Games

Entwicklung einer leistungsstarken Engine zur Ausführung von Statecharts für Games

Author:	Matthias Ellerbeck
Supervisor:	Gudrun Klinker, Phd
Advisors:	Msc. Daniel Durda
Submission Date:	April 15, 2021

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Steingau, 13.04.2021

MATTHIAS ELLERBECK

Abstract

The thesis evaluates possible use cases for statecharts in games and extracts requirements from them. With these requirements an engine is developed which executes statecharts in the Unity game engine. This engine is then optimized using caching, cache optimizations, and other optimizations; and an approach for multithreading the engine is presented. The final engine is able to execute 1000 copies of a statechart within nine milliseconds on average. It is publicly available at `https://github.com/file-not-found42/StatechartEngine`.

Acknowledgements

First, I want to thank my supervisor Professor Gudrun Klinker, who made this thesis possible. I also want to thank my advisor Daniel Dyrda, who always responded quickly, pointed me in the right direction, prevented me from loosing myself in feature creep, and generally gave good advice. Then I want to thank my family for supporting me before and during the time I wrote this thesis on many levels. Finally, I would like to thank Michael Grupp for this LaTeX template.

Contents

Eidesstattliche Erklärung iii

Abstract iv

Acknowledgements v

1. Introduction 1

2. Overview on Statecharts 2
- 2.1. Statecharts: A visual formalism for complex systems . . . 2
- 2.2. The STATEMATE Semantics of Statecharts . . . 4

3. Related Work 7

4. Engine Features 8
- 4.1. Use Cases in Games . . . 8
- 4.2. Statechart Features to Implement . . . 12
- 4.3. Semantics . . . 12
- 4.4. Unity Interface . . . 12
- 4.5. Terminology . . . 14
- 4.6. File Format and File Requirements . . . 14

5. Implementation 15
- 5.1. Performance Test Methodology . . . 15
- 5.2. Naive . . . 15
  - 5.2.1. Public Interface . . . 17
  - 5.2.2. Central Classes . . . 17
  - 5.2.3. Step Algorithm . . . 19
  - 5.2.4. Performance . . . 19
- 5.3. Reducing the Workload . . . 21
  - 5.3.1. Precomputation and Caching . . . 21
  - 5.3.2. CPU Cache Optimizations . . . 22
  - 5.3.3. Other Optimizations . . . 25
- 5.4. Increasing the Workforce . . . 27

6. Final Results and Discussion 28

7. Conclusion and Outlook 31

A. Appendix: Full Performance Measurement Data 37

B. Appendix: Unity Profiler Screenshots 41

1. Introduction

As interactive media, games are reactive systems. Reactive systems as defined in [10] are systems which react to external input with both internal change and an output. The concept of statecharts presented in [11], which is an extension of finite state machines, is an approach to visualize and formalize these reactive systems.
A reactive system can often be split into many equally reactive subsystem, this is also the case for games. An example for such a subsystem in computer games are character controllers: Character controllers, among other features, define the movement capabilities of usually human controlled characters in a 2D or 3D level environment. Many games feature complex movement systems; examples include Titanfall 2 [30] and Warframe [8]. Both games feature extensive jumping capabilities, a ground slide, and wall running or wall hopping. Like any reactive systems the character controllers in these games can be modeled with statecharts, which provide formal, clear, and complete information on their capabilities, behavior, and complexity.
The capabilities of statecharts can also be used to inverse the process: Instead of modeling a statechart to visualize a character controller, the character controller is defined by the statechart. This allows for easy iteration over the controller: The different states a character can be in are clearly separated and transitions between the states are explicit and visually connected to their states, as opposed to a state machine implemented in code by e.g., a switch statement.
In addition to character controllers, the development of many other reactive systems in games can also profit from statecharts. For this reason some game engines already provide an implementation of state machines which resembles statecharts, usually in the form of animation graphs for layering and blending character animations [15]. Given the number of reactive subsystems in games and their real-time nature, the execution time for any statecharts is an important factor in the viability of statechart based systems in games. Computer games must produce a new image for the player (frame) usually 30 or 60 times per second, which means that all calculations for such a frame must be done within 33.3 or 16.6 milliseconds.
This thesis documents the implementation of a software which can execute 1000 statecharts in less than 16.6 milliseconds in the Unity game engine. First, it gives an overview on the syntax and semantics of statecharts as defined by their original author David Harel [11] [12] and summarizes the results of other papers concerned with the topic. It then describes use cases for statecharts in games to derive the required features and capabilities of the software. Afterwards the software is implemented and optimized to fulfill the objective of the thesis. Finally it presents and discusses the results.
The software developed in this thesis is available at `https://github.com/file-not-found42/StatechartEngine`

2. Overview on Statecharts

2.1. Statecharts: A visual formalism for complex systems

David Harel first presented statecharts in his article "Statecharts: A visual formalism for complex systems" with the goal to solve state count blowup and visual noise when modeling complex systems in a flat space. He did so by introducing abstraction and concurrency and many consequential concepts to state diagrams. While statecharts were designed as a formal but visual model of real systems, this thesis instead executes them as component of a game engine. As such, the visual aspects of statecharts are only of interest in this thesis to display examples. [11]

Events and Conditions In statecharts, a transition is triggered by a matching *event*, or always if there is no event necessary. When the transition is triggered, an optional *guard* must be true for the transition to be active. The guard can be anything that evaluates to a logical value, e.g., a greater-than comparison. Event triggers can be simulated using guards by testing it the event exists; guards can be simulated with event triggers by adding an event when a value changes. [11]
Events are labeled with their trigger event and a guard expression in square brackets, as depicted in figure 2.1. [11]

Hierarchy The first major concept which distinguishes statecharts from state diagrams is hierarchy. In statecharts, a state can contain other states to accumulate the common properties of these states. Such a *superstate* has one or more *components* or *substates*; a state which has no components is a *basic state*. Inversely, since the state hierarchy forms a tree, the state which is not a component of any other state is the *root*. Unchanged from state diagrams, only one basic state can be active at any time, therefore at most one component of a superstate can be active at any time. Transitions can start and end on any level of the hierarchy. Superstates have a default state similar to the start state of a state diagram in case they are entered by a transition. [11]
Visually, substates are drawn inside their superstates, as shown in figure 2.2. [11]

Orthogonality The second major concept new in statecharts is concurrency. Statecharts can model multiple interacting systems in parallel to avoid quick increases in state count. *Parallel* states are superstates of which every component is active simultaneously or none are, while in normal superstates at most one component can be active at once; for this

——TriggerEvent [guard expression] ⟶

Figure 2.1.: Transition containing both a trigger event and a guard expression

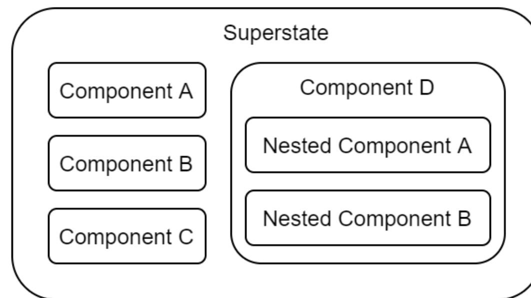

Figure 2.2.: Compound state containing components of which one is another compound state

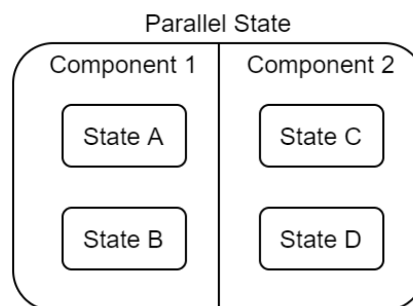

Figure 2.3.: Parallel state containing two components with two basic states each

reason normal superstates are also called OR states, and parallel states are called AND states. The components of parallel states themselves are usually superstates too. The components are inherently independent from each other, i.e., the behavior and actions of one component do not influence the other components by default. This does not prohibit explicit communication; a component can react to state changes in other parallel components, e.g., by transitions triggering when another state is entered or exited.
The components of parallel states are drawn with separating lines between them, as shown in figure 2.3. [11]

Actions and Activities Actions are sequences of statements a statechart executes. Actions are executed when a transitions is taken or a state is entered or exited. Actions take not time, they are defined to be executed instantaneously. Actions can both interface with external system and trigger transitions in their own statechart. [11]
Activities are executed while a state is active and hence take time to execute. Activities can be simulated with actions: Instead of executing an activity A, the state executes the action StartA when entered and the action EndA when exited. [11]

Other Features In addition to hierarchy and orthogonality statecharts also feature some additional and often related capabilities.

History A commonly useful feature in state based models is to resume previous behavior after an interrupt. For this purpose, statecharts feature history states. History

states are states which, when entered, redirect to the last visited state of their superstate. A variation are deep history states which not only remember the last visited state on their own level but also all levels below. The previously visited state(s) of history states can be deleted. History states are the first occurrence of pseudo-states in statecharts, despite the concept of pseudo-states in its entirety not being introduced in the article. Pseudo-states are states in which the statechart cannot stop but must proceed further though transitions or other means. [11]
History states are drawn as circles marked with a capital H; deep history states are marked with H*. [11]

Transition Pooling Statecharts also support the only visual transition pooling to reduce visual clutter brought by multiple transitions with their labels connecting distant states. Pooled transitions represent common sources, targets or events among transitions. They are drawn as pseudo-states in the form of single filled dots. [11]

Selection Selection states again are pseudo-states which, when entered, present the user with a choice between all components of their super state. [11]

Timeouts and Delays States with timeouts automatically trigger an exiting transition after a time span passed after the state was entered. Inversely, states with delays ignore all exiting transitions until the delay has passed. [11]

Problems When examining statecharts, some problems are evident. When multiple transitions originating from a state are triggered simultaneously the behavior is undefined. When transitions of superstates are triggered in addition to transitions starting from lower levels the behavior is equally undefined. The order of transitions and actions executed between the components of a parallel state is also undefined, which is especially problematic in the case of these transitions and actions triggering other transitions in other components, as is the behavior when states in multiple concurrent components have active transitions exiting the parallel superstate to different states. Michael von der Beeck lists these problems and more in [35], but the original article identifies some of them already. Many will be solved in the proposed semantics in [12].

2.2. The STATEMATE Semantics of Statecharts

Since no governing body for statecharts exists and no semantics were presented in [11] many organizations developed their own semantics for statecharts. The semantics used in the STATEMATE system, developed by the original inventor of statecharts David Harel and Amnon Naamad among others, are presented in [12] and provide the closest approximation to official semantics available. [12]
In the article some core terminology is defined: A *run* describes everything happening in a statechart from the start of execution to the end. It consists of *statuses*, which describe the current status of the statechart, and *steps*, the transitions between those statuses. [12]

General Rules A few general rules are set for the semantics: Statuses are immutable; during a step the status the step operates on does not change; therefore a step cannot influence itself. Events have a lifetime of exactly one step. Following the first rule, an event originating in step n only exists in step $n + 1$ and it cannot influence the step it is created in; the same applies to changes in the set of active states. Everything that can be done in a step must be done in this step. Finally, a step takes no time; it happens in an instant to outside observers. [12]

Configuration A *configuration* of a statechart is a valid set of active states. A configuration is valid if the following properties taken directly from [12] apply:

> C contains R.
> If C contains a state A of type OR, it must also contain exactly one of A's substates.
> If C contains a state A of type AND, it must also contain all of A's substates.
> The only states in C are those that are required by the above rules.

Consequently, if a configuration contains a basic state it also contains all of its superstates, and knowledge of the currently active basic states is sufficient to uniquely determine the current configuration. Such a set of basic states is a *basic configuration*. When a step does not change the configuration of a statechart the step is empty. [12]

Transitions A maximal sequence of transitions which can be executed is a *basic compound transition* or *CT*. A *full CT* consists of one or more basic CTs and results in a valid configuration of the statechart. The scope of such a full CT is the superstate which contains all states entered and exited by this full CT. A CT is enabled when all its transitions are triggered and the guards evaluate to true. Two or more CTs are in *conflict* when they share exited states. In STATEMATE, the CT with the largest scope is chosen; if there is no largest scope the system halts or chooses a CT at random. [12]
The paper also introduces some new pseudo-states. Connectors are the non-visual variant of transition pooling. In fork pseudo-states all exiting transitions must be contained in any CT including the state; in join pseudo-states all incoming transitions must be contained in the CT. In condition, selection, and junction pseudo-states exactly one incoming and one outgoing transition must be included any CT that passes them. [12]
In history pseudo-states, an incoming transition always enters the superstate containing the history state from the outside regardless of the source of that transition. If there is no previously active state for the history state to enter, the default entry for its superstate is taken or a transition originating from the history state. [12]

Step The paper also lists a complete step algorithm in pseudocode. For details see [12].

Time Models STATEMATE supports multiple time models to synchronize the execution of a statechart with external systems, e.g., physics engines. The *synchronous* model executes one step per external time unit. The *asynchronous* model triggers when an external event occurs, and then steps repeatedly until the configuration stabilizes, i.e., the steps are empty. When a statechart steps multiple times in a single unit of time it performs a *superstep*. STATEMATE also features additional commands to fine-tune its stepping behavior. [12]

Remaining Problems Some problems remain from [11]. Like in many concurrent systems, race conditions can occur in statecharts. When multiple actions from a single superstep modify the same property the result is not clearly defined, as is the order in which parallel components are executed. When CTs with the same scope are in conflict the result is undefined. [12]

3. Related Work

Statecharts are a part of the UML specification in the form of behavior state machines. UML behavior state machines extend statecharts with actual pseudo-states in which the system cannot stop, in addition to initial pseudo-states. Initial pseudo-states are the default state of a superstate and, when entered, immediately forward to their associated single transmission. [27, Section 14.2.3.2]
Another standard for statecharts is SCXML. SCXML is a XML standard for serializing statecharts. A SCXML file contains all data a statechart requires, including e.g., states, transitions, actions, and guard expressions. SCXML does not support the concept of pure pseudo-states, only history states and initial pseudo-states are included. [36]
In [14], E. Höfig examines executing behavior models, e.g., statecharts, directly and compares it to compiling the behavior models into program code regarding performance. The author comes to the conclusion that executing statecharts is sufficiently fast for some of the uses cases presented in [14] but compares unfavorably with compiling regarding performance. Executing statecharts directly is between 3 to 460 times slower than executing the compiled statechart. This trade-off may be balanced by the possibility to adapt a statechart at runtime [14], but this possibility is not examined in this thesis. As such, when maximum performance is required the engine developed here likely cannot compete with compiled code.
In the case the statechart is compiled, [7] presents multiple optimization techniques for event handling in statecharts, e.g., when dealing with event priorities and thread counts. It determines that reducing the thread count where possible and avoiding unfiltered global event broadcasting are effective at decreasing the event management overhead.
[4] presents the DEAL system, which uses statecharts to create convincing interactive dialogue in a serious game. The statecharts in the form of SCXML files models the dialogue of a merchant with whom the player haggles for an object. The system combines these statecharts with natural language recognition to allow immersive interactions with non player characters. While the use case of interactive dialogue is not considered in this thesis it is nevertheless an area of games where innovation is currently rare; and the system presented is another example for the benefits gained from statecharts in games.
An application of statecharts in games which is presented in this thesis is AI. [9] examines the reusability of AI behavior across games by using statecharts. Since their hierarchical nature splits statecharts into smaller components these components can be modular and can be reused across multiple projects. The paper identifies some problems with unfiltered global broadcasting as events can be received by the wrong statecharts.

4. Engine Features

4.1. Use Cases in Games

Before implementing the engine, some considerations were necessary. Which features of the original statechart paper are going to be implemented? How to resolve ambiguities in the semantic of a statechart? How and where to interface with Unity? The use cases were of great value here as they showed which features would be commonly used in game development. Statecharts are not limited to these use cases, an example for another use case can be found in [4].

Player Character Controller Figure 4.1 shows an example for a character controller. The character controlled by it can walk, run, and sprint. It can also slide on the ground for a limited time and dodge incoming attacks. It is able to jump and can slow its fall when aiming, it can also doge when it is falling or jumping. Its statechart contains multiple levels of hierarchy and initial pseudo-states. Its transitions are triggered both by events and by default; they also often feature guards.
While character controllers influence both rendering and the physics simulation of game engines, contrary to rendering engines many physics simulations become unstable when modifications are not synchronized; therefore the statechart should be synchronized to the physics engine of the game engine. The number of character controllers which have to be computed is often limited to less than a hundred, and since players often control instances of similar or even the same character the statecharts in use are also the same which presents opportunities for optimization. In single player games the character controller is spawned once at the start of the game or level; in multiplayer games character controllers may be spawned more often but still less frequent than e.g., AI.

Character Decision AI The statechart in figure 4.2 depicts a primitive AI for a shooter. The AI walks aimlessly until it finds clues which point to the presence of an enemy. It then investigates these clues while staying vigilant. Once it spots an enemy, it tries to kill the enemy by gunfire it it can see the enemy. It the enemy is not visible, it employs suppression fire to try to pin the enemy to a position while approaching carefully. The statechart again features a multi-level hierarchy; its transitions are mostly triggered by events.
Character decision making does not need to be synchronized to the rendering or physics engine since it at most reads from these engines. Characters in multiplayer games however need to make the same decision in every game instance, as such the statechart must be synchronized across all game instances. At the same time, a limited step frequency may resemble a human reaction time, a property which can be desirable to simulate in a game. Depending on the type of the game, a large count of characters has to be simulated therefore the computation time for a step needs to be as low as possible. In many cases, many of these characters share the same AI so the statechart can be reused between these

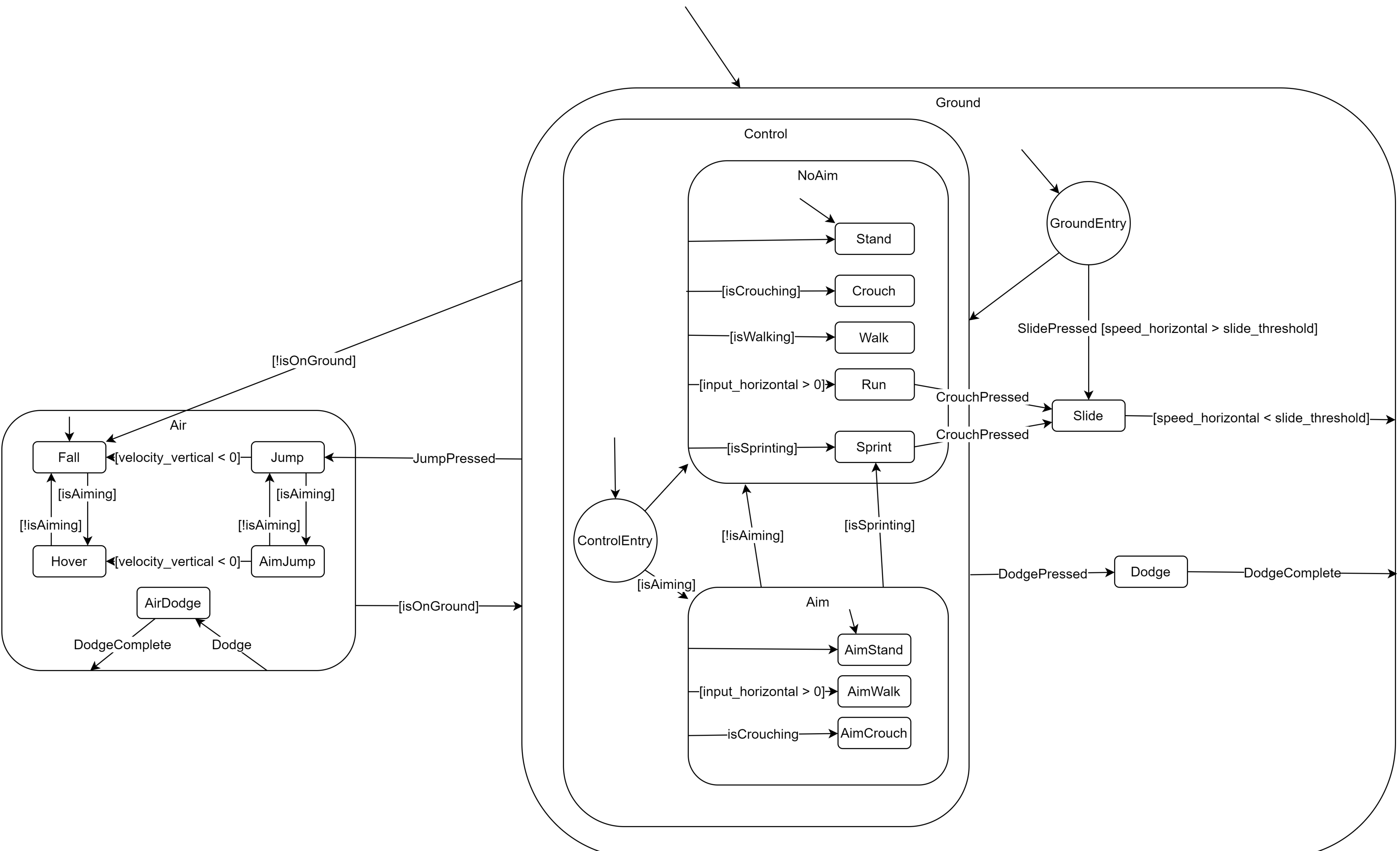

Figure 4.1.: Example statechart for the character controller use case

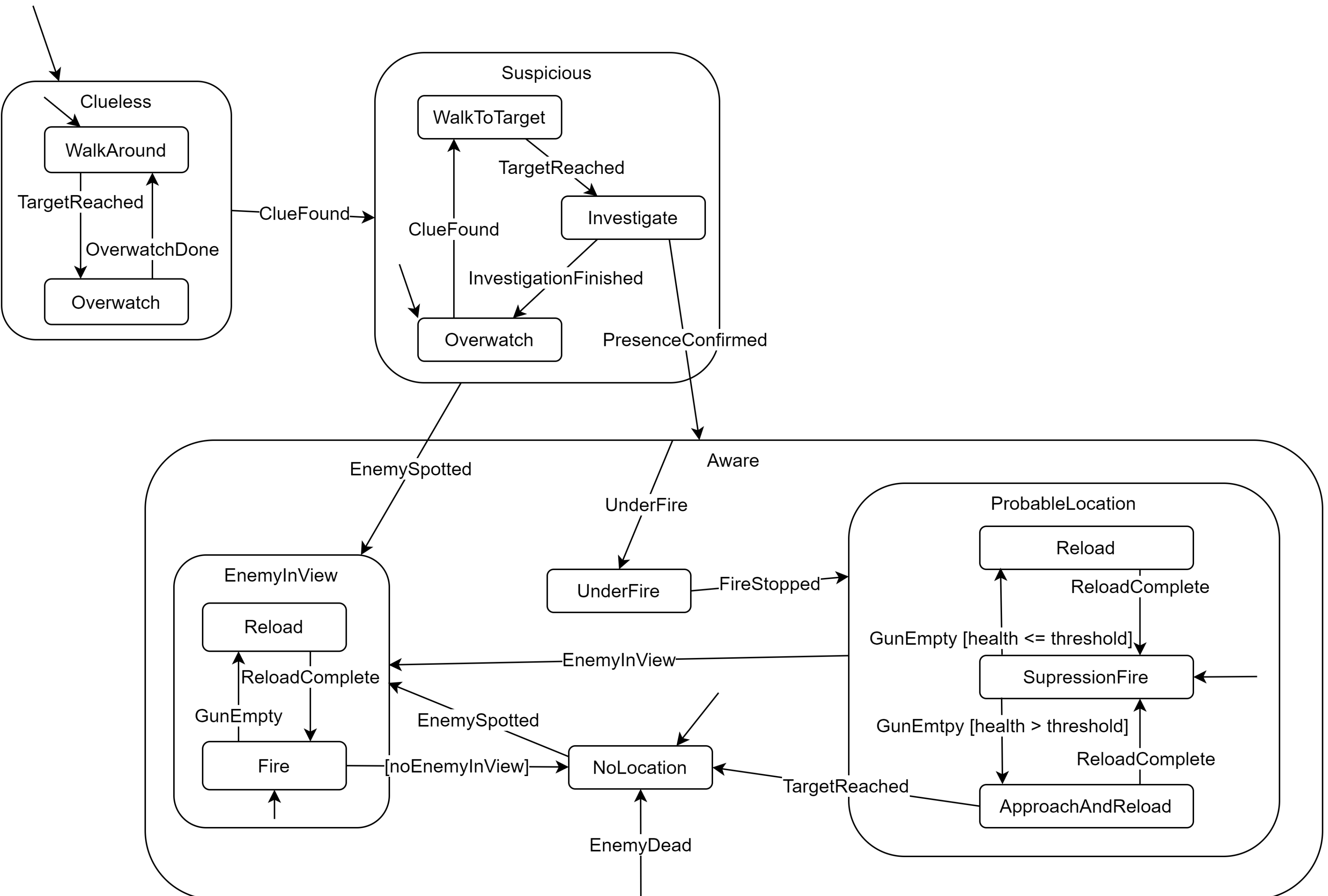

Figure 4.2.: Example statechart for the character decision AI use case

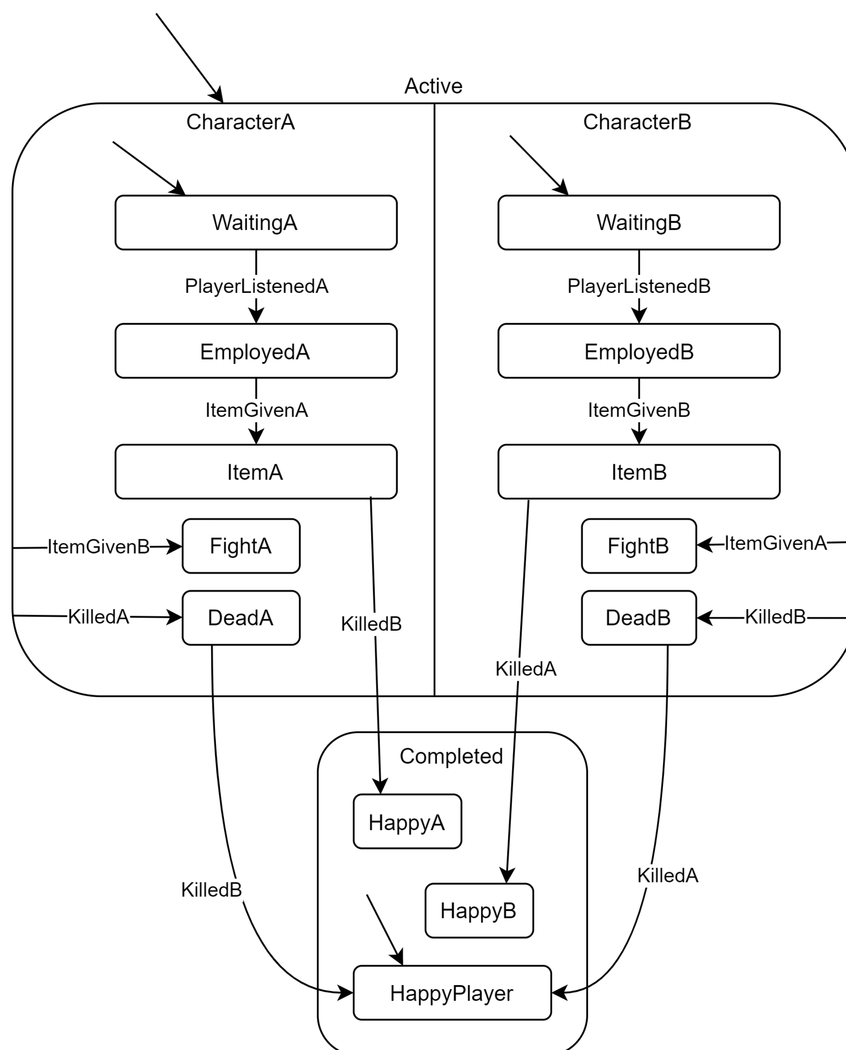

Figure 4.3.: Example statechart for the interactive narrative use case

characters. Instancing time is also important in the case a game featuring many characters loads a level or some player action results in an entire army spawning.

Interactive Narrative An example narrative is depicted in figure 4.3. The narrative involves an item which is wanted by two characters. The player can choose who to give the item; they can also keep the item themselves which requires killing both characters. The statechart consists of a parallel state controlling the states of the characters involved and a compound state which contains the final outcome of the narrative. Its transitions are exclusively triggered by events.
Narratives, in games often called missions or quests, again do not need to be synchronized to anything. Changes in such narratives do not happen often enough to warrant synchronization with the physics or render engine. Hence, it is sufficient when the statechart steps when an event occurs. The events the statechart reacts to are sparse, therefore the statechart rarely steps. Computing the step can even be done asynchronously since latency is less important in this use case. However, the number of unique statecharts

active at any time may be large. Statechart initialization time is also not important; the statechart is initialized at game startup and is then active until the game is finished.

4.2. Statechart Features to Implement

The use cases determine the statechart features included in the engine: All use cases use hierarchy of at least one level. The interactive narrative use case uses orthogonality to enable concurrent state changes. The character controller use cases uses initial pseudo-states to instantly determine the correct state for the character. While actions are not included in the statecharts they are a necessary feature for the engine to allow interfacing with external code. The use cases also use event triggers and guards extensively.
The engine therefore must support hierarchy, orthogonality, (initial) pseudo-states, actions, event triggers, and transition guards.

4.3. Semantics

The engine follows the semantics in [12] closely but to remove ambiguities when executing a statechart the engine includes the following changes: In the semantics multiple transitions originating from a single state can be active at once, the same problem exists in hierarchies. To remove any ambiguities in the single state case, each transition in executed statecharts must have a unique per state priority which is implicitly given by the order in the XML file. Similarly, the order of execution for the components of parallel states is not defined. In this case the order in the engine is depends on the implementation of C# HashSets.

4.4. Unity Interface

Using Unity also suggested using C#, which is well integrated into the engine [32]. C# is designed for object oriented programming, as is the standard Unity scripting system.
To mesh well with the Unity workflow, implementing the engine within the standard Unity scripting system with the name MonoBehaviour was an obvious choice. A script extending the MonoBehaviour class can be added to any object in a Unity game scene, similar to a collision shape, rendered mesh or rigidbody. The component can then interface both with other scripts on this object and with components native to Unity. A statechart instance is implemented as such a MonoBehaviour component. MonoBehaviour components can also implement methods which are called by Unity, e.g., once per frame rendered, once per physics step or when the component is first loaded. Once implemented this behavior can not be deactivated. Therefore, even when the methods predictably do nothing they are called and hinder performance. For this reason a single component spawned by the instances on demand handles these methods for all instances.

Loading a Statechart Statecharts are a new Unity asset introduced with the engine by using [34]. This statechart asset consists of both a text asset in which a statechart is serialized and some settings, e.g., a synchronization mode. The asset can be assigned to a statechart instance either in the editor or by code. If no asset is assigned the instance

raises an error. The engine performs no validity checks on the statechart on load or execution.

Synchronizing with the Unity Game Loop As seen in the use cases, the instances need to be able to step at several different moments in the Unity game loop. For this purpose, the following modes of synchronization are available per statechart:

- Unity Update
  The instance steps whenever Unity renders a frame. The order in which objects and in extent their components and the centralized engine are called depends on Unity. The order in which the instances themselves are called is undefined.
- Unity Late Update
  The instance steps whenever Unity renders a frame after the Update() method of all objects has been called. It otherwise behaves like Unity On Update.
- Unity Fixed Update
  The instance steps whenever the physics simulation advances. The exact moment within a physics step again depends on Unity. The order in which the instances themselves are called is undefined.
- On Event
  The instance steps whenever a statechart event is handed to it.
- Manual
  The instance steps whenever its SuperStep() public method is called.

External Input For the statecharts to be useful external input is needed, which comes in two forms. External code can hand custom events to a statechart instance which trigger transitions when the instance steps. Transitions can also contain guards, which need of an external value to evaluate. These values are relayed by external code to the instance and inserted into a data structure. From there they are accessed by the transition when needed.

Statechart Output Once a new configuration is computed, all corresponding actions are executed. Multiple methods for how these actions trigger external functions exist within C#. For example, it is possible to connect arbitrary external functions to internal triggers using reflection. The actions withing the statechart file contain the names of the functions they trigger. Using this name, an instance then searches through other components on its object to find matching functions and connects them. This method allows actions to only be executed when they are connected, resulting in potentially increased performance. Unfortunately, it also limits statechart reusability by binding actions to specific functions on a statechart level. Additionally, support for reflection is limited on some platforms Unity can export to [33]. For this reasons the engine ultimately uses the subscriber pattern.
External code hands the engine a reference to a method to execute when a specified action occurs. Once an instance steps, all actions are triggered and the connected methods are executed. This approach allows any code running in a scene to subscribe to any instance's actions. A method can be unsubscribed from an action in a similar way.

4.5. Terminology

The remainder of this thesis contains some previously not mentioned terms. The term *node* covers states and pseudo-states and describes a connection point for transitions. States are *explicitly* exited when they are superstates of the source of a transition. States are *explicitly* entered when they are the superstates of a transition's destination or they are the states entered when the transition's destination is a parallel state. A state is *implicitly* exited or entered when it is not explicitly entered or exited but entering or exiting is required to arrive at a valid configuration. A *property* is an externally set value which is evaluated in a guard.

4.6. File Format and File Requirements

The engine uses a custom XML format derived from SCXML. It only supports states, parallel states, and transitions. Additionally it supports pseudo-states which are designated with the node name *pseudo*. All states in the XML file must have a unique *id* attribute. All non-parallel states which contain at least one component must include a default state using the *initial* attribute. Transitions must include an *event* and a *destination* attribute containing the trigger type and destination state id. Optionally, transitions can have a *cond* attribute which contains the name of a property which is used as guard.

5. Implementation

5.1. Performance Test Methodology

All tests were done on the same computer with the following specifications:

- CPU: *AMD Ryzen 9 3950X*, featuring 16 cores and simultaneous multi-threading, resulting in 32 threads. It computes with a minimum frequency of 3.5 GHz, which it increases to 4.7 GHz depending on its temperature.
- RAM: Four times *G-Skill F4-2133C15-8GRB* for a total of 32 GB of RAM, running at 2133 MHz
- CPU-Cooler: *Corsair iCUE H150i RGB PRO XT* all-in-one liquid cooler mounted in the top of the case.
- Motherboard: *ASUS PRIME B550-PLUS*
- Case: *be quit! Silent Base 801*
- Operating system: *Microsoft Windows Home 64 Bit* 20H2 (Build: 19042.867)

Unity was set to the Mono scripting backend for all tests. The Mono backend uses just-in-time compilation which may influence the first sample of any test [32].
All measurements were done using the C# Stopwatch class [24]. On the test computer, Stopwatch measures time spans with a precision of 100ns. According to tests, using the Stopwatch class removes any measurable measurement overhead. The measurements start immediately before the code of interest and stop immediately after.
The engine generated at least 10.000 samples per run. Minimum, maximum, and mean of all samples were determined. In addition, for the last 5000 samples of any run the minimum, maximum, mean, and median were determined separately. For every sample an instance receives ten random events from a list oft recognized events; additionally, twelve random properties are set per sample.
All tests performed are only a limited simulation of a real workload. The amount of states, transitions, and added events may differ greatly from the values used in the tests which will lead to different results. Figure 5.1 shows the statechart used for measuring the execution time of the engine. It contains both compound and a parallel state in a five level deep hierarchy; and its transitions feature a mixture of events and guards. It should therefore be sufficient to represent the complex statecharts used in games.

5.2. Naive

The naive implementation is feature complete but ignores any performance aspects; it serves to evaluate any optimizations made. The naive implementation also fixates the interface for the software.

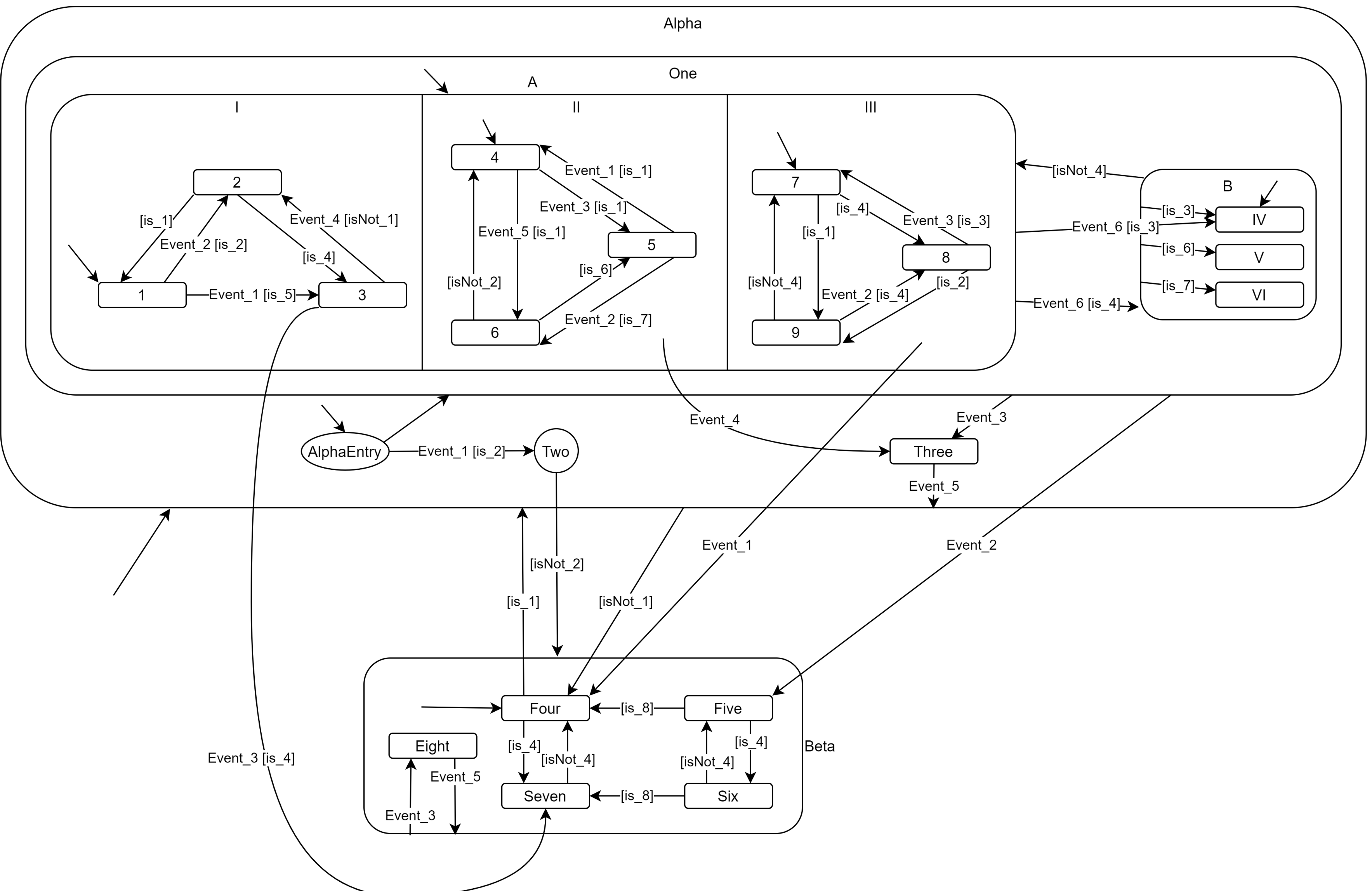

Figure 5.1.: Statechart used for testing the engine

5.2.1. Public Interface

Statecharts

The statechart asset exposes some properties to the editor. First, a Unity text asset containing a statechart in the form of an XML file. Second, the synchronization mode which will be used for all instances of this statechart. And third, the maximum number of steps a superstep may perform. This is useful to remove spikes in the execution time and to avoid infinite loops.
The statechart asset does not contain any methods designed for public use, nevertheless some methods are exposed out of necessity. The `Instance()` method converts the XML file into a custom memory format (it this has not been done already) and then creates and returns the default status for this statechart.

Instances

The statechart instance component exposes a field for a statechart asset in the editor. This facilitates an editor-based workflow and removes the need to add instances with scripts. When an instance is created from code, the `Initialize(statechart)` method must be called directly after. The method also initializes the instance to the default configuration for the given statechart. The `SuperStep()` methods performs repeated steps either until the step limit given in the statechart asset is reached or a step is empty. It also reports whether the superstep was empty itself. Any script can add events to the instance using the `AddEvent(event)` method. If the statechart's synchronization mode is On Event the statecharts immediately performs a superstep. To be notified when an action occurs, scripts can subscribe an event handler to an action with the `Subscribe(name, type , event handler)` method. It requires the name of the statechart element (node or transition), the type of action to subscribe to (exit, stay, passthough or enter), and the event handler. The `Unsubscribe(name, type, event handler)` method unsubscribes an event handler from an action. Properties for use in guards can be set or read using `SetProperty(name, value)` and `GetProperty(name)`. `IsStateActive(name)` returns whether the state with the given name is currently active.

5.2.2. Central Classes

Statechart The `Statechart` contains lists of statechart elements and can parse an XML file into these lists. It does so by first parsing all states in the statechart and then parsing all transitions. It can report some basic errors in the file such as missing defaults states in compound states and missing attributes in transitions. It does not explicitly report duplicate states or referenced states which do not exist.

Statechart Elements The custom memory format for statecharts consists of nodes and transitions which both inherit from the empty interface `ISCElement`. Nodes are then split into states, which can be active, and pseudo-states, which cannot. States are again split into the types basic, compound, and parallel.
All nodes contain a method which tries to enter them: It returns the default basic states which are active when the node is active or, in the case of pseudo-states, forwards to the next transition. All states also contain a method to try to exit them: It checks every

transition in the current hierarchy of states, starting with the root. Transitions similarly contain a method which tries to traverse them. All three methods can fail if a necessary transition is not active, in this case they return an empty set. All methods also record the elements they traversed which are later used to send the corresponding actions.
Nodes also contain some utility methods for computing both the list of superstates and the common superstate for a set of nodes. When the name of a node is accessed it generates a name consisting of all names of its hierarchy, e.g., the name of the node A with the superstate B (itself with the superstate $Root$) is $Root.B.A$.

StatechartInstance The `StatechartInstance` inherits from MonoBehaviour and is therefore a Unity MonoBehaviour component. It contains a reference to its statechart, its status and a dictionary which connects statechart actions to external functions. The interface class performs steps autonomously when in the OnEvent synchronization mode, the step can also always be executed manually. In other synchronization modes the instance will autonomously register itself in the statechart engine when enabled, and deregister itself when disabled. The statechart can log every internal and many external actions to the Unity log output, this must be activated using the preprocessor directive `SC_LOG_FUNCTIONALITY` in the Unity project settings.

Status The `Status` contains the basic configuration, events, and properties of a statechart and allows access to them. It also contains a reference to the statechart it belongs to. Using this reference a status can validate itself, i.e., it verifies that its configuration is valid for the statechart. This is not done in the normal operating mode and must be activated using the preprocessor directive `SC_DEBUG`.

CT The CT class represents a full compound transition. It contains its source state, destination states, and all elements (nodes and transitions) traversed. The CT class also computes all its explicitly exited and entered states. It implements the IComparable<CT> interface [20] and can be sorted by scope, which is used in the step algorithm.

StatechartEngine The `StatechartEngine` is a singleton which is automatically created whenever a statechart instance is created. It contains a list of statchart instances per synchronization mode. It iterates over these lists and executes the statecharts at the corresponding points in the Unity main loop.

SCEvent An `SCEvent` represents an internal or external event the statechart can react to. It only contains its type in form of a string. The class overloads all equality operators and methods for easy comparison.

Guard The `Guard` class represents a transition guard. It contains a property name which is used to retrieve a boolean value from a statechart status. Currently this summarizes the entire functionality but the class can easily be extended to support full expressions for arbitrary property types.

5.2.3. Step Algorithm

The step algorithm in the instance class executes a single step in the statechart. It is separated into four steps:

Prepare The algorithm prepares data structures for later use. It creates an empty list of CTs and fills a set of all currently active states. It also initializes sets for all entered and exited states.

Search The algorithm searches for active full CTs in all active states and adds them to the list.

Validate First, the algorithm sorts the found CTs by scope from high to low. Then it computes preliminary entered and exited states for each CT and adds them to the sets. If a CT is in conflict with a CT with a higher scope the Ct with the lower scope is discarded. The algorithm then removes any remaining states of which a superstate was exited before. Finally, it enters all remaining parallel components which must be active for a valid configuration.

Execute The algorithm updates the basic configuration in the status. The algorithm then executes the ENTER, EXIT, STAY, and PASSTHROUGH actions for the computed sets. After clearing the list of events it finally adds any generated events to the status for the execution of the next step.

5.2.4. Performance

Table 5.1.: Last 5000 easurements of the step algorithm in the naive implementation in the editor (in µs)

	Minimum	Maximum	Mean	Median	Median/Mean
Prepare	4.1	21.3	4.8	4.5	93.8%
Search	2.8	1939.0	5.3	4.4	83.0%
Validate	5.3	1891.5	15.9	14.0	88.1%
Execute	5.6	3263.5	15.1	11.6	76.8%

Table 5.2.: Last 5000 measurements of the step algorithm in the naive implementation outside the editor (in µs)

	Minimum	Maximum	Mean	Median	Median/Mean
Prepare	1.4	237.9	2.3	2.0	87.0%
Search	0.9	133.6	2.1	1.7	81.0%
Validate	1.7	232.9	6.9	4.9	71.0%
Execute	2.2	251.7	6.6	4.3	65.2%

Step Algorithm Table 5.1 shows the measurements of the step algorithm when running in the Unity editor. It is quickly visible that both mean and median of the last 5000 samples are closer to the minimum than to the maximum. Meanwhile the mean and median

are still close. Combined with the high values in the maximums, with the exception of the low value in the prepare maximum, these observations lead to the conclusion that the C# garbage collector likely interferes with the measurements but, since these extremes appear to be rare, this does not impact the median severely. Still, spikes in the execution time may lead to unpleasant spikes in the frame time.
Table 5.2 shows the same measurements when the project is built and runs outside the editor. The values show that the execution time is much lower in this case, especially the maximums. Despite that, the median deviates slightly more from the mean than in the editor measurements, indicating a more skewed value distribution.
Both tables show that the least amount of time is spent searching for active CTs while executing the CTs is the slowest part of the algorithm. Were functions subscribed to the actions generated by the step the execution time of theses functions would have to be added to the execute measurements. Despite this being not the case in the test scenarios the execute part of the algorithm takes longer to compute than any other part. It and the prepare and validate parts therefore provide the largest potential for performance gains.

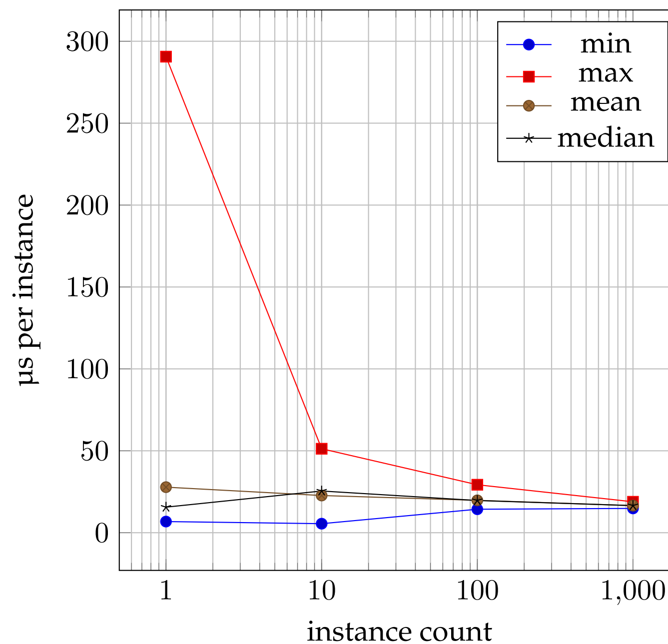

Figure 5.2.: Time per instance in µs for the naive engine when built

Table 5.3.: Last 5000 measurements of the engine in the naive implementation outside the editor when executing one instance (in µs)

	Minimum	**Maximum**	**Mean**	**Median**
Engine measurements	6.8	290.6	27.8	15.6
Sum of the algorithm measurements	6.2	856.1	17.9	12.9

Engine Figure 5.2 shows the execution time per statechart instance for 1, 10, 100, and 1000 instances in the last 5000 samples of the run. It shows that minimum, mean, and median are largely constant whereas the maximum falls quickly as any outliers are smoothed out by the larger total execution time. The mostly constant values indicate an expectedly low engine overhead. For large instance counts all values converge to a single value. At

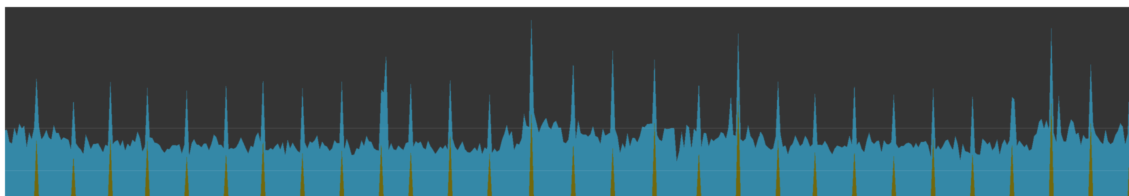

Figure 5.3.: Screenshot of the Unity profiler showing the script (blue) and garbage collector (brown) execution time of the project in the naive implementation when built.

a mean of 16.5361 ms the engine barely fulfills the goal of 1000 staecharts within 16.6 ms already, but currently there is no time left for rendering or other calculations.
Table 5.3 shows the execution time for a single statechart when measuring the engine and the sum of the measurements of the algorithm. The large discrepancy in the maximum case indicates that maximums in the algorithm measurements do not appear together. The still noticeable deviation in the mean and median is surprising though since the minimum values are close. Figure 5.3 shows a screenshot of the builtin Unity profiler measuring the script and garbage collector execution times of the engine running a single instance outside the editor. While theses values include the test script used to generate events and change properties and the usual profiling overhead they clearly show that the spikes in the script execution time mostly originate from the garbage collector. Screenshots showing methods calls from both a low and a high execution time can be found in Appendix B.

5.3. Reducing the Workload

This section discusses how the execution time for the step algorithm can be reduced using various techniques.

5.3.1. Precomputation and Caching

The engine does not need to spend time on computing information during the runtime which can be either computed beforehand or remembered from the last time it was computed. This may increase the memory footprint of the software which is a common tradeoff. However, the rising fidelity of game graphics and audio and the accompanying increase of the memory footprint dwarfs the increase caused by precomputing or caching in the software. Though as section 5.3.2 shows an increase in memory can lead to a reduction of performance which the following exaggerated example depicts: Computing the prime factors of a large number is a notoriously intensive problem in computer science. Reading the prime factors from memory or even retrieving them via the internet will generally be faster than computing them given a large enough number. In contrast, adding two small numbers is barely slower than even reading the result from a CPU register.

Precompute

As the performance results of the step algorithm show the validate part is a candidate for precomputation. The algorithm currently must compute the exited and entered states of every CT. Before that, the algorithm sorts the active full CTs by scope. Computing the scope is another intensive calculation. Therefore, replacing the basic transitions currently in use with precomputed full CTs containing all exited, entered, and visited nodes and its scope would remove these computations from the runtime of the engine. These computations can be done when building the project.
A problem immediately arises: Computing all full CTs containing a pseudo-state leads to a large increase in the number of CTs. Given a pseudo-state with x entering and y exiting transitions results in $x * y$ full CTs. This number increases exponentially with every pseudo-state connected. Not only must all these CTs be saved in memory but they must also be searched for every step the originating state is active in. Additionally, including history states in the software would make precomputing the entered states of a CT impossible since the target of a history state changes at runtime.
While these issues can be solved by including both precomputed full CTs and basic transitions and choosing between them at runtime, the work required for this moves the cost-benefit ratio beyond that of other optimizations.

Cache

A similar but easier approach is to still compute the required data at runtime but to avoid repeated computation of this data for repeated access. This can be done for the initial status of a statechart, the scope of a full CT and the full names of nodes. The initial status is required every time the statechart is instanced. Caching it should lead to reduced initialization time for an instance, which is not measured. The scope is required for every comparison between two CTs in the sorting phase of the step algorithm, caching it should result in a reduced mean time of the validate part of the step algorithm. The full names of nodes are required for executing actions and generating internal events, caching them should result in a reduced mean time of the execute part of the step algorithm.

Results

Table 5.4 shows the results of the caching efforts. As expected, the mean of the prepare and search parts of the step have not changed. The mean of the validate parts is reduced but the change is low which implies that the number of active CTs is often so low that sorting them does not involve many comparisons. The biggest gains occur in the execute part of the step where the number of string concatenations is reduced.

5.3.2. CPU Cache Optimizations

With rising CPU speeds memory was left behind. Accessing a random location in main memory can take 50-100 ns according to [13, Chapter 2.1]. In this time, the CPU of the test computer performs approximately 200 to 400 cycles when running at 4 GHz. Faster memory costs more to produce and needs more power which must be dissipated. To mitigate this issue modern desktop CPUs have a memory hierarchy consisting of the CPU registers with instant access, a L1 cache with a latency of a few cycles, a L2 cache

Table 5.4.: Last 5000 measurements of the step algorithm outside the editor with caching and also the difference to the naive implementation (in µs)

	Minimum	Maximum	Mean	Median
Prepare previous	1.4	237.9	2.3	2.0
Prepare	1.5	152.2	2.3	2.1
Difference	+0.1	-85.7	0.0	+0.1
Search previous	0.9	133.6	2.1	1.7
Search	0.8	160.6	2.1	1.8
Difference	-0.1	+27.0	0.0	+0.1
Validate previous	1.7	232.9	6.9	4.9
Validate	1.8	276.9	6.4	4.7
Difference	+0.1	+44.0	-0.5	-0.2
Execute previous	2.2	251.7	6.6	4.3
Execute	1.5	150.4	4.6	3.3
Difference	-0.7	-101.3	-2.0	-1.0

with a latency of 3-10 ns, a L3 cache with a latency of 10-20 ns and the RAM with the aforementioned 50-100 ns [13, Chapter 2.1]. The sizes of these layers in the CPU of the test computer are 1 MB L1 cache distributed across all 16 cores, 8 MB L2 cache also distributed across all cores, 64 MB L3 cache shared between all cores and finally the RAM with a total size of 32 GB.
When a memory location is accessed its cache line containing a number of bytes is loaded into the cache. Accessing another memory location already loaded into the cache because it also is contained within this line then results in a reduced access latency. Therefore, if data is small more of it is contained in a single cache line and more accesses to this data result in a cache hit as opposed to a cache miss. For this to work the data must be contained within consecutive memory locations which limits using data structures with a variable capacity such as linked lists. Arrays or array based data structures are suited for this purpose. The C# library data structures List, HashSet, and Dictionary belong to this category [18] [21] [17].

Changes to the Software

In the naive implementation, despite being contained in a List, the statechart data is split into many small objects which may be distributed over the entire available memory. This is because a class is a reference type in C#, which means that the data of the class is allocated in a large enough memory location while the list contains only a reference to this location. C# also features the structure type for data. Structures are a value type, i.e. their content is located where a variable exists. This also means that structures themselves cannot be contained within more than one object and passing a structure as a method parameter instead copies the structure. Structures also cannot inherit from other structures, therefore a hierarchy of references like in the naive implementation is not possible with structures. [25]
To optimize the statechart for the cache means reducing the size of the elements and moving them to continuous memory. However, statecharts are a multidimensional structure which cannot be flattened to one-dimensional memory in an optimal way. The state hierarchy in statecharts forms a tree while the net of states and transitions forms a graph.

Approaches on optimal memory layout for trees [2] and for graphs [1] exist but are not designed for the access patterns in statecharts. [2] optimizes trees for root to leaf searches which are not used here. [1] optimizes graphs for general accesses but the breadth-first-search recommended does not work for hierarchical graphs.
In the optimized statechart both nodes and transitions are structures. The characteristics of structures mean that instead of nodes containing direct references to their neighbors they contain their index in the list of states. The destination in transition functions similar. Saving the components of states required for parallel states (and validating a status) is more problematic since the number of components is variable. If the components of a state follow immediately after a state in the node array only saving the component count would be possible if it were not for the components of these components. Instead the indices of the components of a state are saved into a different array while the state itself only contains the index of the first element in this list. Since the order of components follows the order of nodes in the array the component list of a node ends where the component list of the next node begins. The mapping of transitions to states follows the same principle. Compound states additionally require an index for their default state. Additionally nodes must contain their type, either basic, compound, parallel, or pseudo-state. Using 32 bit integers for all indices and assuming 32 bit for the type enum as well results in a total structure size of 160 bits or 20 bytes per node.
Similarly, transitions are structures too. A transition still contains its trigger, a guard, and its destination. Assuming a 64 bit environment this results in 64 bit for the trigger, 64 bit for the guard and 32 bit for the destination, or in total also 160 bits or 20 bytes.
In addition to the list of nodes, list of components, and list of transitions the statechart also requires a dictionary mapping strings to node indices to make the `IsInState()` method of the instance class possible. The statechart also contains arrays mapping node and transition indices to their names for debugging and logging purposes. All methods previously contained in the various statechart element classes are now located in the statechart class.
Since the status already uses continuous data structures only the configuration changed from state references to integer indices.

Results

Since the search part of the algorithm depends on the structure of the statechart the most, the biggest performance gains were expected here. Table 5.5 shows that this is not the case. Since all data for the statechart was previously allocated in immediate succession it is possible that it was already laid out in an efficient way. Again, most gains appear in the execute part but all parts of the algorithm perform faster. A possible reason for this could be the hash-function of the HashSets used in the algorithm. Since the Node class does not contain a custom hash function the set uses the default hash function of C# object. This function cannot be faster than converting a 32 bit integer index into a 32 bit hash integer since the integer is its own hash in this case.

Table 5.5.: Last 5000 measurements of the step algorithm outside the editor with cache optimizations and the difference to the previous implementation with caching (in µs)

	Minimum	Maximum	Mean	Median
Prepare previous	1.5	152.2	2.3	2.1
Prepare	1.2	126.0	1.9	1.7
Difference	-0.3	-26.2	-0.4	-0.4
Search previous	0.8	160.6	2.1	1.8
Search	0.5	146.2	1.8	1.7
Difference	-0.3	-14.2	-0.3	-0.1
Validate previous	1.8	276.9	6.4	4.7
Validate	1.2	226.6	4.9	3.9
Difference	-0.5	-50.3	-1.5	-0.8
Execute previous	1.5	150.4	4.6	3.3
Execute	1.2	222.1	2.4	1.7
Difference	-0.5	+71.7	-2.2	-1.6

5.3.3. Other Optimizations

Remove Dictionary Lookups

Since C# dictionaries are hash maps the lookup operation complexity approaches $O(1)$ [17]. Still, the string hash function which must be computed for a lookup can be optimized away. Instead of guards containing the name of a property as a string, and having to perform a dictionary lookup every time the guard is evaluated, the guard contains an integer index pointing into an array of property values contained in the status. The lookup is then only necessary for either setting or reading the property from external code. The dictionary mapping the name to the index can then be moved to the statechart which also reduces the memory footprint of the status.

Remove String Comparisons

Another area with potential for optimization are the string comparisons performed when events are compared. Instead of storing the type of an event as a string, the string is hashed and the resulting integer is stored. While the computation of such a hash is potentially slower than copying the string this operation is performed once per event while the type is accessed and compared for every transitions searched. Unfortunately the hashes produced by the functions in C# cannot be assumed to be collision free [23]. A collision in this case would mean that different event types produce the same hash value and are therefore equal according to the comparison. This could lead to rare but severe failures of the software and would be hard to recognize by users. Hence, the software does not include this optimization.

Reduce List Allocations

A step uses many data structures of the types List and HashSet. These data structures resize themselves at runtime when their capacity is insufficient for new elements. While

the resize operation amortizes itself over all insertions [5, Section 17.4] it still adds to the insertion time, and it is performed often in the case of fresh structures containing only a few elements. To reduce the time spent on resizing a list it can be initialized with sufficient capacity from the start. For HashSets this is not possible in .Net Standard 2.0 [29] which is recommended for libraries such as the software developed here [32].

Garbage Collector

As the measurement results of the naive implementation indicated, the C# garbage collector produces spikes in the runtime of the software when it interrupts its execution [31]. The garbage collector frees memory which belongs to objects which are no longer in use [3]. To avoid the spikes produced by the garbage collector the number and size of memory allocations can be reduced. To do this it is possible to reuse objects or memory, e.g., by using an object pool [26, Section 19]. Due to the cache optimizations in section 5.3.2 most temporary objects in the software are now value types which are not allocated on the heap. However, the collections used to store these objects are reference types and are therefore cleaned by the garbage collector. Instead of creating several temporary data structures for every step the algorithm now reuses structures created at instance creation time. These collections must be cleared or emptied before they can be reused which takes time depending on the number of objects within the collection [22] [19].
To reduce the number of allocations further, both SCEvent and the internal Action class are short lived data types and were converted to structures.
The changes outlined here have a side effect: Some of the HashSets are reused over many steps, and since their capacity is not reduced by clearing them this means that resizing operations are mostly limited to the early steps of an instance.

Results

Table 5.6.: Last 5000 measurements of the step algorithm outside the editor with all optimizations and the difference to the previous implementation with cache optimizations (in µs)

	Minimum	**Maximum**	**Mean**	**Median**
Prepare previous	1.2	126.0	1.9	1.7
Prepare	0.8	25.1	1.5	1.5
Difference	-0.4	-100.9	-0.4	-0.2
Search previous	0.5	146.2	1.8	1.7
Search	0.4	130.5	1.6	1.6
Difference	-0.1	-15.7	-0.2	-0.1
Validate previous	1.2	226.6	4.9	3.9
Validate	1.1	227.7	4.0	3.5
Difference	-0.1	+11.0	-0.9	-0.4
Execute previous	1.2	222.1	2.4	1.7
Execute	1.0	234.5	1.8	1.4
Difference	-0.2	+12.4	-0.6	-0.3

The reduction in dictionary lookups should be visible in the measurements of the search parts. The results in table 5.6 show a performance gain not only in the search step but

also in most other areas. This likely stems from the reduced list resize operations and reduced object allocations. A reduction in the length or frequency of garbage collector runs which would show itself mostly in lower maximums is not clearly visible.

5.4. Increasing the Workforce

While CPUs have become faster with every generation [6] the performance gains cannot be fully exploited with a single-threaded program as the test CPU shows: It contains 16 physical cores with simultaneous multithreading and therefore requires 32 threads for optimal utilization.
Fortunately, assuming a large number of instances, the statechart engine is close to perfectly parallelizeable. All instances are independent of each other as long as actions are not connected to functions which write data to or read data from other statecharts. The instances themselves only read from the shared statechart.
In this problem each statechart instance forms a task. These engine must schedule these task for the different threads. There are two approaches to scheduling task: Static scheduling divides the task in n sets of equal sizes where n is the number of threads in use. The threads then compute the results while the main thread waits. Dynamic scheduling divides the task in batches of a fixed size of which n batches are initially distributed (n again is the number of threads in use). Whenever a thread finishes its batch it fetches a new batch from the batch pool. [28, Section 2.11.4, Table 2.5]
In the case of the engine the execution times of the tasks vary wildly. When using static scheduling the engine must then wait for the slowest set of tasks to be finished to continue. This again results in suboptimal use of the available resources. However, static scheduling incurs a lower scheduling overhead than dynamic scheduling. The batch size in dynamic scheduling balances these tradeoffs and can be changed to suit the task at hand.
Another characteristic of the engine is the repeated computation. Instead of creating the required number of threads every frame and then discarding them immediately after the threads are reused. Since threads which have finished their computations cannot be restarted the threads must be active continuously. Repeatedly looping would strain the CPUs resources unnecessarily; therefore the threads wait until the next frame when no more task are available. Meanwhile the main thread waits while the worker threads complete the tasks. The concept for this stems from [16] and was then extended to support an arbitrary amount of worker threads.

Results

Despite the instances being independent of each other the implemented threading system produces race conditions, i.e., more than one thread accesses a memory location which leads to nondeterministic and erroneous behavior. In this case the batches of at least two threads overlap and they operate on the same statechart instance. At the same time this seems to lead to some threads never finishing their work and an effective program halt. While the engine contains code to detect this behavior this invalidates any attempts to measure its execution time. Furthermore, random exceptions occur infrequently. Therefore, measurement results do not exist for the threaded engine.

6. Final Results and Discussion

Table 6.1.: Last 5000 measurements of the step algorithm outside the editor with all optimizations and the difference to the naive implementation (in µs)

	Minimum	Maximum	Mean	Median
Prepare naive	1.4	237.9	2.3	2.0
Prepare optimized	0.8	25.1	1.5	1.5
Difference	-0.6	-212.8	-0.8	-0.5
Search naive	0.9	133.6	2.1	1.7
Search optimized	0.4	130.5	1.6	1.6
Difference	-0.5	-3.1	-0.5	-0.1
Validate naive	1.7	232.9	6.9	4.9
Validate optimized	1.1	227.7	4.0	3.5
Difference	-0.6	-5.2	-2.9	-1.4
Execute naive	2.2	251.7	6.6	4.3
Execute optimized	1.0	234.5	1.8	1.4
Difference	-1.2	-17.2	-4.8	-2.9

Algorithm Table 6.1 shows the improvements made during the development. On average, most gains show themselves in the execution part of the algorithm but the validate part is also considerably faster now. The least gains can be found in the search part, where most gains were originally expected due to the cache optimizations. Unsurprisingly, the minimum shows less improvements than the mean and the median in most cases. The maximum seems to still depend on the garbage collector and did not noticeably change, despite efforts to reduce it. Apparently most gains stem from precomputing the node names and the cache optimizations combined with the now unused hash functions in sets.

Still, especially in the case of the cache optimizations a much larger improvement was expected. This leads to the conclusion that the algorithm itself may be the problem here. Reducing the number of necessary computations especially in the validate part may lead to greater reductions than ordinary optimization techniques. At the same time, an algorithm which computes the correct set of valid states for a given state will always be complex thanks to the orthogonality inherent to statecharts.

Table 6.2.: Last 5000 samples of the engine at one instances when built compared to the sum of the algorithm measurements (in µs)

	Minimum	Maximum	Mean	Median
Engine measurements for one instance	4.0	257.3	17.0	11.5
Sum of the algorithm measurements	3.3	617.8	8.9	8.0

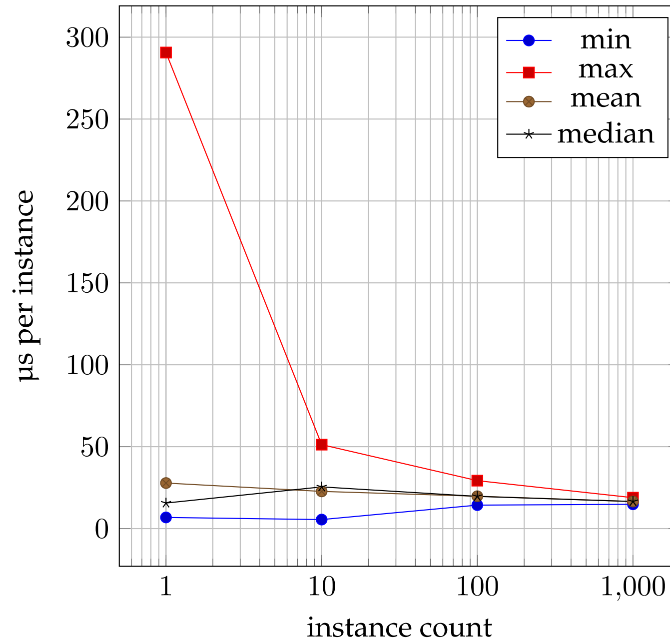

Figure 6.1.: Time per instance in µs for the optimized engine when built

Table 6.3.: Last 5000 samples of the engine at 1000 instances when built compared to the not optimized engine (in µs)

	Minimum	Maximum	Mean	Median
Engine naive	14825.9	18871.7	16536.1	16529.2
Engine optimized	7667.6	10220.0	8872.0	8874.0
Difference absolute	-7158.3	-8651.7	-7664.1	-7655.2
Difference percent	-48.3%	-45.8%	-46.3%	-46.3%

Engine The mean of the engine measurements in one instance again deviates heavily from the sum of the algorithm measurements as table 6.2 shows. The reason for this in unknown, and there is not enough data to hypothesize.
Table 6.3 compares the measurements of the optimized engine to the naive engine. It shows an execution time reduction of about 46% on average. The goal of 1000 statechart instances within 16.6 ms is fulfilled for the test statechart and some time for simple rendering and physics calculation is available.
Judging from the measurements about 2000 instances could be executed if the engine were decoupled from the Unity main thread, but this leads to major problems with race conditions and synchronization.

Problems Late during the performance measurements the logging functionality of the engine showed that the test statechart, depicted in figure 5.1, heavily tends towards the state Eight. While any reductions in the execution times can still be assumed to hold the values measured likely differ from those of a balanced statechart.

Limitations The engine has some limitations: If a parallel state contains a component whose default state is a pseudo-state and all of the exiting transitions of this pseudo-state are inactive and this component is implicitly entered by a CT, the resulting configuration

will not include this component and therefore be invalid.
The engine also unintentionally violates the semantics in one aspect: Events added by external code during a step can be sensed immediately and are also removed when the algorithm finishes. This only comes into play when the external code runs concurrently to the engine.

7. Conclusion and Outlook

This thesis documented the development of a high performance system to execute statecharts in Unity. It first showed the usefulness of statecharts in game development, then it presented some necessary synchronization modes for Unity to interface with statecharts and defined an interface to communicate with external code. Afterwards, it gave an overview on the developed engine, and presented some optimizations to decrease its performance impact in the form of caching, cache optimizations and threading in case of many instances.

Further Optimization

To increase the number of statecharts instances which can be executed at 60 fps some further optimizations are possible:
Currently, both nodes and transitions occupy 160 bytes per object (without padding). When imposing some restrictions in the state and transition count this number can be reduced by using 16 bit integers (or less) for the fields of statechart elements. Additionally, all nodes currently possess a component index, transition index and default state index even though not all states require this data. If the length of a node in the array can be made dynamic, then the size of nodes can be reduced even further. The engine could also employ data oriented design principles, which in this case may lead to major speedups by improved cache utilization across instances.
An optimization which requires users to specify the executed actions in the statechart itself can reduce the number of actions which are triggered but do not have subscribers.
While the core count of desktop CPUs and their performance still increases, most CPUs do not come close to the raw computing power of GPUs. Due to their massively parallel nature and the limits on memory bandwidth between CPUs and GPUs this computing power cannot be employed for every problem but it may be possible to translate the search, validate and parts of the execute step to shaders. The resulting jump in possible instance count would be worthwhile.

Additional Features

Some features are currently missing from the engine. It does not support history states and some other features present in statecharts, such as guard expressions. Preferably, statecharts should be parsed and validated at project built time to catch any errors early. For some statecharts additional synchronization modes may be advantageous: Scheduled synchronization executes the statechart at fixed time intervals. Delayed On Event synchronization waits a fixed interval after receiving an event to also catch any events which follow the first in a single step.

List of Figures

2.1. Transition containing both a trigger event and a guard expression 2
2.2. Compound state containing components of which one is another compound state 3
2.3. Parallel state containing two components with two basic states each 3

4.1. Example statechart for the character controller use case 9
4.2. Example statechart for the character decision AI use case 10
4.3. Example statechart for the interactive narrative use case 11

5.1. Statechart used for testing the engine 16
5.2. Time per instance in µs for the naive engine when built 20
5.3. Screenshot of the Unity profiler showing the script (blue) and garbage collector (brown) execution time of the project in the naive implementation when built. 21

6.1. Time per instance in µs for the optimized engine when built 29

B.1. Screenshot of the Unity profiler showing the time spent in methods of the statechart engine in the naive implementation when built. Minimum frame time. 42
B.2. Screenshot of the Unity profiler showing the time spent in methods of the statechart engine in the naive implementation when built. Maximum frame time. 43

List of Tables

5.1. Last 5000 easurements of the step algorithm in the naive implementation in the editor (in µs) . . . 19
5.2. Last 5000 measurements of the step algorithm in the naive implementation outside the editor (in µs) . . . 19
5.3. Last 5000 measurements of the engine in the naive implementation outside the editor when executing one instance (in µs) . . . 20
5.4. Last 5000 measurements of the step algorithm outside the editor with caching and also the difference to the naive implementation (in µs) . . . 23
5.5. Last 5000 measurements of the step algorithm outside the editor with cache optimizations and the difference to the previous implementation with caching (in µs) . . . 25
5.6. Last 5000 measurements of the step algorithm outside the editor with all optimizations and the difference to the previous implementation with cache optimizations (in µs) . . . 26

6.1. Last 5000 measurements of the step algorithm outside the editor with all optimizations and the difference to the naive implementation (in µs) . . . 28
6.2. Last 5000 samples of the engine at one instances when built compared to the sum of the algorithm measurements (in µs) . . . 28
6.3. Last 5000 samples of the engine at 1000 instances when built compared to the not optimized engine (in µs) . . . 29

A.1. Full measurement data of the naive implementation . . . 38
A.2. Full measurement data of the implementation with caching . . . 39
A.3. Full measurement data of the implementation with cache optimizations . 39
A.4. Full measurement data of the optimized implementation . . . 40

Bibliography

[1] I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 298–302, 1998.

[2] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Efficient tree layout in a multilevel memory hierarchy. In Rolf Möhring and Rajeev Raman, editors, *Algorithms — ESA 2002*, pages 165–173, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[3] Hans-J. Boehm. Mark-sweep vs. copying collection and asymptotic complexity. `https://www.hboehm.info/gc/complexity.html`, 2021. Accessed: 2021-04-09.

[4] Jenny Brusk, Torbjörn Lager, Anna Hjalmarsson, and Preben Wik. DEAL: Dialogue Management in SCXML for Believable Game Characters. In *Proceedings of the 2007 Conference on Future Play*, Future Play '07, page 137–144, New York, NY, USA, 2007. Association for Computing Machinery.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[6] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: Recording Microprocessor History. *Commun. ACM*, 55(4):55–63, April 2012.

[7] Anna Derezińska and Marian Szczykulski. Performance evaluation of impact of state machine transformation and run-time library on a c# application. In Shin-ya Kobayashi, Andrzej Piegat, Jerzy Pejaś, Imed El Fray, and Janusz Kacprzyk, editors, *Hard and Soft Computing for Artificial Intelligence, Multimedia and Security*, pages 328–340, Cham, 2017. Springer International Publishing.

[8] Digital Extremes. Warframe, March 2013.

[9] Christopher Dragert, Jörg Kienzle, and Clark Verbrugge. Toward High-Level Reuse of Statechart-Based AI in Computer Games. In *Proceedings of the 1st International Workshop on Games and Software Engineering*, GAS '11, page 25–28, New York, NY, USA, 2011. Association for Computing Machinery.

[10] D. Harel and A. Pnueli. On the Development of Reactive Systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[11] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[12] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, October 1996.

[13] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A*

Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[14] Edzard Höfig. *Interpretation of Behaviour Models at Runtime: Performance Benchmark and Case Studies*. Doctoral thesis, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Berlin, 2011.

[15] Juan Linietsky, Ariel Manzur, and the Godot community. AnimationTree. `https://docs.godotengine.org/en/3.3/tutorials/animation/animation_tree.html`, 2021. Accessed: 2021-04-11.

[16] Richard Meredith. Simple multithreading for unity. `https://richardmeredith.net/2017/07/simple-multithreading-for-unity/2/`, 2017. Accessed: 2021-04-05.

[17] Microsoft Corporation. Dictionary<tkey,tvalue> class. `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=netstandard-2.0`, 2021. Accessed: 2021-04-09.

[18] Microsoft Corporation. Hashset<t> class. `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1?view=netstandard-2.0`, 2021. Accessed: 2021-04-09.

[19] Microsoft Corporation. Hashset<t>.clear method. `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1.clear?view=netstandard-2.0`, 2021. Accessed: 2021-04-09.

[20] Microsoft Corporation. IComparable<T> Interface. `https://docs.microsoft.com/en-us/dotnet/api/system.icomparable-1?view=netstandard-2.0`, 2021. Accessed: 2021-04-11.

[21] Microsoft Corporation. List<t> class. `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=netstandard-2.0`, 2021. Accessed: 2021-04-09.

[22] Microsoft Corporation. List<t>.clear method. `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.clear?view=netstandard-2.0`, 2021. Accessed: 2021-04-09.

[23] Microsoft Corporation. Object.GetHashCode Method. `https://docs.microsoft.com/en-us/dotnet/api/system.object.gethashcode?view=netstandard-2.0#remarks`, 2021. Accessed: 2021-04-13.

[24] Microsoft Corporation. Stopwatch class. `https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=netstandard-2.0`, 2021. Accessed: 2021-03-31.

[25] Microsoft Corporation. Structure types (c# reference). `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct`, 2021. Accessed: 2021-04-09.

[26] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014. Available at https://gameprogrammingpatterns.com/. Accessed: 2021-04-09.

[27] Object Management Group, 9C Medway Road, PMB 274 Milford, MA 01757 USA. *OMG® Unified Modeling Language® (OMG UML®)*, 2.5.1 edition, 2017.

[28] OpenMP ARB. *OPENMP API Specification*, 5.1 edition, 2020. Available at https://www.openmp.org/spec-html/5.1/openmp.html.

[29] Josh Peterson. Updated scripting runtime in unity 2018.1: What does the future hold? https://blogs.unity3d.com/2018/03/28/updated-scripting-runtime-in-unity-2018-1-what-does-the-future-hold/, 2018. Accessed: 2021-04-09.

[30] Respawn Entertainment. Titanfall 2, October 2016. Publisher: Electronic Arts.

[31] Unity Technologies. Understanding automatic memory management. https://docs.unity3d.com/2020.2/Documentation/Manual/UnderstandingAutomaticMemoryManagement.html, 2018. Accessed: 2021-04-09.

[32] Unity Technologies. Unity manual: Overview of .net in unity. https://docs.unity3d.com/2020.2/Documentation/Manual/overview-of-dot-net-in-unity.html, 2020. Accessed: 2021-03-31.

[33] Unity Technologies. Unity manual: Scripting restrictions. https://docs.unity3d.com/2020.2/Documentation/Manual/ScriptingRestrictions.html, 2020. Accessed: 2021-03-31.

[34] Unity Technologies. Unity Manual: ScriptableObject. https://docs.unity3d.com/Manual/class-ScriptableObject.html, 2021. Accessed: 2021-04-12.

[35] Michael von der Beeck. A comparison of statecharts variants. In Hans Langmaack, Willem-Paul de Roever, and Jan Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[36] W3C. *State Chart XML (SCXML): State Machine Notation for Control Abstraction*, w3c recommendation 1 september 2015 edition, September 2015. Available at https://www.w3.org/TR/2015/REC-scxml-20150901/.

A. Appendix: Full Performance Measurement Data

Table A.1.: Full measurement data of the naive implementation

			Mean	**Minimum**	**Maximum**	**Median**
Algorithm	Editor	Prepare Total	5.1	4.0	3266.2	
		Prepare Last 5000	4.8	4.1	21.3	4.5
		Search Total	5.2	2.7	2106.1	
		Search Last 5000	5.3	2.8	1939.0	4.4
		Validate Total	16.5	5.2	3303.3	
		Validate Last 5000	15.9	5.3	1891.5	14.0
		Execute Total	14.1	5.5	3369.3	
		Execute Last 5000	15.1	5.6	3263.5	11.6
	Built	Prepare Total	2.3	1.3	239.7	
		Prepare Last 5000	2.3	1.4	237.9	2.0
		Search Total	2.2	0.8	264.0	
		Search Last 5000	2.1	0.9	133.6	1.7
		Validate Total	6.8	1.6	290.8	
		Validate Last 5000	6.9	1.7	232.9	4.9
		Execute Total	6.6	2.1	266.0	
		Execute Last 5000	6.6	2.2	251.7	4.3
Engine	Editor	1 instance				
		Total	59.0	19.1	3410.2	
		Last 5000	60.1	19.8	3308.4	38.2
		10 instances				
		Total 6466.4	134.5	3981.6		
		Last 5000	467.9	150.0	3921.3	383.4
		100 instances				
		Total	4228.1	2661.4	10365.2	
		Last 5000	4226.3	2697.8	7338.8	3898.1
		1000 instances				
		Total	39531.8	34407.6	90361.6	
		Last 5000	39514.6	35267.4	46222.3	39330.1
	Built	1 instance				
		Total	28.0	6.3	380.8	
		Last 5000	27.8	6.8	290.6	15.6
		10 instances				
		Total	227.2	45.8	631.0	
		Last 5000	227.0	55.0	512.5	254.5
		100 instances				
		Total	1983.9	1432.5	7235.2	
		Last 5000	1975.7	1432.5	2927.2	1964.9
		1000 instances				
		Total	16537.5	14490.8	41471.3	
		Last 5000	16536.1	14825.9	18871.7	16529.2

Table A.2.: Full measurement data of the implementation with caching

			Mean	**Minimum**	**Maximum**	**Median**
Algorithm	Editor	Prepare Total	5.0	3.8	2551.6	
		Prepare Last 5000	4.8	4.1	21.5	4.4
		Search Total	5.2	2.8	2629.0	
		Search Last 5000	5.3	2.8	2301.9	4.3
		Validate Total	15.2	5.1	3282.8	
		Validate Last 5000	15.1	5.3	2307.7	13.4
		Execute Total	10.1	3.6	3276.9	
		Execute Last 5000	10.6	3.7	2315.2	8.7
	Built	Prepare Total	2.4	1.4	242.6	
		Prepare Last 5000	2.3	1.5	152.2	2.1
		Search Total	2.1	0.8	231.7	
		Search Last 5000	2.1	0.8	160.6	1.8
		Validate Total	6.4	1.7	285.6	
		Validate Last 5000	6.4	1.8	276.9	4.7
		Execute Total	4.6	1.4	255.2	
		Execute Last 5000	4.6	1.5	150.4	3.3

Table A.3.: Full measurement data of the implementation with cache optimizations

			Mean	**Minimum**	**Maximum**	**Median**
Algorithm	Editor	Prepare Total	4.6	4.0	1540.2	
		Prepare Last 5000	4.5	4.0	23.7	4.3
		Search Total	4.7	1.6	3161.2	
		Search Last 5000	4.5	1.7	17.0	4.8
		Validate Total	11.7	4.6	3504.9	
		Validate Last 5000	12.0	4.7	1521.1	11.3
		Execute Total	4.9	3.3	3503.6	
		Execute Last 5000	5.0	3.4	1551.8	4.2
	Built	Prepare Total	2.0	1.2	256.4	
		Prepare Last 5000	1.9	1.2	126.0	1.7
		Search Total	1.8	0.4	236.2	
		Search Last 5000	1.8	0.5	146.2	1.7
		Validate Total	4.8	1.2	271.1	
		Validate Last 5000	4.9	1.2	226.6	3.9
		Execute Total	2.2	1.1	257.4	
		Execute Last 5000	2.4	1.2	222.1	1.7

Table A.4.: Full measurement data of the optimized implementation

			Mean	Minimum	Maximum	Median
Algorithm	Editor	Prepare Total	4.0	3.4	118.2	
		Prepare Last 5000	4.0	3.4	103.9	3.8
		Search Total	4.7	1.6	1707.7	
		Search Last 5000	4.6	1.6	109.5	4.7
		Validate Total	10.8	4.1	3422.0	
		Validate Last 5000	10.7	4.1	1601.6	10.7
		Execute Total	4.0	2.8	2008.0	
		Execute Last 5000	4.2	2.8	1522.3	3.7
	Built	Prepare Total	1.5	0.8	227.6	
		Prepare Last 5000	1.5	0.8	25.1	1.5
		Search Total	1.6	0.4	235.9	
		Search Last 5000	1.6	0.4	130.5	1.6
		Validate Total	4.0	1.0	293.9	
		Validate Last 5000	4.0	1.1	227.7	3.5
		Execute Total	1.8	0.9	241.2	
		Execute Last 5000	1.8	1.0	234.5	1.4
Engine	Editor	1 instance				
		Total	39.7	13.2	3285.7	
		Last 5000	38.3	13.8	1912.1	32.2
		10 instances				
		Total	291.8	80.5	3888.1	
		Last 5000	293.9	96.0	3744.2	248.2
		100 instances				
		Total	2693.1	1678.2	8613.8	
		Last 5000	2691.2	1678.2	6602.7	2316.4
		1000 instances				
		Total	26134.8	21832.8	58286.0	
		Last 5000	26616.2	22269.4	40692.7	26408.5
	Built	1 instance				
		Total	16.9	3.7	3102.8	
		Last 5000	17.0	4.0	257.3	11.5
		10 instances				
		Total	131.1	27.6	3659.3	
		Last 5000	130.5	30.8	342.7	95.1
		100 instances				
		Total	1108.9	813.2	6158.9	
		Last 5000	1105.0	813.2	1564.8	1102.5
		1000 instances				
		Total	8863.5	7667.6	24690.0	
		Last 5000	8872.0	7667.6	10220.0	8874.0

B. Appendix: Unity Profiler Screenshots

Figure B.1.: Screenshot of the Unity profiler showing the time spent in methods of the statechart engine in the naive implementation when built. Minimum frame time.

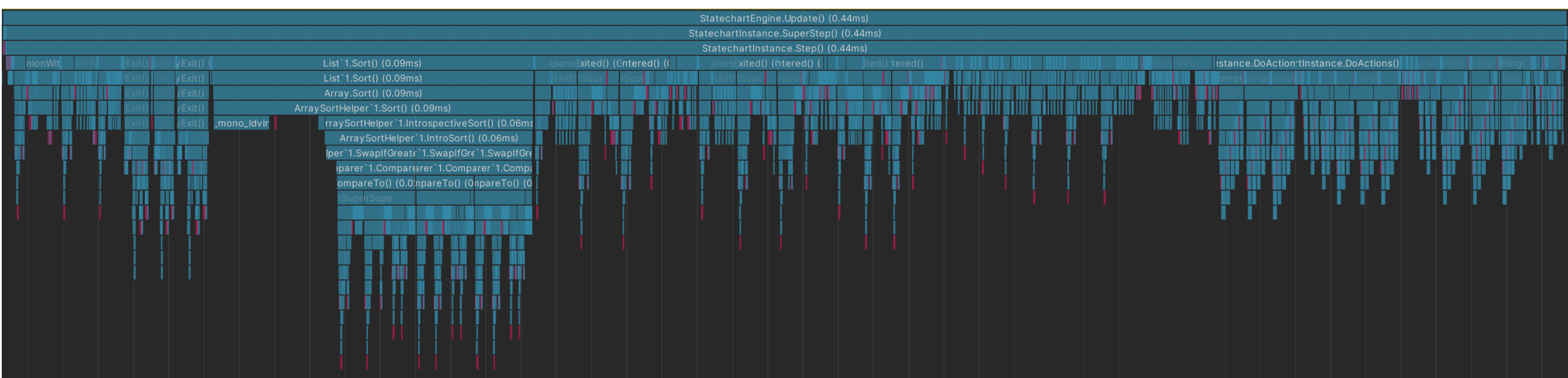

Figure B.2.: Screenshot of the Unity profiler showing the time spent in methods of the statechart engine in the naive implementation when built. Maximum frame time.