EXAMENSARBETE I DATATEKNIK 300 HP, AVANCERAD NIVÅ
*STOCKHOLM, SVERIGE 2016*

# Behavior Trees for decision-making in Autonomous Driving

MAGNUS OLSSON

**KTH KUNGLIGA TEKNISKA HÖGSKOLAN**

**SKOLAN FÖR DATAVETENSKAP OCH KOMMUNIKATION**

KTH Computer Science
and Communication

# Behavior Trees for decision-making in autonomous driving

MAGNUS OLSSON

Master's Thesis at NADA
Supervisor: John Folkesson
Examiner: Patric Jensfelt

# Abstract

This degree project investigates the suitability of using Behavior Trees (BT) as an architecture for the behavioral layer in autonomous driving. BTs originate from video game development but have received attention in robotics research the past couple of years. This project also includes implementation of a simulated traffic environment using the Unity3D engine, where the use of BTs is evaluated and compared to an implementation using finite-state machines (FSM). After the initial implementation, the simulation along with the control architectures were extended with additional behaviors in four steps. The different versions were evaluated using software maintainability metrics (Cyclomatic complexity and Maintainability index) in order to extrapolate and reason about more complex implementations as would be required in a real autonomous vehicle. It is concluded that as the AI requirements scale and grow more complex, the BTs likely become substantially more maintainable than FSMs and hence may prove a viable alternative for autonomous driving.

# Referat

## Behavior Trees för beslutsfattande i självkörande fordon

Detta examensarbete undersöker lämpligheten i att använda Behavior Trees (BT) som underliggande mjukvaruarkitektur för beteendekomponenten inom självkörande fordon. Behavior Trees härstammar från datorspelsindustrin men har de senaste åren fått uppmärksamhet inom robotikforskning. Detta arbete inkluderar även implementationen av en simulerad trafikmiljö med hjälp av Unity3D-spelmotorn, där användandet av BTs utvärderas och jämförs med en implementation som använder finite-state machines (finita automater eller FSM). Efter att den initiala simulationen var färdig utökades den och styr-arkitekturerna med utökad funktionalitet i fyra ytterligare steg. De olika versionerna utvärderades med hjälp av två olika mjukvarumått på underhållbarhet (Cyclomatic complexity och Maintainability index). Detta för att kunna extrapolera och resonera kring de mer komplexa system som skulle krävas för ett faktiskt självkörande fordon. Slutsatsen dras att efterhand som storleken och komplexiteten på AI:n ökar, blir sannolikt BTs betydligt lättare att underhålla jämfört med FSMs och kan därför vara ett lämpligt alternativ för självkörande fordon.

# Contents

# Chapter 1

# Introduction

For the past couple of decades, the field of autonomous vehicles has been an active area of research and development. Prize competitions such as the DARPA Grand Challenge has also helped propel the state of the art forward to the point where a commercial company, Google, now has logged over 1,000,000 miles driven on public roads with their driverless vehicles [1]. Several other car companies are developing their own autonomous solutions as well. The car manufacturer Tesla has already enabled a limited form of autonomous driving in their consumer car [2].

Every year, approximately 1.2 million people are killed in automobile accidents. A significant number of these accidents are caused by driver error and other preventable causes [3]. The increased proliferation of autonomous vehicles on public roads has the potential to provide large benefits in terms of fewer accidents, more cost effective transports and reduced traffic congestion in urban areas.

While the many benefits are promising, there are also numerous challenges which need to be overcome, chief among them may be the reliability and safety of the autonomous driving system itself [4]. It is extremely difficult to fully understand the inner workings of a system with the complexity required to successfully control a self-driving car. To safely be able to modify, extend or repair the behavioral logic of such a system, a sufficient overview of the system is required. Successfully building such complex systems is greatly aided by choosing a suitable architecture for the task.

**Behavior Trees**

Behavior Trees (BTs) offer one such candidate architecture which could prove a promising foundation for reliable behavioral systems. While BTs emerged from the video games industry in the mid-00s, it has prompted increasing research within the academic robotics community in recent years. With autonomous vehicles and robots having much in common, BTs warrant further investigation regarding their suitability as a platform for self-driving cars.

The BT concept emerged from the computer gaming industry, where they were first described as powerful tools to provide intelligent behaviors for non-player characters in high profile computer games, such as the Halo series [5–7]. Since then, they have become ubiquitous within the gaming industry [8–10].

A BT is a control structure formulated as a tree, with a single root node where the execution always starts. The root node has children, which can have different properties which control the flow of the execution. At the end of each branch of the tree are leaf-nodes, where the actual behavior and execution of tasks resides. The tree is principally traversed in a depth-first fashion and always starts from the beginning in each execution cycle.

The analogy has been made between finite-state machines (FSMs) being to BTs as what the GOTO-statement is to function calls. I.e. FSMs offer a one-way control transfer of control much like GOTO enables you to jump anywhere in a program, making it dangerously easy to make the program into an unmaintainable mess. Function calls on the other hand offer a two-way transfer of control, much like in BTs where return values are being passed up and down the tree structure [11].

## 1.1 Thesis motivation

BTs appear to offer significant benefits in terms of robustness, readability, maintainability and modularity, to name a few. Due to the enormous complexity involved in creating an AI capable of safely driving a vehicle in traffic, BTs may prove to be a significant improvement over current control system technologies for the particular application of autonomous driving.

## 1.2 Thesis objective

The main purpose of this thesis is to serve as a resource on whether BTs are suitable architecture choices for self-driving cars. The second contribution of this work is to implement a simulated driving environment in the Unity3D engine. The simulation will serve as a proof of concept and testing ground.

The simulation will consist of a road with two lanes, with cars heading in both direction. The AI controlled car will need to overtake cars while safely avoiding oncoming traffic. Later, the simulation is extended with two intersections and additional traffic which the car must navigate. The AI of the car will be implemented in two different architectures. Once with a Behavior Tree and once with a finite-state machine (FSM). This behavioral component will be deciding which action to take, when to take them and how long to take them.

Software metrics are employed at different stages in development as to extrapolate the projected maintainability and reusability in systems more extensive and complex than in this work. In chapter 6 the suitability, advantages and disadvantages of BTs are discussed in detail.

This thesis deals only with the architecture of the behavioral layer of the AI. Other subsystems involved in autonomous vehicles are outside the scope of this thesis.

# Chapter 2

# Background

## 2.1 History of autonomous cars

The first car one could consider robot controlled was developed in the early 1980s by a team lead by Ernst Dickmanns. It was a 5-ton van modified to give an onboard computer control over steering, throttle and brakes. Control commands were issued by the computer based on real-time computer vision and probabilistic techniques such as Kalman filters [12]. The vehicle was tested on empty roads as a safety precaution and was eventually able to drive all by itself at speeds up to 96km/h.

In 1987 the European organization EUREKA funded a large scale research project in autonomous driving dubbed Prometheus (PROgraMme for a European Traffic of Highest Efficiency and Unprecedented Safety). The member states contributed €749 million and the project ran from 1987 to 1995. The cars were initially slated to utilize autonomous guidance by buried cables but focus shifted to the machine vision approach proposed by Dickmanns, following his encouraging results. By the mid-90s, the project had developed vehicles able to drive on highways at 80km/h in busy traffic [13]. The Prometheus-project laid much of the foundation for coming research, having developed techniques like convoy driving, automatic tracking of other vehicles, and lane changes left and right with autonomous passing of other cars [13].

The next spurt of innovation would come in 2004 from DARPA (Defense Advanced Research Projects Agency) holding it's first of Grand Challenges and offering a $1 million prize to any team able to create an autonomous car capable of navigating a 150-mile course through the Mojave Dessert. Several teams now utilized GPS systems and LIDAR sensors. Disappointingly, no team successfully completed the entire course. The $1 million prize was awarded to Carnegie Mellon University's car *Sandstorm* for reaching the farthest (7.4 miles). The next year, in the second Grand Challenge, five teams completed the 132 mile course, with all but one of the 23 starting teams reaching farther than previous year's winner. The competition was won by *Stanley*, the entry from Stanford University headed by Sebastian Thrun [14].

DARPA's 2007 challenge took to a closed urban environments with the contestant cars driving on the roads at the same time. The $2 million prize was won by Carnegie Mellon University's car *Boss*. The car was equipped with global positioning system, LIDAR, radars, and cameras. A three-layer planning system combined mission, behavioral, and motion planning to drive in urban environments [15].

### 2.1.1 Current solutions

While there is no standardized architecture design for autonomous vehicles, one can look at the documented instances in the literature to see some examples. The several DARPA Grand Challenges has led to most teams publishing reports outlining the inner working of their cars. Of the cars researched, all employ some form of finite-state machine.

Team Tartan Racing, the winning team of the DARPA Urban Challenge in 2007, implemented their behavior generation component as a hierarchical finite-state machine, decomposing the mission task into a set of top-level behaviors and their simpler, sub-behaviors in order to complete a mission [16]. At the highest level, the three contexts are road, intersection and zone, with their corresponding behaviors being, respectively, lane driving, intersection handling and achieving a zone pose [15].

The Stanford University team, finishing in second place after Team Tartan Racing with their entry "Junior", designed their car's behavioral subsystem as a hierarchical FSM [17]. The top-level hierarchy contains 13 states categorized as normal behavior with the other category of lower level states being referred to as exception states. Another team in the DARPA Urban Challenge 2007, from the Ohio State University, also utilized the hierarchical FSM architecture on their car OSU-ACT [18].

California Institute of Technology also chose a hierarchical state structure for their car. Higher-level states (road region, zone region, off-road, intersection, U-turn, failed and paused) which could then be decomposed into more intricate discrete steps [19].

Unfortunately, there are currently no public details as to what architecture Google's driverless-car software utilizes. However, the founder of their driverless car project, Sebastian Thrun, hails from Stanford University where he led their winning team in the DARPA Grand Challenge 2005. In that winning car, "Stanley", the global driving mode was maintained in a finite-state machine, with a state estimation module identifying the state [14].

## 2.2 Behavior Trees

Behavior Trees did not originate from academia and thus there is no single person credited with their inception nor much early literature about them. Instead, BTs originated around 2004-2005 from the computer gaming industry, most notably for Halo 2 by Damian Isla [5] and Façade by Michael Mateas and Andrew Stern [6].

Another early adopter who has helped popularize BTs is Alex J. Champandard, whom up until recently was Lead AI Programmer at Rockstar Games. He is the founder of AiGameDev.com where early tutorials and discussions about Behavior Trees appeared [20].

Since then, BTs have become ubiquitous and are now called the de facto standard in computer game AI [9]. For example, the popular Unreal Engine 4 uses BTs as its built-in AI design system [10].

In academia, BTs have mainly received attention in the field of robots in recent years [11, 21–26], but also for its possible applications in evolutionary learning [27–30].

# Chapter 3

# Theory

## 3.1 Finite-state machines

Finite-state machines (FSM) is a mathematical model of computation commonly used in computer programs. It is conceptualized as an abstract machine which can only reside in one out of a number of pre-defined states, between which transitions are performed to enter a different state. Once in that state, some computation or action is performed before proceeding again to the next state.

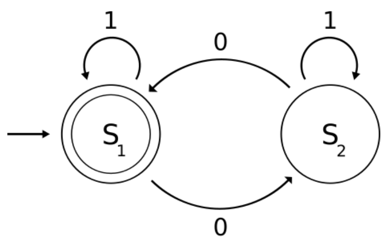

**Figure 3.1.** Example finite-state machine

FSMs are a common way to solve the high-level control problem in robotics and AI. However, there are a number of challenges and disadvantages which may arise when attempting to implement a control system using FSMs.

FSMs are known to become unmanageable for large complex systems [23], sometimes called the state and transition explosion. To have a fully reactive system, every state needs to have a transition to every other, making it a fully connected graph ($\mathcal{O}(n^2)$). This makes maintenance and modification quite labor intensive and prone to bugs. I.e., removing a particular state requires correctly modifying those other states able to transition to the old one.

This lack of modularity also complicates delegation of work for independent components of a behavioral control system, due to state transitions being interconnected to such a high degree. FSMs simply become very hard to collaborate on as they grow.

## 3.2 Hierarchical FSMs

Hierarchical state machines (HFSM), also known as statecharts, were developed by D. Harel in 1987 [31] in an attempt to alleviate the cumbersome transition duplication required in large FSMs as well as add structure in order to aid comprehension of complex systems. It clusters together states into a group (named superstate) where all the underlying internal states (substates) implicitly share the same superstate.

While it is more modular than traditional FSMs, it still inherits most of the disadvantages, such as limited reusability.

Rather than having to duplicate transitions to a particular state for every other state, an HFSM allows for some or all of the transitions to be inherited from a superstate through polymorphism.

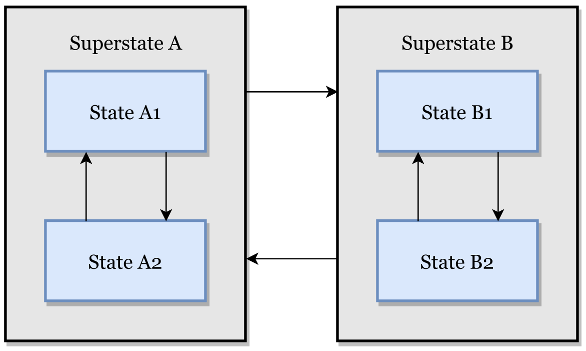

**Figure 3.2.** Hierarchical FSM.

## 3.3 Behavior Trees

### 3.3.1 Definition

Behavior Trees are formulated as directed graphs with a tree structure. The topmost node, denoted as the root node, has one or more child nodes, whom in turn can also have children. Nodes which have children are sometimes referred to as composite nodes. A child node with no children of its own is called a leaf. The root node is the only one without any parent.

Each node in the tree belongs to one of the six types of nodes shown in 3.3.1. For non-leaf nodes, they can be any of the following four types, Selector, Sequence, Decorator or Parallel. Leaf nodes come in the form of either Action or Condition. The execution of a BT starts at the root node every time and then progressively traverses the structure in a depth-first fashion, polling (usually referred to as ticking) every node as the execution proceeds through the tree. Depending on the type and status of the nodes, the execution proceeds down and up the tree. When a tick reaches a leaf node, some computation or action is performed, and then returns either *Success*, *Failure*, or *Running*. The return status propagates up the tree, eventually back to the root node. Executing the BT immediately again will poll the same nodes, even though they returned success last time. I.e. there is no memory from the last execution or skipping of previously successful or failed nodes in a rudimentary BT.

| Node type | Succeeds | Fails | Running |
|---|---|---|---|
| Selector | If one child succeeds | If all children fail | If one child is running |
| Sequence | If all children succeed | If one child fails | If one child is running |
| Decorator | Varies | Varies | Varies |
| Parallel | If N children succeed | If M-N children succeed | If all children are running |
| Action | Upon completion | When impossible to complete | During completion |
| Condition | If true | If false | Never |

**Table 3.1.** Node types [11]

### 3.3.2 Selector node

A selector node will begin to tick its children in order. If the first child fails, the execution continues to the following child and it is ticked. I.e. if a node fails, go to the next one as a fallback, hence the Selector node is sometimes called Fallback-node. If a child succeeds, the selector also returns success and does not move on to the following children. It is depicted as an oval with a question mark.

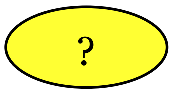

**Figure 3.3.** Selector node

**Algorithm 1** Selector node
```
1: for each node $n \in children$ do
2:     $childstatus \leftarrow$ tick($n$)
3:     if $childstatus$ = running then
4:         return running
5:     else if $childstatus$ = success then
6:         return success
7:     end if
8: end for
9: return failure
```

### 3.3.3 Sequence node

A sequence node ticks its children in sequence, trying to ensure that a number of sequential tasks are all performed. If any child return failure, the sequence has failed and it will propagate up. The sequence node only returns success if all children succeed. It is drawn as a oval with an arrow pointing right.

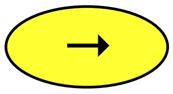

**Figure 3.4.** Sequence node

**Algorithm 2** Sequence node

```
1: for each node $n \in children$ do
2:     $childstatus \leftarrow$ tick($n$)
3:     if $childstatus$ = running then
4:         return running
5:     else if $childstatus$ = failure then
6:         return failure
7:     end if
8: end for
9: return success
```

### 3.3.4 Parallel node

A parallel node ticks all its children at the same time, allowing several Action nodes to enter a running state at the same time. The requirement for how many children need to succeed before the Parallel node itself reports success/failure can be customized on a per-instance basis. It is represented as an oval with two arrows pointing right.

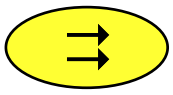

**Figure 3.5.** Parallel node

**Algorithm 3** Parallel node

```
1: for each node $n \in children$ do
2:     $childstatus[i] \leftarrow$ tick($n$)
3: end for
4: if all_running($childstatus$) then
5:     return running
6: else if success_critera($childstatus$) then
7:     return success
8: else
9:     return failure
10: end if
```

### 3.3.5 Decorator node

The decorator node wraps the functionality of the underlying child or subtree. It can for example influence the behavior of the underlying node(s) or modify the return state. A typical kind of Decorator node is an inverter, which flips the return status of its child from Success to Failure and vice versa. Another example of a Decorator can set a time limit on the execution of its child, breaking and returning failure if the child did not complete fast enough.

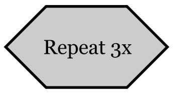

**Figure 3.6.** Example Decorator node

**Algorithm 4** Decorator node

```
1: $childstatus \leftarrow$ tick($n$)
2: return func($childstatus$)
```

### 3.3.6 Action node

An Action node performs a task and returns Success if the action is completed, Failure if the task could not be completed, and Running while the task is being performed. It is drawn as a simple rectangle.

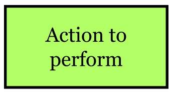

**Figure 3.7.** Action node

### 3.3.7 Conditional node

The Condition node is analogous to a simple if-statement. If the conditional check is true, the node returns Success and Failure if false. A Condition node will never return a Running status. It can be denoted visually as a rhombus.

**Figure 3.8.** Condition node

### 3.3.8 Properties

A naive implementation of BTs exhibits blocking behavior, meaning that once a task is running it will necessarily continue until it is finished. Consequently, the BT is not able to react during this time. Should a potentially hazardous situation arise, the self-preservation behavior may not be invoked in time. To make BTs suitable for autonomous driving, this limitation needs to be addressed (See 3.3.9).

Unlike in FSMs, transitions in the BT are implicit encoded in the tree structure, thereby absolving the developer from explicitly having to implement all transitions. Adding or removing a node or subtree therefore has no impact on the other nodes, which makes development easier and less prone to bugs.

The tree is traversed depth first from the root node, which will essentially mean that left-most nodes are visited first and therefore the highest priority behavior should be placed left of less important behaviors (The parallel node is an exception to this). To exploit this, the designer of the behavior tree must place the important behaviors such as self-preservation and collision avoidance at the left branches of the tree. The default behavior will thus end up at the far right of the tree. If the desired default behavior is to do nothing, then having all branches be conditional will result in an idling behavior, should no condition be met.

Many graphical editors exist for making Behavior Trees in the context of computer game AI. There are multiple community made plug-ins for the Unity engine. The Unreal Engine 4 has an advanced built-in BT editor and utilizes behavior trees as the foundation of its NPCs [10]. Thanks to the ease of flow in a BT and with a visual editor even non-coders can design complex behaviors.

### 3.3.9 Concurrency

In order to overcome the blocking nature of naive BTs, tasks need to be run independently from the system ticking the nodes in the tree. This can be achieved either by executing the behavior in a thread concurrently as the main thread keeps ticking the tree, or, by means of asynchronous tasks running on the same thread.

Depending on the application and particular task at hand, the BT may need to resume a series of tasks where it left off, rather than starting from the first step again. There is no established or standardized means of implementing concurrency in a behavior tree and is left up to the implementer as an engineering decision. Three of the approaches are via event-driven design, an open/closed node pattern or via recursive abort-calls.

In the event-driven approach, the BT itself is responsible for notifying nodes about task executions throughout the tree, thereby allowing them to reset. This design was implemented in the simulation and is discussed further in 5.2.

The open/closed pattern keeps track of which nodes have not yet begun executing (open) and which have completed their execution (closed). Given this information, the nodes can be re-initiated appropriately on the following tick.

Recursive abort-calls have the benefit of not requiring blackboard knowledge shared between nodes or the BT itself. Once a task wishes to execute and returns *Running* or *Success*, a recursive abort-call is issued to all subsequent nodes (i.e. all nodes to the right of the executing node in the tree). The call reaches the root-node which proceeds to send it down again to the remaining branches.

## 3.4 Related work

In the scientific literature, researchers have recently considered BTs as a viable control structure to increase modularity, robustness and safety of controlled systems (e.g. UAVs and robots) [11, 23].

The main advantage to using BTs over FSMs is that the transitions between states are implicit in the tree structure of the BT [11], eliminating the labor-intensive (and error-prone) need to explicitly implement the potentially $\mathcal{O}(n^2)$ number of transitions. Additionally, actions, conditions and even subtrees can be added and removed anywhere in the BT without requiring modification of the other components. This is in stark contrast to FSMs where e.g. removing a state means revising all transitions leading to or from that particular state. For potentially dangerous applications, being able to modify and add behavior without risk of compromising basic reliability functions is a significant advantage.

Due to there being no limitation on the computation performed by leaf action nodes in a BT, any Hybrid Dynamical System (HDS, a type of state machine) can be converted into a BT. [11].

Traditional BTs have certain intrinsic limitations. Two such limitations, nodes being memory-less (having no memory of previous running node), and several BTs executing independently (making coordinating collaborative tasks between agents difficult) have been addressed and solutions proposed [21]. However, the proposed framework does not deal with interrupting behaviors, as is required for autonomous driving.

For communication between nodes, behavior trees typically rely on a blackboard, which is a centralized repository of data to which all interested parties have access [32]. This approach is not suitable for encapsulation and, as a result, complicates subtree reuse. The behavior tree structure itself can make it difficult to track what and where data is stored where in the blackboard. For example, several subtrees may use the same field of the blackboard several layers down in their hierarchy, and could overwrite each other in a manner difficult to trace. To alleviate this problem and reduce BT reliance on blackboards, parameterization of subtrees has been proposed [33]. This allows later reuse of the subtree where the argument can be supplied as needed.

According to [24], there are no claims that BTs are superior to FSMs from a purely theoretical standpoint. Rather, all BTs can most likely be formulated in terms of an FSM, just as most general purpose programming languages are equivalent in the sense of Turing completeness. There is however a significant difference in terms of modularity, readability and reusability.

BTs have also been shown to be well suited for evolutionary learning, automatically generating BTs for game AI and automatic planners [27–29]. Automatically generated BTs have also been explored as the as the control system for a simulated semi-autonomous surgical robot for brain tumor ablation with successful results [25]. Thanks to the human readability of BTs, the tree can easily be pruned of unnecessary branches after generation.

# Chapter 4

# Method

In order to evaluate the suitability of Behavior Trees, and to contrast them against finite-state machines, a proof of concept simulation of an autonomous car was implemented using the Unity3D engine. The simulation was intended to be realistic in terms of vehicle dynamics and physics. Additionally, the vehicle should only make use of information which could possibly be obtained by a physical car with sensors.

The behavioral layer component was developed once using a BT and once using an FSM driven car, with the goal that both approaches exhibit identical driving behavior. To investigate the supposed strong suit of BT and weakness of FSM, namely maintainability, once the simulation was completed both the BT and FSM were extended in four steps to include additional behaviors. At each point, the code associated with the different architectures was evaluated using two software metrics, *Cyclomatic complexity* (CC) and *Maintainability index* (MI).

## 4.1 Unity3D

Unity3D is an integrated development environment geared towards video game development. It combines a visual editor for placing and visualizing 3D-models in virtual space, with the ability to attach behaviors or scripts to the objects. Unity3D supports development in C#, their own adapted version of JavaScript, and a python-derived scripting language of their own named Boo.

Unity3D has a large selection of available plugins. There exists several visual drag-and-drop editors for creating and generating behavior trees. However, in order to better understand, evaluate and reason about the BT it was implemented from scratch.

The Unity engine has a few limitations in terms of concurrency. All scripted logic is executed from the main thread only, meaning that the developer is not allowed to interact with the game physics from a separate thread. Instead, Unity requires the use of asynchronous tasks, called coroutines. The coroutines only get executed on the main thread and are therefore allowed to interact with the world. Car behaviors with a non-zero running duration need to be executed in such a coroutine.

## 4.2 Implementation

### 4.2.1 Simulation environment

The initial implementation constructs a scenario where a virtual car is driving along a closed circuit track. The track has two lanes with traffic going in opposite direction and with other car actors also driving. The car equipped with the AI is driving at a faster speed than the other cars and thus will need to overtake cars in its own lane while avoiding oncoming traffic safely. While the car is performing an overtaking maneuver, it will must be able to respond to oncoming traffic and potentially abort the procedure. This is the main behavior this implementation is meant to model. The other cars in the simulation only follow their lane at a set speed and do not react.

The car navigates by following one of two waypoint circuits matching the two lanes of the track. In order to change driving lane the car switches between the different waypoint circuits.

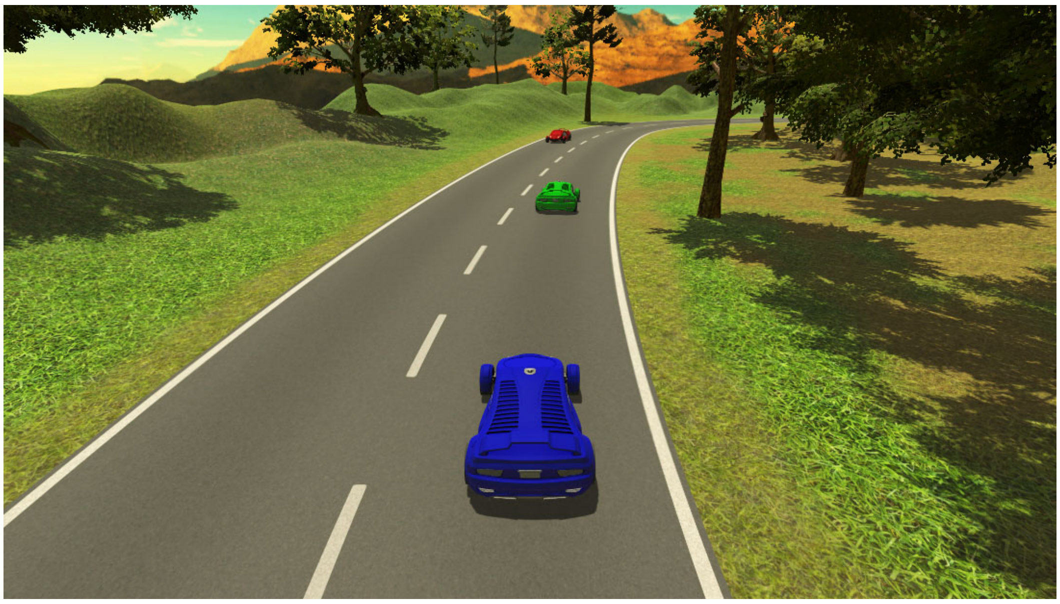

**Figure 4.1.** Image of the simulated environment. Red cars are oncoming traffic while green drive in the AIs lane.

The AI car has three simulated sensors, taking the form of collider objects in Unity. These can detect (trigger) when other models intersect them and are used as analogues to sensors on a self-driving car. This particular implementation used one large circular collider as a long range sensor and a smaller circular collider as a flag for nearby objects. The third collider is rectangular and senses along the side of the car, enabling the AI to know if we are side-by-side with another car, for example during an overtaking maneuver.

The model of the car, along with some scripts to make it drive and turn, were included in the Standard Assets Example Project which comes bundled with the Unity engine. The logic controlling the car was adapted to suit the purpose of this implementation. The road was constructed with a tool called EasyRoads3D from the Unity asset store.

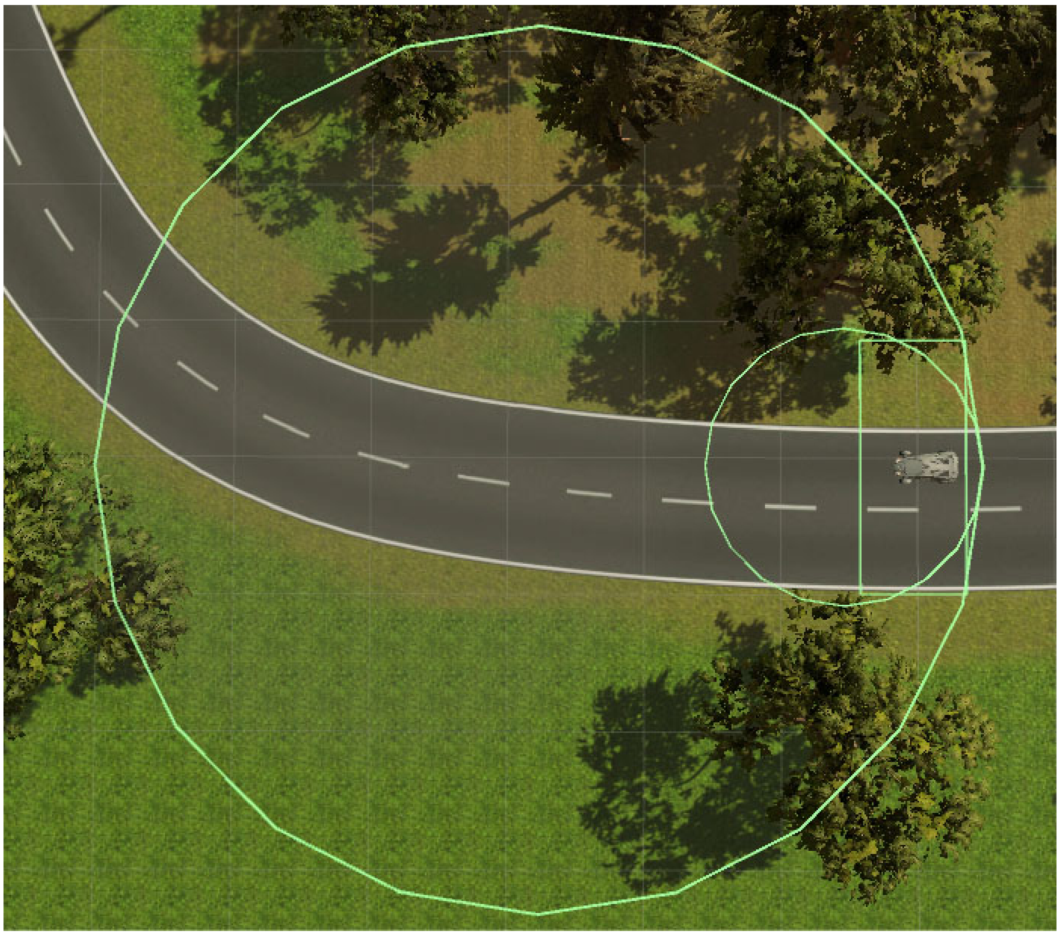

**Figure 4.2.** The three colliders acting as sensors for the car.

### 4.2.2 Car behavior

In order to act as an authentic vehicle in traffic, the car needs to be able to perform in a number of behaviors or actions.

The car needs to be able to:

- Detect other cars on the road and determine whether the detected car(s) are driving in our lane or are oncoming traffic.
- Change lanes when necessary for evasive maneuver or overtaking.
- Change speed and brake.
- Cruise along normally.

These behaviors are codified in the nodes making up the Behavior Tree and the states in the finite-state machine discussed below.

The detection of other cars is facilitated by their collider intersecting with any of the three colliders attached to the AI car. When an intersect occurs, a trigger event is generated in the Unity engine which can be processed. The type of car (oncoming or not) is determined in the script by checking the type of car. In an actual sensor-equipped autonomous car this would be achieved e.g. by comparing the positions after two LIDAR scans and determining which direction the car is moving.

Changing lanes is performed by modifying which set of waypoints is active in the car's waypoint tracking system. The collections of waypoints in the outer and inner lanes have the same number of points, meaning that the car continues from the current waypoint index when switching lanes.

To change speed the car's maximum velocity is either lowered or raised by a simple property setting.

Cruising consists of simply following the waypoints and maintaining cruising speed. The waypoint tracking script iterates through the waypoints and updates the next coordinate to head towards.

### 4.2.3 Behavior Tree

Because the overtaking maneuver is a long running action, one cannot use a simple blocking Behavior Tree. Simply running the behavior and wait until it completes would remove the ability to react to new hazards when they occur. Hence, a blocking BT would not be able to perform this task while maintaining full reactivity. Instead, the BT was implemented with an event-driven architecture.

Two different types of action leaf nodes were implemented; *Action* and *AsyncAction*. The Action node in this context only serves to modify variables and settings which can be completed instantly, e.g. setting a different active lane or modifying the maximum driving speed. AsyncActions are for behaviors that do not complete instantly and therefore require a Unity asynchronous coroutine to be spawned. Once the coroutine is started, the node returns the status code *running* up the BT. This causes the parent composite node (a *Sequence* or *Selector*) to save the index of the previous running child. The next tick it will skip the preceding steps and directly tick the running node again. Once the coroutine has finished running, the AsyncAction node will be able to signal success or failure on the next tick.

Only *Pass car* is a long-running task in this implementation. Due to the fact that *Avoid obstacle* is the very highest priority behavior, it does not need to be long running since it doesn't get aborted by anything else. By introducing asynchronous tasks, and thanks to the eccentricities of Unity3D, the BT was required to have knowledge about which task was currently running to be able to abort it. That knowledge was not possible for a lone node to have and so this forced the BT to have state and mutable data. Thereby defeating one of the major benefits to the idealized classical BT (flow controlled by nodes, no state or mutability). It should be stressed that this is due to Unity3D's coroutine design, where the execution of the coroutine resides in the parent *GameObject*, and not due to BTs themselves. With access to real multi-threading, this would not be a problem.

In this implementation, only a single task can be running at the same time. Should one task take priority to run over the current one, the older task gets terminated. This is not compatible with the standard Parallel node so this work does not support that particular concept. Should this BT implementation be reused, it may need to be adapted for running multiple tasks in parallel.

For this particular simulation, *Parallel* nodes and *Decorator*s were not used and hence were only implemented as stubs.

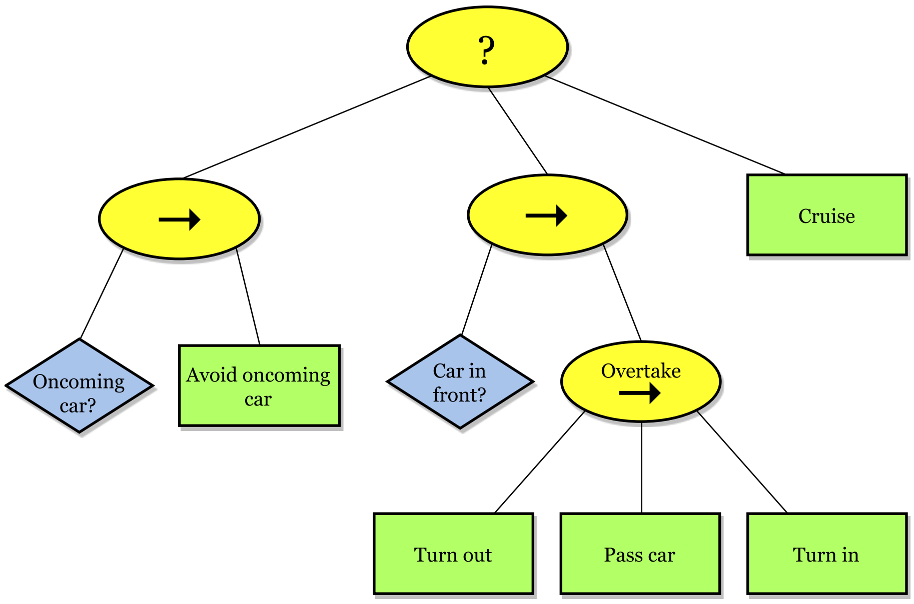

**Figure 4.3.** Behavior Tree used in simulation

### 4.2.4 Finite-state machine

The FSM implementation was straight-forward. The internal states correspond to the nodes comprising the BT but with the conditional node logic instead encoded in the state transition checks. Every "tick" of the game engine entails execution of one state's body followed by logic to determine which state will be executed in the next tick.

Unlike the aforementioned BT implementation, the FSM does not need any global state or knowledge shared between them. The BT required global tree-wide knowledge of which Action was executing in order to be able to interrupt it from any location in the tree. In the FSM, the task running is always at the current execution point in the code, allowing for local task termination before moving on to the next state. Technically, in this particular implementation, the coroutine is started and controlled in another object, but conceptually it could all be handled from scope of the current state.

High-priority tasks, such as avoiding obstacles in this particular case, need to be accessible from all the other states. Hence, all transition functions were required checking whether to transition to the avoid obstacles state.

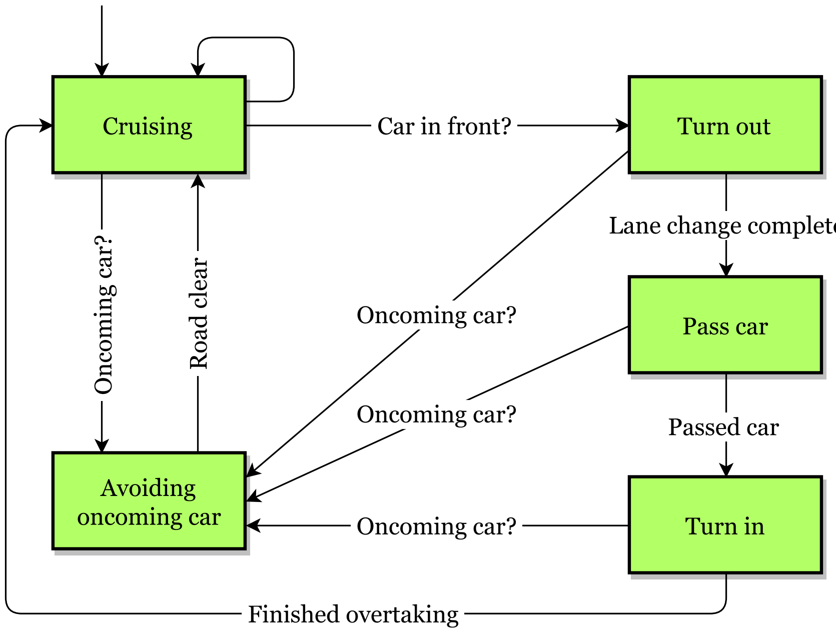

**Figure 4.4.** State machine used in simulation. Note the many transitions to the *Avoid obstacle* state

## 4.3 Extending the simulation

Since BTs and FSMs in principle are able to produce the same behavior if implemented correctly (i.e. there are no inherent differences in terms of the results one can achieve) the quality of the car's behavior is not a useful measure in trying to distinguish BTs from FSMs. Any discrepancy would simply be the failure of the developer to properly implement the designed behavior.

One of the major benefits attributed to BTs over FSMs is superior maintainability and ease of extensibility [11, 23, 24]. To highlight this difference, the two contending architectures can be compared in terms of how well they scale when new functionality is added.

To evaluate this, after the simulation was completed, four additional steps of extending functionality were performed. Both the BT and FSM needed to incorporate new behaviors to accommodate the new features required of the car AI.

There are five distinct versions of the simulation, each building on top of the next. Each of these points in development serve as a point of measurement for the code metrics.

Measurements were taken at the following five points:

- Initial working simulation with a single closed track. Cars going in both directions at constant speeds, with the AI car overtaking and avoiding oncoming traffic as necessary.

- In the first extension of the simulation, where cars occasionally came to a full stop, simulating emergency braking for sudden wildlife for example. This required the AI car to also be able to come to a full stop, whereas previously it merely slowed down and kept its distance. The ability to completely stop would also be required for the second extension of the simulation.

- In extending a second time, an intersecting road was added (see Fig.4.5), creating two different intersections which needed to be navigated safely. Additional cars travelling from both sides were added, spawning with randomized frequency. The older cars circling the track are required to yield to them, only able to drive across the intersection when an opening presents itself.

- The ability to detect when approaching an intersection was added, as to mimic more responsible driving behavior. When approaching an intersection, the car merges back to its original lane so as to not perform an overtake maneuver while crossing the perpendicular road. This was feature was mostly added to gain additional data points for the software metrics measurements.

- After adding the ability of the AI to metaphorically honk its horn at a stationary car, given no other car can be detected at the time. Again, this was added to gain further data for the complexity measurements.

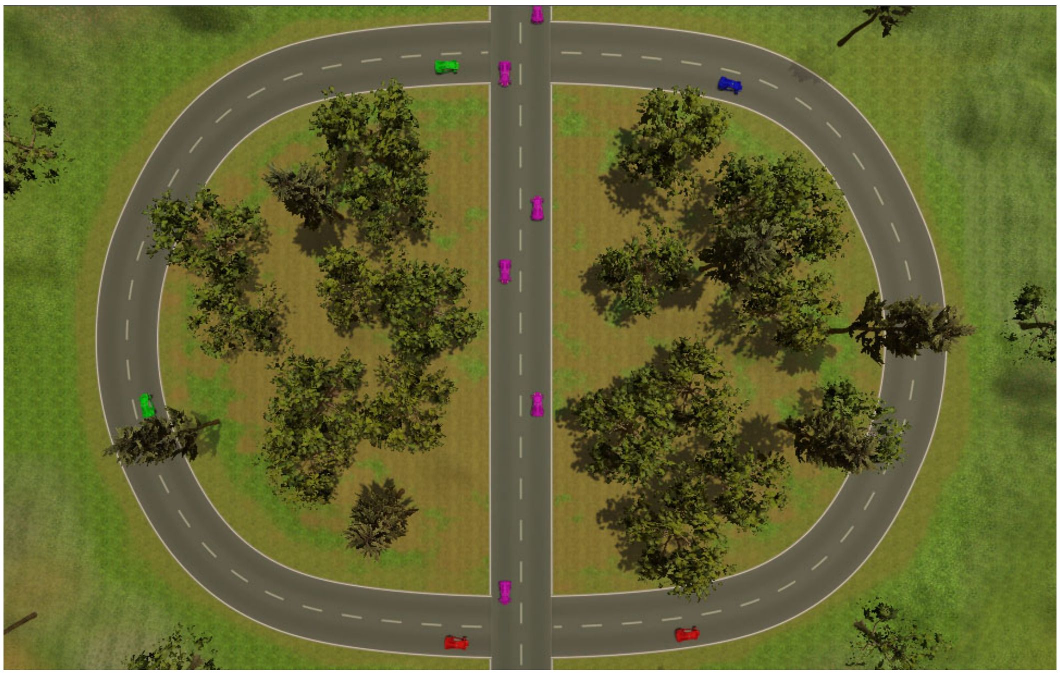

**Figure 4.5.** Overview of the track after adding an intersecting road. Magenta cars have right of way whereas blue, green, red cars yield.

The listed additions to the AI took the form of new states in the FSM (along with transitions between them) and as new nodes in the BT.

## 4.4 Software metrics

For the evaluation of the code associated with the BT implementation and the FSM implementation, two software complexity metrics are employed.

### Cyclomatic complexity

Cyclomatic complexity is a complexity measure based on graph-theoretic concepts proposed in 1976 by T.J. McCabe [34]. It gives a quantitative measurement of the number of linearly independent paths through the execution of a program. For example, a program without any flow control statements (such as if-statements) would have the cyclomatic complexity of 1, since there is but a single execution path. Adding an if-statement would increase the number of paths to 2. Lower value is associated with better maintainability.

There are several formulas for calculating CC, one of which is:

$$CC = E - N + 2P$$

$$\begin{aligned} \text{where } &E \text{ is the number of edges of the graph} \\ &N \text{ is the number of nodes of the graph} \\ &P \text{ is the number of connected components} \end{aligned}$$

While this number naturally depends to a great deal on the size of the program (one could divide by the lines of code to get a kind of complexity density), comparing two programs which perform the same task, such as this case, gives a meaningful metric. pi Another way too look at CC is that it corresponds to how many tests would be needed to have 100% test coverage. This type of structured testing methodology based on the cyclomatic complexity is called basis path testing [35].

### Maintainability index

Maintainability Index is a software metric introduced by P. Oman and J. Hagemeister, originally presented at the International Conference on Software Maintenance ICSM 1992 [36] and later distilled into its more common form in a paper that appeared in IEEE Computer [37]. It is an amalgamation of several other metrics, weighted and combined to produce the MI figure. The original formula is:

$$MI = 171 - 5.2\ln(HV) - 0.32CC - 16.2\ln(LOC) + 50\sin\sqrt{2.46 * COM}$$

where $HV$ is the Halstead volume [38]
$CC$ is the cyclomatic complexity
$LOC$ is the number of lines of code
$COM$ is the percentage of comments in the code

There are however adaptions to the MI formula. In the evaluation used for this thesis, the adaption implemented in Microsoft Visual Studio is used, which does not take amount of comments into account. It also normalizes the produced value to between 0 and 100. A higher value represents better maintainability. The formula is:

$$MI = MAX(0, (171 - 5.2\ln(HV) - 0.23CC - 16.2\ln(LOC)) * 100/171)$$

where $HV$ is the Halstead volume [38]
$CC$ is the cyclomatic complexity
$LOC$ is the number of lines of code

The range between 100 and 20 has been classified as good maintainability, 19 and 10 as moderately maintainable, and 9 to 0 as low maintainability.

The MI values for the two architectures were obtained by averaging the MI over the different constituent components (typically Classes).

# Chapter 5

# Results

## 5.1 Simulation

The simulation was implemented successfully with both a Behavior Tree and a finite-state machine. The car's AI can be toggled between using either architecture. The two implementations exhibit exactly the same driving behavior, with the only difference being the underlying structure.

The car AI is able to cruise in its lane (following waypoints) until it catches up to a slower car. The AI slows down to match the speed of the slower car, while checking it's sensors for oncoming traffic. Should no oncoming traffic be present, a lane change maneuver takes place and the speed is increased. During the passing of the slower car, the longer range sensor keeps monitoring for approaching cars. If an oncoming vehicle is detected, the AI decelerates and aborts the passing maneuver by changing back to the default lane. If no abort is required, the car AI continues to pass the slower car until it falls outside of its sensors. At that point, the AI turns back in to the default lane and resumes cruising speed and behavior.

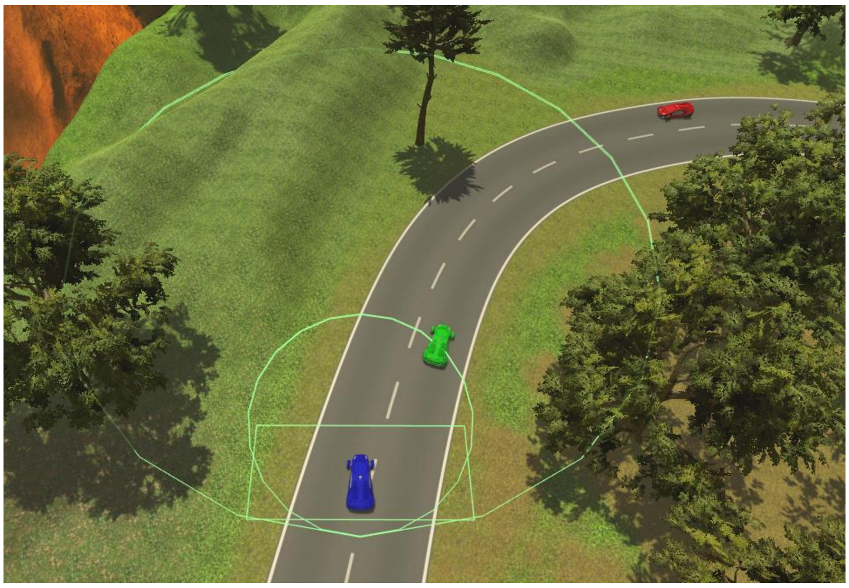

**Figure 5.1.** A newly initiated overtaking maneuver. It will be aborted momentarily as the red car passes into the long range sensor.

In extending the simulation, two intersections were added to the track. When handling the intersection, the cars intersection-sensor checks for approaching cars from both sides. Once a car triggers the sensor, its position and direction are determined. If the other car is determined to be heading away, the AI can proceed to cross the intersection, despite the other car still occupying the sensor area.

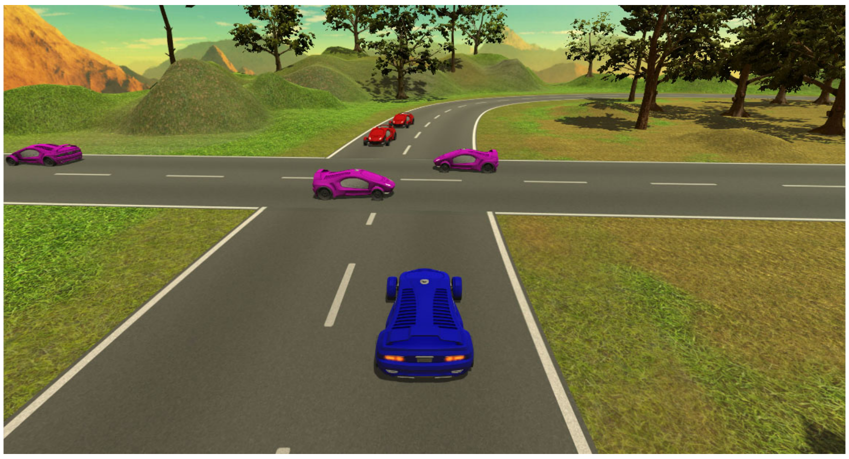

**Figure 5.2.** Blue and red cars waiting for an opening to cross the intersection.

## 5.2 Code architecture

The resulting code for the initial BT implementation was considerably more complex than that of the FSM, due to the many different node-types. An uninitiated reader of the code would likely find the BT considerably more difficult to understand in its entirety from reading the code alone (a diagram would however be quite straightforward). After extending the BT and FSM with the new behavior associated with stopping, intersection handling and horn-honking, the execution flow through the FSM grew more difficult to understand and growing increasingly interconnected.

The BT was initially implemented with the open/closed method (see 3.3.9). While functional, it did not appear to be a particularly practical solution to the interruptibility problem. Each node had to inherit three additional functions with lots of state to keep track of during execution. The state made debugging and reasoning about execution unnecessarily burdensome.

For the sake of completeness and to gain better understanding, the BT was rewritten using an event-driven approach the second time around, which became the final design. The underlying concept is that nodes which require knowledge of any task interruptions are notified when this occurs. For example, composite nodes (Sequences and Selectors) maintain a *current-child* pointer in order to resume execution of a running child rather than incorrectly start from the first child. This pointer needs to be reset to the first child, should another action interrupt one of it's children. This is achieved by event notifications, implemented as the common Observer pattern [39]. Similarly, all AsyncActions are notified when an action is about to run, allowing them to abort their own coroutine, should it be running.

Due to Unity3Ds limited concurrency via coroutines, the execution of the long-running actions had to take place in higher shared context than on node or FSM-state level. In the case of the FSM implementation, this results in an architecture deviation from what one would normally expect with the full language features available. In that case, the action would be executed as a new thread spawned from the state context. However, this had no effect on the execution outcomes.

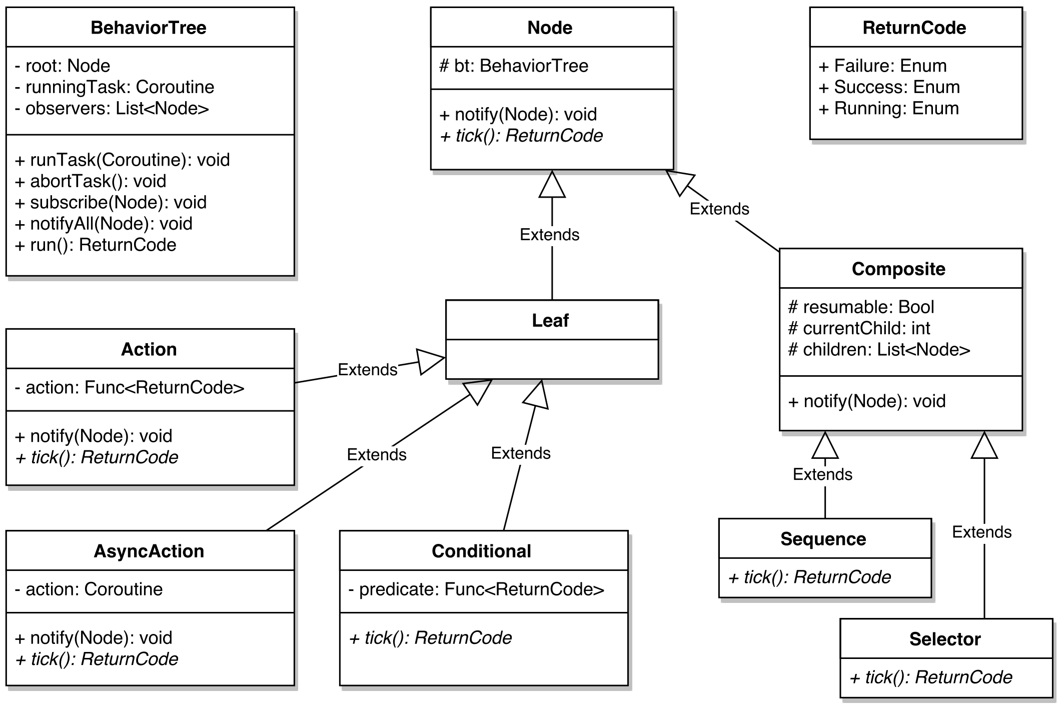

**Figure 5.3.** Structure of the Behavior Tree implementation

## 5.3 Code complexity

After adding a new safety state (one that needed transitions from all other states) to the FSM, every other transition function tediously required modification. In an FSM, adding or removing the states had reverberations throughout the entire code.

In adding the additional behaviors to the BT, very little effort was required. There was minimal risk of introducing an error. The new nodes had no side-effects on the other part of the tree and they could easily be rearranged to change priority. Such a priority rearrangement would have required modification of $n - 1$ transition functions in the FSM. With the BT it was simply a matter of inserting the node at the desired position, which had a negligible on the readability of the code.

### 5.3.1 Metrics

The two complexity metrics were measured at each step of the simulation implementation:

1. After the initial working version with only overtaking maneuvers (*Overtaking*)
2. After stopping behavior was added (*+Stopping*)
3. When the two intersections with crossing traffic had been added (*+Intersection*)
4. After adding intersection detection (*+DetInter*)
5. After adding the ability to honk its horn at stationary cars (*+Honk*)

**Cyclomatic complexity**

| | BT | FSM |
|---|---|---|
| Overtaking | 74 | 41 |
| +Stopping | 79 | 53 |
| +Intersection | 84 | 67 |
| +DetInter | 92 | 86 |
| +Honk | 98 | 112 |

**Table 5.1.** Measured values for cyclomatic complexity (less is better).

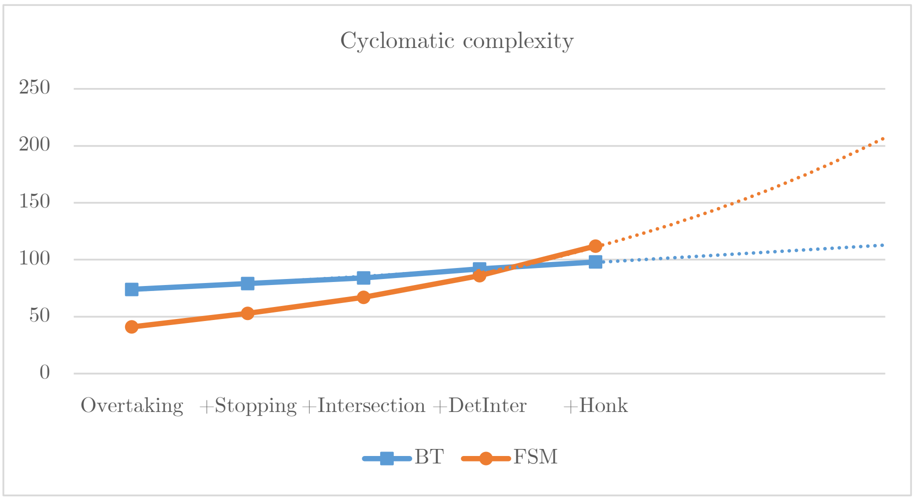

**Figure 5.4.** Plot of Cyclomatic complexity metrics with extrapolation (less is better).

Adding additional behavior to the BT increased the CC only linearly, while the FSM is super-linear. This fits well with the knowledge that in the worst case, the number of transitions in an FSMs grows by the order of $\mathcal{O}(n^2)$ [11], whereas BTs grow by $\mathcal{O}(n)$. With the increase in CC there would be additional tests required to ensure full coverage. We see that the CC of the FSM surpasses that of the BT after the fifth measurement. This indicates that the FSM has grown more complex than the BT, despite them implementing the same behavior.

**Maintainability index**

| | BT | FSM |
|---|---|---|
| Overtaking | 81,0 | 85,4 |
| +Stopping | 80,3 | 83,4 |
| +Intersection | 80,1 | 81,4 |
| +DetInter | 79,8 | 79,3 |
| +Honk | 79,6 | 77,3 |

**Table 5.2.** Measured values for maintainability index (higher is better).

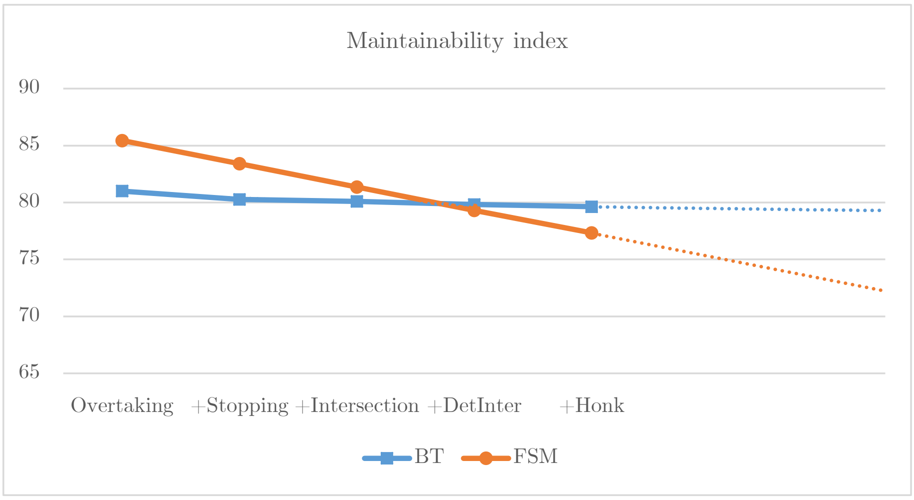

**Figure 5.5.** Plot of Maintainability index metrics with extrapolation (higher is better).

Again, the FSM initially was rated more maintainable than the BT but degraded quickly once more functionality was added and eventually declined past the BT. The downward slope of the FSM is quite significant. Assuming the linear downward trend holds for the FSM it's maintainability would be classified as "low" rather quickly.

### 5.3.2 Structure

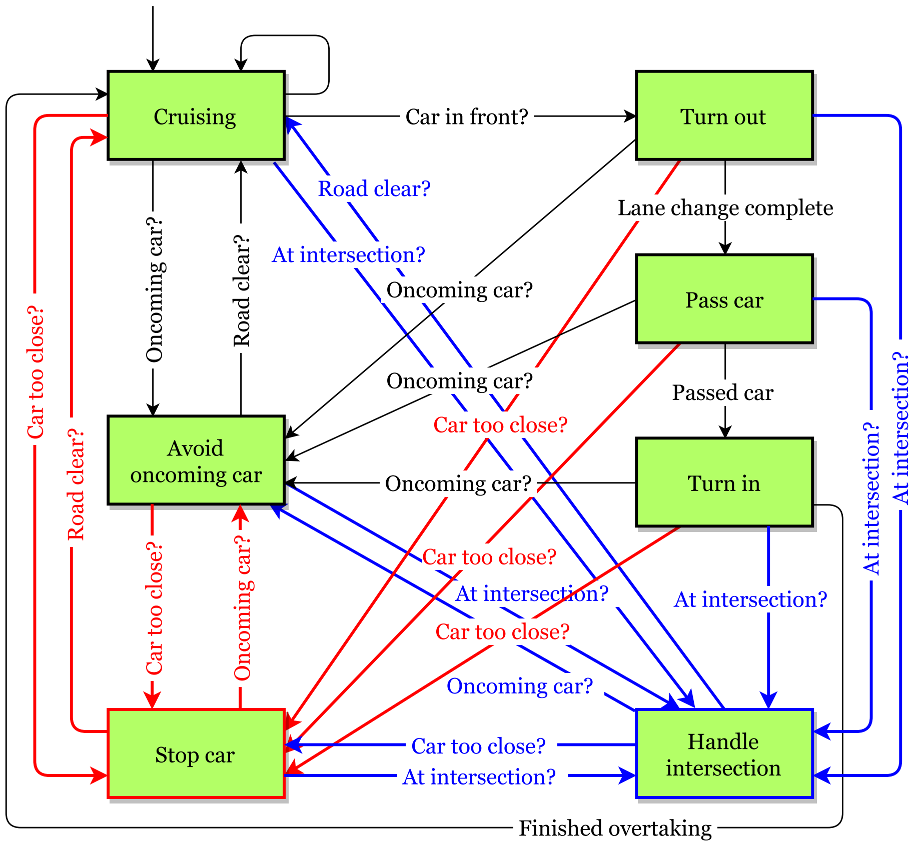

**Figure 5.6.** Updated state machine used in simulation. Note the many transitions and states colored red and blue, which were added due to the extended behavior.

Looking at this state diagram it is not possible to tell the priority of the transitions from any state. What if several transition conditions are met at the same time? To know which transition is actually takes precedence, one has to look at the actual code, which is a large deficiency. One would have to annotate each transition with a number, which would make it even more cluttered. Note that the final two features added are not represented in this diagram. It is clear from the figure that adding just a few additional states yet again would make the diagram incredibly cluttered.

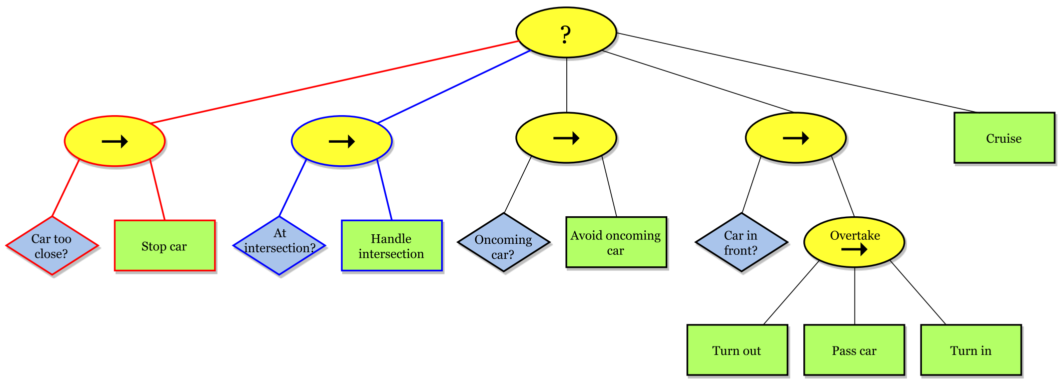

**Figure 5.7.** Updated Behavior Tree used in simulation. Red and blue nodes are new. Note how easy it is to understand the execution flow and priority of behaviors compared to the state machine.

The two added behaviors do not add much complexity to the diagram. Their place in the priority hierarchy is also easily comprehensible. The contrast in readability to the extended FSM diagram is substantial. Again, the two final features are omitted from the figure.

## 5.4 Performance

Performance-wise, there was no measurable difference in frame rate due to the graphics rendering dwarfing the behavior logic in terms of processing power. While it's easy to imagine the BT being less efficient due to visiting more nodes on every tick compared to an FSM (the FSM only ever visits one state per tick), such a conclusion would be missing the fact that the conditional logic in the FSM's transition calculations are the equivalent of the Conditional nodes in a BT. However, if the BT ends up executing a Sequence with multiple instant Actions (i.e. actions that complete instantly and cannot return *running*) then the BT could indeed be more demanding than an FSM, given the same tick frequency.

Memory consumption was not directly measured in this simulation on account of their tiny footprint. Nevertheless, BTs are known to push a lot of function calls to the stack (which could cause a stack overflow) because of its inherent architecture [21], unlike FSMs where one could conceivably even instantiate states on demand.

# Chapter 6

# Discussion

Due to the autonomous driving application considered in this thesis, the central concern is safe and reliable operation. Lower complexity of the software is a means of helping developers reason about the system, thereby lowering the risk for bugs and unexpected behavior. Keeping system complexity to a minimum can therefore be considered a goal in itself, which will be the presumption in this chapter.

## 6.1 The BT architecture

While the classical Behavior Tree is an elegant concept, the academic treatises largely omit the implementation details necessary to consider it from a perspective of fully reactive systems. I.e., in contexts where any behavior or task must be instantly interruptible with a higher-priority one, while ensuring that next visit to the same task will not be broken. Perhaps it is considered an engineering problem and not interesting to BTs as a whole. However, one of the attractive things about the naive blocking BTs is their lack of state or dependence on external variables. The flow of the execution is controlled entirely by the nodes, passing returned values up and down the tree. That type of statelessness is what makes them so easily applicable in terms of reusability and modularity, since there is no state to be mindful of. Specifically, even the same instances of nodes and subtrees can be reused elsewhere in the BT multiple times by using a simple pointer.

However, as the BT is adapted to enable concurrent tasks to be executed separately from the ticking thread, this complicates reusability. To illustrate this with an example, imagine ActionB has a thread or coroutine running and has returned it's running-status to the root. In the following tick, the higher precedence ActionA wishes to execute instead, and ActionB must be aborted. ActionA cannot know which other (if any) task is executing and is hence required to delegate the abortion to, lets say, the BT itself. Additionally, any composite node which has a child node interrupted needs to be reset in order to begin iteration from the first child again in the next visit. In this scheme, the BT is required to in some way have knowledge of which task is currently executing.

In this implementation, there seemed no way around this. It was an unfortunate detriment to the modularity, making reusing the same subtree (the very same instance) not possible. While it is in no way a death-blow to modularity or the suitability as an architecture for autonomous driving, but a notable by-product of the interruptible adaption. Adding state (in the form of variables) increases complexity, making it harder for developers to reason about the architecture, which could introduce bugs.

The aforementioned adaption is an engineering matter. For the implementation work in this thesis, an event-driven approach was chosen. This was deemed to be an efficient and elegant way to reset only the nodes requiring it, in the event of an interrupting task. Another approach considered, keeping track of their open/closed status and iterating over all nodes to close them, was deemed inefficient and resulted in needlessly cluttered code. Both versions were implemented but event-driven replaced open/close in the final code.

An approach which could possibly solve the statefullness problem is using recursive interrupt-calls sent from the soon-to-execute node, propagating up and down to all nodes to the right in the diagram. Though, this would need to be through some means of signaling other than actually returning from the node. Also, there could possibly be timing-issues related to this design where the executing node could not be sure if it's safe to execute yet. Unfortunately this type of design was not considered at the outset of this thesis and is therefore not exhaustively explored, but it may be the most promising one.

Once the underlying architecture was in place, adding nodes and behavior was relatively simple and straight-forward. Adding a visual tree editor tool could enable non-technical behavior designers to work independently with extending the tree, which would be a large benefit over FSMs.

One significant concern, perhaps the biggest one, when designing a BT is the highest-priority behavior (farthest to the left in the tree) where typically one would place self-preservation and safety mechanisms. Careful consideration needs to be taken to which other states the BT host (in this case a vehicle) may find itself in when this self-preservation behavior is called upon. If every other behavior can be interrupted by the left-most one, the safety-task needs to be designed in such a way as to be able to take over even if the car is in the middle of a peculiar or hazardous state. For robots, it is easy to imagine hazardous states which require "undoing" before performing another task. In cars, one may need some creativity to find such a problem, but it might still become an issue someday. Behavior Trees seem not well suited to such "undo actions", whereas it is easier to imagine an elegant remedy in a finite-state machine. In BTs there is execution supposed to take place in two different places, one in the new task and one undo-task, which creates a timing issue where one would like to be sure that the undoing is finished before executing the new task is started. In an FSM, this is trivial due to the fact that the transition can wait for the undo-task to complete (although to not adversely effect reactivity, this undo action needs to terminate quickly). For this reason, it is a good idea to design the self-preservation behavior as one capable of reaching a safe default state regardless of which other task it supersedes, which is challenging.

This introduces the real problem of having to be aware of what every other behavior is up to. It is an impediment in a development environment with many team members working on separate parts of the tree. Granted, this is a problem that FSMs also suffer from.

A large benefit of the BT architecture is the tree structure. The tree implicitly encodes the transitions between nodes, meaning that adding or removing nodes does not require editing or extending of other nodes. In FSMs, removing or adding a state entails editing of any number of other states' transition logic, which can be both laborious and a source of mistakes. This was made obvious by the extension of the simulation performed in this work, where the transition explosion was evident even when working at this small scale. Performing the equivalent extension in the BT architecture was effortless in comparison, with little risk of losing an overview of the behavior of the system.

The work of extending the BT for the added behaviors was absolutely trivial, which shows great promise. In terms of complexity, the BT was barely affected. Being able to clearly visualize where the new behaviors are situated in terms of the priority hierarchy and being able to move and insert at any point without side-effects is also hugely beneficial. In an FSM you cannot easily grasp which behaviors take precedence without looking into the code, and even then it is tough to get a bird's-eye view.

## 6.2 Finite-state machines

Compared to the BT, the FSM architectural pattern is quite simple to implement. Both reasoning about the flow of execution and implementing it was accomplished in barely any time. Granted, it was made easier by the experience gained during the implementation of the BT, but only to a point. The FSM required for this particular simulation was initially much less complex than the corresponding BT, but grew less elegant as the new behaviors were added. In a much larger and complex behavior model, at some point the FSM would experience substantial growing pains and become very difficult to maintain [23], as hinted by the metrics.

FSMs have an advantage over BTs in terms of allowance for ad hoc solutions. Take for example the case of exiting hazardous states before transitioning to another behavior, as discussed in 6.1. It is easy to add such behavior in an FSM since the execution is taking place at the same location responsible for wishing to transition elsewhere. However, this flexibility can surely also be a detriment if exploited carelessly.

When it comes to running multiple tasks or behaviors concurrently, FSMs struggle almost by definition. While it was not explored in great detail in this thesis, BTs offer considerably better support for concurrent tasks through the use of *Parallel* nodes.

One of the main points against FSMs, the need to duplicate the same transitions many states, is greatly alleviated (if not altogether solved) by inheriting transitions from superstates. The transitions are just about the only reusable parts in an FSM. This describes the hierarchical finite-state machine architecture and it appears to be the most widely used architecture in real-world autonomous vehicles.

In the implementation produced for this work, the states where expressed as tasks, much like the leaves in the BT were behaviors or actions. FSMs have been argued to be difficult to make goal-oriented, while BTs are considered better at this [23]. The difference may in some part lie in the way one chooses to express their FSM states. This implementation went from action to action, rather than using the mentality of dwelling in "states". Having this in mind might aid in the use of FSMs. Grouping states into superstates meant to achieve a particular goal is another tool to consider.

## 6.3 Unity3D

Using Unity3D as the platform for the simulation had both pros and cons. The platform enables the creation of realistic environments (both in terms of physics and visuals) with relatively little effort. In that aspect it is an excellent tool to prototype ideas and realize simulations. On the other hand, conforming to the quirks of the Unity3D platform imposed some design limitations which would not have been present otherwise, if for example one had programmed this for an actual vehicle on a blank platform. However, the limitations didn't result in any concrete degraded performance or failed behavior, as best I can tell. Instead, it exhibited itself as an unnecessarily inelegant structure of the BT software. With proper access to threading or more permissive coroutines, the implementation would have been more straight-forward. Some of the issues can be remedied by implementing a proper manager for coroutines, which was not part of the scope of this thesis. With that in mind, the platform can still be considered suitable for simulations of this kind.

## 6.4 Limitations of this work

The implementation produced likely isn't complex enough to show the real benefits, disadvantages or differences between BTs and FSMs. The simplistic scenario modelled in the simulation made the BT architecture seem excessive. To more accurately assess the suitability of BTs compared to (H)FSMs, a more ambitious and complex simulation would have been necessary.

Additionally, this work has primarily pitted Behavior Trees against FSMs, when in fact the predominant competing architecture is HFSMs. For a project this small however, the difference would not have been significant. All states would have inherited from a superstate containing the transition check to *Avoid obstacle*. This work did not devote much space to the concepts of Parallel nodes and Decorator nodes, which are powerful concepts in the BT toolbox. Decorators have for example been proposed as means to synchronize multiple BT agents for collaborative tasks [21]. These nodes in all likelihood offer benefits to designing a BT for autonomous driving that was not given consideration here.

As mentioned previously, the recursive abort-call design of the BT implementation could have been a superior design over the chosen event driven one.

Another aspect which could have been explored in greater detail was reusability and modularity, which has been argued as one of the major benefits of BTs [11]. This implementation did not attempt to re-use any subtree or nodes, which perhaps would have offered more insight into the suitability and implementation of BTs.

## 6.5 Future work

As previously mentioned, modelling a more complex driving scenario requiring more complex driving behavior would give better insight into the suitability of Behavior Trees for autonomous driving. By scaling up significantly the differences between BTs and FSMs would presumably become more pronounced, as well as more relevant to the actual application considered.

Behavior Trees have been shown to be suitable for automatic learning and generation [27–29]. Conceivably, this could have implications for autonomous driving, which would be interesting to explore.

## 6.6 Ethical aspects of autonomous driving

The ultimate cost of faulty software in this realm is human lives. For this reason, it is important to approach the development of this technology from many angles and with different techniques. Exploring software architecture with its roots in computer games could prove worthwhile.

Since we have reason to believe computers can become better, more attentive, more consistent drivers than humans [40], it could be argued that it is our moral imperative to quickly develop this technology and hand over the wheel to the AIs. A failure to do so would result in needless deaths and injuries.

From a judicial perspective, it is unclear who is the responsible party if a driverless car malfunctions causes an accident. More problematic still, should the AI be implemented with calculated sacrifices of human lives in order to save more? Does the manufacturer have a responsibility to the customer, to put the car's occupant's life before other? There is much discussion yet to take place in the public sphere on such topics. The laws in this area are still non-existent, even though the age of autonomous cars is practically upon us.

# Chapter 7

# Summary

The purpose of this thesis was to investigate the suitability of using Behavior Trees as the software architecture for the behavioral subsystem in autonomous driving.

Currently, FSMs and HFSMs are the most commonly used architectures for autonomous driving.

BTs originated from the computer game industry where they are widely used today in the AI of non-playing characters.

Behavior Trees are formulated as directed graphs with a tree structure. The top-most node, denoted as the root node, has one or more child nodes, whom in turn can also have children. Nodes which have children are sometimes referred to as composite nodes. A child node with no children of its own is called a leaf. The root node is the only one without any parent. Execution of the tree starts at the root node and proceeds through the tree in a depth-first fashion, with branching dictated by nodes returning either Success, Failure or Running.

Robotics research has shown interest in BTs in the past couple of years, where the major benefits of BTs have been argued to be modularity and reusability.

In adapting BTs for interruptible tasks, some of the reusability is negatively impacted. For example, some nodes cannot be reused multiple times in the same tree since they now contain stateful variables.

In implementing a reactive BT one must put extra consideration into the design of the left-most node (the highest priority behavior), which is typically where self-preservation and collision avoidance should be placed. Since any other node can be interrupted mid-execution, it could be challenging to design this correctly.

Despite these two caveats, the BT architecture shows great promise in coping with the increasing complexity associated with an autonomous driving system. In the comparison to the FSM, the BT clearly scaled better in terms of readability and complexity.

This thesis presented a driving simulation produced using Unity3D engine as a proof of concept and to evaluate BTs and FSMs as the behavioral layer. As further functionality was added, the FSM quickly showed signs of becoming more difficult to maintain, while the BT did not. This difference would be further exacerbated as the system grows larger.

Both architectures produced the same driving behavior but the BT is deemed to be better suited for autonomous driving due to its benefits in terms of readability, maintainability and reusability.

# Bibliography

[1] Google self-driving car project. https://plus.google.com/+SelfDrivingCar/posts/iMHEMH9crJb. *(last accessed: 2015-08-23).*

[2] Tesla Motors, your autopilot has arrived. http://www.teslamotors.com/blog/your-autopilot-has-arrived. *(last accessed: 2016-01-12).*

[3] World Health Organization. Global status report on road safety 2013. http://www.who.int/iris/bitstream/10665/78256/1/9789241564564_eng.pdf. *(last accessed: 2015-08-23).*

[4] IHS Automotive. Autonomous cars - Not if, but when. *Automotive Technology Research*, 2014.

[5] Damian Isla. Handling complexity in the Halo 2 AI. In *Game Developers Conference*, volume 12, 2005.

[6] Michael Mateas and Andrew Stern. Façade: An experiment in building a fully-realized interactive drama. In *Game Developers Conference*, volume 2, 2003.

[7] Damián Isla. Building a better battle. In *Game Developers Conference, San Francisco*, 2008.

[8] Alex J Champandard, Michael Dawe, and David Hernandez-Cerpa. Behavior trees: Three ways of cultivating game ai. In *Game Developers Conference, AI Summit*, 2010.

[9] Tomáš Plch, Matej Marko, Petr Ondrácek, Martin Cerny, Jakub Gemrot, and Cyril Brom. An ai system for large open virtual world. In *Proceedings of Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment. This proceedings*, 2014.

[10] Unreal Engine 4. http://www.unrealengine.com/what-is-unreal-engine-4. *(last accessed: 2016-01-12).*

[11] Petter Ögren. Increasing modularity of UAV control systems using computer game behavior trees. In *AIAA Guidance, Navigation and Control Conference, Minneapolis, MN*, 2012.

[12] Ernst D Dickmanns and Alfred Zapp. Autonomous high speed road vehicle guidance by computer vision. In *International Federation of Automatic Control. World Congress (10th). Automatic control: world congress.*, volume 1, 1988.

[13] Ernst Dieter Dickmanns, Reinhold Behringer, Dirk Dickmanns, Thomas Hildebrandt, Markus Maurer, Frank Thomanek, and Joachim Schiehlen. The seeing passenger car 'vamors-p'. In *Intelligent Vehicles' 94 Symposium, Proceedings of the*, pages 68–73. IEEE, 1994.

[14] Mohr Davidow Ventures. Stanley: The robot that won the DARPA grand challenge. *Journal of field Robotics*, 23(9):661–692, 2006.

[15] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, MN Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[16] Chris Urmson, J Andrew Bagnell, Christopher R Baker, Martial Hebert, Alonzo Kelly, Raj Rajkumar, Paul E Rybski, Sebastian Scherer, Reid Simmons, Sanjiv Singh, et al. Tartan racing: A multi-modal approach to the darpa urban challenge. *Robotic Commons*, 2007.

[17] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, et al. Junior: The stanford entry in the urban challenge. *Journal of field Robotics*, 25(9):569–597, 2008.

[18] Arda Kurt and Ümit Özgüner. Hierarchical finite state machines for autonomous mobile systems. *Control Engineering Practice*, 21(2):184–194, 2013.

[19] Tichakorn Wongpiromsarn and Richard M Murray. Distributed mission and contingency management for the DARPA urban challenge. In *International Workshop on Intelligent Vehicle Control Systems (IVCS)*, 2008.

[20] A. Champandard. Understanding Behavior Trees. *AiGameDev.com*, 6, 2007.

[21] Alejandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ögren. Towards a unified behavior trees framework for robot control. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 5420–5427. IEEE, 2014.

[22] Michele Colledanchise, Alejandro Marzinotto, and Petter Ögren. Performance analysis of stochastic behavior trees. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 3265–3272. IEEE, 2014.

[23] Andreas Klöckner. Behavior trees for UAV mission management. In *GI-Jahrestagung*, pages 57–68, 2013.

[24] Michele Colledanchise and Petter Ogren. How behavior trees modularize robustness and safety in hybrid systems. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1482–1488. IEEE, 2014.

[25] Danying Hu, Yuanzheng Gong, Blake Hannaford, and Eric J Seibel. Semi-autonomous simulated brain tumor ablation with ravenii surgical robot using behavior tree. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 3868–3875. IEEE, 2015.

[26] Andreas Klöckner, F van der Linden, and D Zimmer. The modelica behaviortrees library: Mission planning in continuous-time for unmanned aircraft. In *Proceedings of the 10th International Modelica Conference*, number 96, pages 727–736, 2014.

[27] Chong-U Lim, Robin Baumgarten, and Simon Colton. Evolving behaviour trees for the commercial game DEFCON. In *Applications of evolutionary computation*, pages 100–110. Springer, 2010.

[28] Michele Colledanchise, Ramviyas Parasuraman, and Petter Ögren. Learning of behavior trees for autonomous agents. *arXiv preprint arXiv:1504.05811*, 2015.

[29] Diego Perez, Miguel Nicolau, Michael O'Neill, and Anthony Brabazon. Evolving behaviour trees for the Mario AI competition using grammatical evolution. In *Applications of Evolutionary Computation*, pages 123–132. Springer, 2011.

[30] Kirk YW Scheper, Sjoerd Tijmons, Coen C de Visser, and Guido CHE de Croon. Behaviour trees for evolutionary robotics. *arXiv preprint arXiv:1411.7267*, 2014.

[31] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[32] Ian Millington and John Funge. *Artificial intelligence for games*. CRC Press, 2012.

[33] Alexander Shoulson, Francisco M Garcia, Matthew Jones, Robert Mead, and Norman I Badler. Parameterizing behavior trees. In *Motion in Games*, pages 144–155. Springer, 2011.

[34] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[35] Arthur H Watson, Thomas J McCabe, and Dolores R Wallace. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST special Publication*, 500(235):1–114, 1996.

[36] Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 337–344. IEEE, 1992.

[37] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[38] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[39] Wikipedia, Observer pattern (software design pattern). `https://en.wikipedia.org/wiki/Observer_pattern`. *(last accessed: 2016-01-12)*.

[40] Virginia Tech transportation institute, Automated vehicle crash rate comparison using naturalistic data. `http://www.vtti.vt.edu/featured/?p=422`. *(last accessed: 2016-01-12)*.