Universität Stuttgart

Yase - A Holistic Approach for Intelligent Vehicle Simulation

Von der Fakultät für Energie-, Verfahrens- und Biotechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Ingenieurwissenschaften (Dr.-Ing.) genehmigte Abhandlung

Vorgelegt von

Max Paul Bauer-Marquardt

aus Bühl, Baden-Baden

Hauptberichter: Prof. Dr.-Ing Dr. h.c. (DonNTU) Dr. h.c. (RAS/Sib) Hon. Prof. (RAS/Sib) Michael M. Resch

Mitberichter: Prof. Dr.-Ing Hans-Christian Reuss

Tag der mündlichen Prüfung: 21.01.2025

Institut für Höchstleistungsrechnen der Universität Stuttgart

2025

Abstract

In the automobile industry there is an ongoing fundamental change towards more intelligent vehicles. This is mainly driven by high hopes regarding economical, safety and ecological benefits. However, the step towards intelligent vehicles lead to new challenges in the development and validation of such. This arising complexity cannot be handled any more without proper simulation. However, current simulation approaches still suffer from practical limitations and requirements are not yet fully solved. On the one hand, simulation must be modular, maintainable, and adaptable for manyfold use cases and test levels. On the other hand, the inconsistent landscape of proprietary and standardized formats enforces massive redundant work. This thesis provides a framework concept with several contributions to solve these needs.

First, a holistic approach is developed, which solves the need for modularity as well as the easy combination of multiple input formats and simulator solutions. The architecture is designed according to the established concept of programming language compilers and is developed with state-of-the-art software design standards. The Yase (Acronym for Yet Agnostic Scenario Engine / Yet Another Scenario Engine) framework is a three-stage architecture, which allows to connect different input formats to different simulators or sub scopes of them. Hereby all input formats are translated via front ends into an intermediate format, which resides in the middle end. Against this middle end interface, simulators, or sub scopes of them can be implemented independent of the actual input format.

For the middle end as the central part of the framework, it is shown what kind of controller architectures are adaptable. Here, multiple controller architectures are analyzed and compared against each other with the help of ISO defined quality goals. The best fitting concept of behavior trees is then analyzed further regarding their different implementation strategies and a survey of existing open-source projects is presented. Since none of the existing projects fit all the domain specific needs, a Yase specific implementation called Agnostic Behavior Tree (ABT) is presented.

Based on the middle end work, an exemplary back end is built up to demonstrate all derived domain specific use cases. Here, it is shown the first time that the behavior tree concept is capable enough to express all relevant intelligent vehicle simulation use cases. First, the implementation of a concrete scenario is shown, as well as extensions such as combinations with evaluations and the usage within an exploitative search. Last, the implementation of driver and traffic models with the same behavior nodes is presented. With a massive traffic simulation, the potential for exploratory searches is highlighted.

At the end, it is demonstrated with the OpenSCENARIO DSL front end, how a state-of-the-art scenario language standard can be integrated into the framework. It is explained how the translation with a transpiler to an executable behavior tree instance works. Furthermore, OpenSCENARIO DSL language specific constructs are discussed, and implementation strategies are presented.

Zusammenfassung

Die Automobilindustrie ist in den vergangenen Jahren einem fundamentalen Wandel unterworfen hin zu immer intelligenteren Fahrzeugen. Dieser Trend ist getrieben sowohl von hohen ökonomischen und ökologischen, als auch von Erwartungen im Hinblick auf die Erhöhung der Verkehrssicherheit. Der Weg zu solch hochintelligenten Systemen ist jedoch gepflastert mit zahlreichen ungelösten Herausforderungen bei der Entwicklung und der Validation. Die damit einhergehende Komplexität kann nicht mehr ohne Simulation gemeistert werden. Jedoch erfüllen die aktuellen Simulationsansätze nicht alle Anforderungen optimal. Auf der einen Seite müssen Simulationslösungen modular, wartbar und anpassbar sein an unterschiedliche Anwendungen und Test-Level. Auf der anderen Seite gibt es aktuell einen massiv redundanten Arbeitsaufwand für die verschiedenen Simulationslösungen aufgrund einer Vielzahl proprietärer und standardisierter Formate.

In dieser Arbeit wird das Yase Framework-Konzept vorgestellt, mit dem diese Probleme gelöst werden können. Zuerst wird dazu ein ganzheitlicher Ansatz entwickelt, der zum einen Modularität und zum anderen die Kombination von verschiedenen Input-Formaten sowie letztlich Simulatoren ermöglicht. Dabei orientiert sich die Architektur an Programmiersprachen-Compilern und an neusten Software-Design-Standards. Das Ergebnis ist ein Framework namens Yase (Akronym für Yet Agnostic Scenario Engine, beziehungsweise Yet Another Scenario Engine). Das entwickelte Drei-Phasen-Konzept erlaubt es, unterschiedliche Input-Formate mit verschiedenen Simulatoren, beziehungsweise Subsets von Simulatoren zu verbinden. Dazu werden alle Input-Formate von Frontends in ein Zwischenformat übersetzt, welches im Middleend verortet ist. Gegen dieses Middleend-Interface können die Simulatoren, oder Subsets von diesen, implementiert werden. Sie sind daher unabhängig vom tatsächlichen Input-Format.

Für das Middleend als zentralem Teil der Architektur wird aufgezeigt, welche Controller-Architektur am besten geeignet ist. Dafür werden verschiedene Architekturen analysiert und mithilfe von ISO-definierten Qualitätszielen miteinander verglichen. Die am besten zum Problem

passende Lösung, das sogenannte Behavior-Tree-Konzept, wird daraufhin weiter analysiert hinsichtlich schon existierender Open-Source-Projekte und Implementierungsstrategien. Da keines dieser Projekte den Ansprüchen der vorliegenden Aufgabenstellung genügt, wird daraufhin eine angepasste Yase Behavior-Tree-Implementierung namens Agnostic Behavior Tree (ABT) vorgestellt.

Basierend auf dem Behavior Tree des Middleends wird ein exemplarisches Backend implementiert, um Domain-spezifische Anwendungen demonstrieren zu können. Hier wird erstmals aufgezeigt, dass damit alle relevanten Simulationsanwendungen für die Entwicklung intelligenter Fahrzeuge ausgeführt werden können. Zuerst wird die Implementierung eines konkreten Scenarios dargestellt sowie Kombinationen dieses Szenarios mit Evaluierungen. Basierend auf diesem Szenario wird die Anwendung in einer explorativen Suche dargelegt. Daraufhin wird mit den gleichen Bausteinen die Implementierung von Fahrer- und Verkehrsmodellen demonstriert. Das Potential solcher Modelle bei der explorativen Suche wird dann mittels einer umfangreichen Verkehrssimulation aufgezeigt.

Schließlich wird mit Hilfe eines OpenSCENARIO-DSL-Frontends beschrieben, wie eine komplexe Szenario-Sprache mithilfe des Yase Framework implementiert werden kann. Dabei wird erklärt, wie mithilfe eines Transpilers aus Szenario-Dateien ausführbare Behavior-Tree-Instanzen erzeugt werden können. Des Weiteren werden OpenSCENARIO-DSL-spezifische Sprachkonstrukte diskutiert und Implementierungsstrategien vorgestellt.

Acknowledgements

"No significant research work can be completed done without the contribution of many people." - Unknown. This statement is definitely valid for this thesis. I would like to thank all, who supported me in this journey.

First and foremost, I want to express my gratitude from the depths of my heart to my fiancée Kira Marquardt for the ongoing support, trust, and patience. It has been years with long nights, which are challenging for relationships, especially the closest ones. Your ongoing support gave me the energy to pull of this thesis. This is also true for my family and friends, who I like to thank for their support as well.

A special thanks is towards my Professor Michael Resch and my Bosch company supervisor Frank Lattemann, who facilitated and supported me during this thesis. The trust, support and given freedom were the foundation for the creativity within this thesis. Furthermore, I like to thank all the colleagues I was working with over the last years. May that be colleagues from within my team, from other companies due to the Yase open-source activities or with standardization projects such as SetLevel or Asam OpenSCENARIO DSL. I want to thank especially those closest ones, who supported me in the development of Yase or gave as active users constant feedback, ideas, and reviews. The list of names is hereby too long to mention, but they will know if they are meant. A dedicated acknowledgment is also towards Anthony Ngo, who also wrote a thesis in the same team and was a partner in crime during this time. The mutual support and exchange were essential for this thesis.

Last, but not least, in the spirit of Calvin Cordozar Broadus Jr: "I want to thank me. I want to thank me for believing in me, for doing this hard work. And I want to thank me for never quitting" - and especially for writing this thesis without any usage of AI such as ChatGPT, purely with my own skills and creativity.

Contents

List of Figures

List of Tables

List of Acronyms

ABS Antilock Braking System

ABT Agnostic Behavior Tree

ACC Adaptive Cruise Control

AD Autonomous Driving

ADAS Advanced Driver Assistance System

AEB Automatic Emergency Braking

ANTLR ANother Tool for Language Recognition

ASAM Association Standardization of Automation and Measuring Systems

AST Abstract Syntax Tree

ATS Agnostic Type System

BNF Backus–Naur form

BT Behavior Tree

CD Continuous Development

CI Continuous Integration

DSL Domain Specific language

DT Decision Tree

EBNF Extended Backus–Naur form

EDA Event-driven architecture

ESC Electronic Stability Control

FSA Finite State Automaton

FSM Finite State Machine

GCC GNU Compiler Collection

HIL Hardware in the Loop

H-FSM Hierarchical Finite State Machine

LLVM Low Level Virtual Machine

NPC Non Player Characte

SIL Software in the Loop

SUMO Simulation of Urban MObility

VTD Virtual Test Drive

V2V Vehicle-to-Vehicle

V2X Vehicle-to-X

YASE Yet Agnostic Scenario Engine / Yet Another Scenario Engine

Chapter 1

Introduction

The development towards more intelligent vehicles is together with the change from combustion engines to electric engines the most significant trend in the automobile industry. Intelligent vehicles take over more responsibilities from human drivers and economic and ecological benefits due to less accidents, optimized traffic flow, car sharing, etc. are expected. However, the rising complexity of intelligent vehicles requires a more complex development and validation. In the following, general development challenges of such systems are elaborated. Subsequent, it is shown that simulation is a crucial element to tackle the rising complexity. Here, the state of the art is presented, and the current limitations are discussed. Based on this, a problem statement for holistic simulation frameworks is defined. This includes a requirement and quality goal analysis with the help of international standards and software architecture templates. Last, the contribution of this thesis with the Yase concept to solve these holistic needs are discussed and the outline of this thesis is presented.

1.1 Intention Behind Intelligent Vehicles

In the last decades a development towards an increased vehicle automatization can be observed [1], [2]. At the beginning, small systems such as the Anti-lock Braking System (ABS) were introduced to reduce skid while braking. Further Advanced Driver Assistance Systems (ADAS) such as Electronic Stability Control (ESC) were added later to improve overall safety of the drivers. With time, the focus opened from safety to comfort as well. Next to safety systems such as Automatic Emergency Braking (AEB), other systems focused on increasing comfort as the Adaptive Cruise Control (ACC). In parallel to the automatization of the vehicle systems itself, vehicles are also getting more connected to benefit from synergy effects. Vehicle-to-Vehicle (V2V) communication between vehicles allows to predict and forecast situations better. Vehicle-to-X (V2X) communication with infrastructure enables the interaction with the static environment such as traffic lights. The overall goal of these developments are Autonomous Driving (AD) vehicles, which can take over all responsibilities from the driver and integrate themselves in the surrounding environment in the most comfortable, safest, economic, and ecological way.

The conglomerate of these developments is summarized in the following thesis as intelligent vehicle development. The reasons for this trend within the automotive industry towards full vehicle automatization is hereby manifold. On the economic side car manufacturers try to segregate themselves from other competitors by offering more advanced assistance systems. These assistance systems can increase the driving comfort and the safety of passengers. Additionally, full automatization towards autonomous driving levels allow to exploit new market fields. AD vehicles can be deployed as robo cabs, which allows manufacturers to become mobility providers. With resource efficient utilization with less idle states of such AD vehicles, even high development costs can be amortized fast. Despite these economic interests, there is also a general interest of the society in intelligent vehicles. The system automatization promises to reduce the number of fatal accidents, injuries and collision related financial damages. Also, less traffic is promised, as better utilization and organization of the traffic could lead to improved traffic flow. And last, there is an expectation of a more ecological and resource efficient transportation approach, as the traffic flows could be improved, and shared resources lead to overall less consumption.

1.2 Development and Validation Challenges

While intelligent vehicles are desirable in general from a safety, economic and ecological perspective, there are still challenges which must be solved first. The development of such intelligent vehicles is an ongoing process, and it is getting more and more complex. This is caused by two main challenges: On the one hand the increasing variety of intelligent vehicle features to develop and the resulting large domain space to test. And on the other side the agile software development environment with continuous development and testing and an increased testing level depth.

1.2.1 Increasing Problem Domain Space

With more and more intelligent vehicles, the problem domain space gets also more complex. Simple safety systems such as Automatic Emergency Braking (AEB) may be restricted to simple domain spaces with highway similar environments and no lane interpretation at all. Full AD systems on the other hand must be able to handle significant more situations. As an example, this includes consideration of all surrounding lanes with an appropriate scene understanding, interpretation of regulatory elements such as traffic lights and speed limits, prediction of vehicles, pedestrians, bicyclists, emergency vehicles, and any combination of those. This sheer size of this new technology domain space introduces a never seen complexity in the automobile industry. With a continuous test space, each smallest parameter variation or combination can create a new unique situation. Therefore, it is impossible to completely cover the scenario space fully, but only to a certain extent.

Under the assumption, one could span up an adequate domain scenario set D, the set would be extremely large. In [3], the authors claim that it would be necessary to test "11 billion miles to demonstrate with 95% confidence that autonomous vehicles (AVs) are 20% better than humans". Other sources come to similar numbers in the same magnitude of order, stating that such test coverage cannot be provided with realistic capabilities before market launch [4]. However, from this full domain space only a subset R is relevant for testing, as in the majority of scenarios the system under test has no impact on the situation. An example of an irrelevant scenario may be a vehicle behind with enough safety distance performing a lane change, which has no effect on the system under test at all. This set of relevant scenarios can be divided further into a smaller subset O of objective scenarios. In these scenarios the system under test must react over the normal extent to prevent accidents or other incidents. The most important subset within these objective scenarios is the critical subset C. These scenarios are challenging scenarios, which have a high probability to cause a critical or undefined system behavior and must be fixed and developed further. Each of these subsets are distinct smaller than the set in which they reside.

$$D \subset R \subset O \subset C \tag{1.1}$$

Many already faced fatal accidents of intelligent vehicle systems occurred in unexpected situations, as it can be observed in the deadly Tesla crash 2016 [5]. Such situations come only into awareness after they happened as it is not known prior where the potential critical situations are located within the scenario space. Finding such critical scenarios in the full domain set D is a non-trivial problem and is literally like finding a needle in a Haystack. Fur-

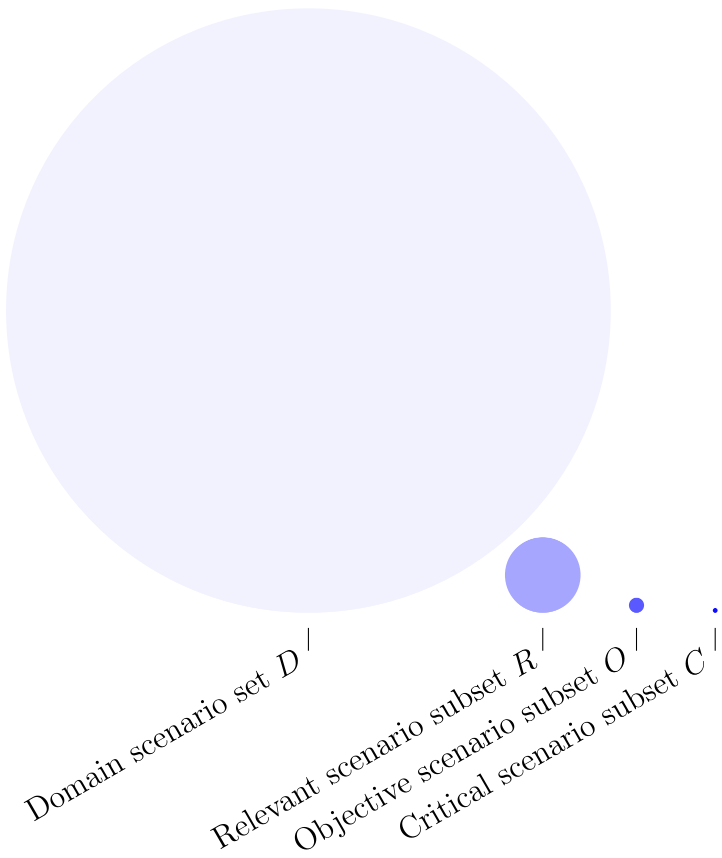


Figure 1.1: Ideogram of the domain space.

thermore, development and testing resources are limited to economical and practical constraints. As stated in [4], the full scenario space is impossible to cover fully. Therefore, it is important to provide solutions, which allow to explore the scenario space in a meaningful way to make statistical assumptions about the general coverage of the remaining scenario space. Furthermore, the areas with critical scenarios should be exploited as much as possible to identify all corner cases.

1.2.2 Agile Development and Testing Level Depth

Besides the sheer size of the domain space to test, another challenge for the automobile industry is the change from a traditional end of line test towards agile software development. Modern software development requires agile development with constant retesting in Continuous Integration & Continuous Development (CI&CD) pipelines. This is due to the fact, that any new commit in a code base has the potential to alter the overall state to a defective state. All previously made tests are in theory therefore not informative anymore. As in large projects with hundreds of developers such code changes come in easily a hundred times a day, these tests must be executed for every change. This is often accompanied by further nightly, weekly and several release stages tests. With the already huge domain scenario space, this need for constant retesting accumulates the numbers of tests even further.

Additionally, the growing complexity with an interplay of more and more components requires more in-depth tests. Final acceptance tests with real test drives with the full system are by far not sufficient enough. Like the development of mechanical engines, each component must be tested in separation before the fully assembled system is tested. This is also an essential requirement of modern software development. Here, the testing levels are typically distinguished into four levels [6]: At first, the smallest units of the compiled code are tested regarding their functionality. The goal is to ensure the intended base unit functionality, such as functions, classes and methods work as expected. This is typically tested with example and corner case values, to capture potential undefined behaviors. Once the base is tested, the first functional components serving a certain self-contained use case can be tested via component testing. As soon as the functionality of each separate component is assured, the overall system can be tested as whole. For this, the components are combined and tested. Hereby errors can be found, which emerge from the complex interplay of the components itself. The last test level are acceptance tests which validate if the customers need is actually fulfilled. Together, all of these levels form up a full testing pyramid as seen in Fig. 1.2.

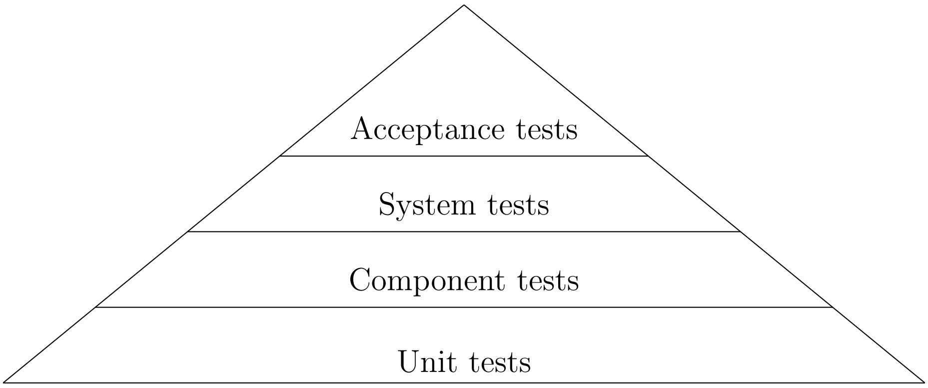


Figure 1.2: Testing pyramid of modern software development.

These testing levels can be mapped for intelligent vehicle development. Acceptance tests are hereby test drives in real traffic, to prove the overall functionality. System tests are specifically created scenarios, in which the system is tested against certain corner cases. In component tests, individual components can be tested in isolation with component specific scenarios. As an example, the planner component, responsible to drive within the environment, can be tested with perfect perception data to ensure its functionality. Last, even the lowest functions such as Kalman filters or perception cluster algorithms can be tested as well in isolation.

1.3 Simulation as Crucial Element

With this great complexity, simulation became one of the essential key elements to tackle these challenges. This is since any system under test can be stimulated with artificial sensor data with varying quality to test it [7], [8], [9]. Here, no limitations are given. The system itself can be simulated in different ways, such as in Hardware in the Loop (HIL) as well as in Software in the Loop (SIL) platforms. Also, the sensor input data can be generated with different quality levels and even with specific fault injection to test corner cases. Since the system under test can be separated into its individual components, simulation can be also used to simulate on all levels of the testing pyramid shown in 1.2. One great benefit is that simulation does not require actual vehicles or test drivers. Therefore, it is not safety critical and does not endanger lives of test drivers or other traffic participants. This allows to test situations with potential crashes, which could not be done in reality due to safety or financial impacts. Furthermore, the absence of target hardware and human drivers make simulations about magnitudes of orders faster and cheaper. While in real world tests a vehicle must be brought in certain situations in a controlled environment during work hours of the testing team, simulation can run the same tests with no preparation even faster than real time at any time. The simulation computation costs are in comparison to the target hardware and employee cost negligible. Also, with the only requirement of computing power, dynamic simulation scaling on servers is only restricted by financial limits. While simulation can never replace real world tests fully, the majority of tests is nowadays done with simulation due to its benefits. This importance can be seen in the 2021 Waymo safety report [10], in which they state that they have driven 15+ billion simulated miles versus 20+ million real world autonomously driven miles.

1.3.1 Simulation Approaches

As shown, simulation is an essential element in the development and testing process. Simulation gives here several possibilities to cover the scenario domain space. For the following thesis, they are differentiated into three main categories: Simulation of concrete scenarios, exploitative searches, and exploratory searches.

Simulation of concrete scenarios

The simplest use case is the simulation of concrete scenarios. Hereby a concrete scenario flow is simulated to test the reaction of the system under test. Concrete scenarios play an essential role in homologation, such as in the Euro NCAP [11] and UN ALKS protocols [12]. Here, regulatory authorities provide test sets with certain concrete scenarios to pass as the bare minimum. With this, vehicle manufacturers and suppliers can show and guarantee a certain quality level. On the development side, concrete scenarios are also important. They allow to create new test situations, against which the system can be developed. Once those scenarios are mastered, they can be added to a CI&CD pipeline as regression test. This prevents that already achieved functionality is altered to a defective state within the later development. The sources for such concrete scenarios can be manifold. One main source are the created use case scenarios of the developers and test engineers themselves. They usually create the most important scenarios, as they know the most about the peculiarities of the system. Once the systems are tested in reality, recorded scenarios can be an important source for concrete scenarios as well. These scenarios can be either a replay of trajectories or with a more generic scenario description, which allow further variation. Additionally, the test set can be enriched by scenarios from accident databases or by representative scenario sets generated with traffic studies.

Exploitation of scenario spaces

Concrete scenarios with a fixed scenario flow are essential to test a certain situation. They often also serve as a base scenario for exploitation when it is required to cover more of such scenario instances. This can be done by discretization and variation of scenario parameters. As an example, the UN ALKS protocols [12] dictates the variation for each concrete scenario parameter. In the cut in scenario for example the speed of a challenging vehicle as well as the cut in distance is varied. It is also defined in which bounds the system under test must react in which way. With this approach it is possible to exploit greater areas in the scenario domain space.

Exploration of scenario spaces

While exploitation allows to search the scenario space around known and representative base scenarios, broader search approaches are needed to cover more of the scenario domain space. Exploratory searches need to explore areas outside of known scopes. In best case the search is here in promising areas for critical scenarios within the domain space. Approaches for such exploratory searches can be manifolded. One approach may be combinatorial composition of smaller scenario behaviors to create greater and unique scenarios. Other approaches can be the generation of random traffic around a system under test, which constantly creates new and unique test situations.

1.3.2 State of the Art

With simulation as such an essential element, a large eco system has developed around it. Multiple organizations work on standards to create a common basis among the industry. Furthermore, multiple open source and commercial simulators aim to solve the industry needs. Also, several co-simulation frameworks claim to enhance the connection between simulators and different systems under tests.

Standards

With multiple companies and organizations working on the same problem, there is a desire for a more standardized development across companies. However as often, unique solutions, incompatible approaches, and company interest lead to a fragmented landscape of diverging standards. The Association Standardization of Automation and Measuring Systems (ASAM) [13] alone lists six different, partly incompatible standards for simulation. Furthermore, they develop more than eleven also relevant standards in the field of Test Automation and Data Management & Analysis. Due to this magnitude of standards, the main focus is in the following on scenario description languages. A broad analysis of such scenario description languages can be found in [14]. Subsequent, only the two OpenSCENARIO language versions are introduced since they are the only standardized scenario description languages.

OpenSCENARIO XML [15] is a xml-based standardized scenario file format. It is designed to execute imperative scenarios across multiple simulators. The standard defines a static hierarchical scenario structure with state machines nodes. Also, it provides concrete catalogs with actors and actions. Parameter variation is a minor focus here. Due to its longer existence, the easy to parse xml-based format and its concrete, static scenario structure, the standard is currently widely distributed.

OpenSCENARIO DSL [16] is the successor of OpenSCENARIO XML and aims to evolve the previous version. However, while the same naming indicates consistency between the versions, it is a completely new language with nearly no connections to the previous one. The changes address the following critics and demanding needs: The first critic of the previous version is the xml-based description. While it is easy to parse xml from a developer perspective, the syntactical overhead which comes with xml decreases human readability. For this reason, OpenSCENARIO DSL changed towards a self-defined Domain Specific language (DSL), which improves readability. A second critic point is the static scenario structure, which makes it hard to modify, extend or combine scenarios. Due to this, the static structure is abducted and replaced with a modular approach in which any tree like hierarchy can be build up. Besides the advanced readability and modularity, the new standard version tried to include a large list of new features. Besides concrete scenario execution, the goal is that the standard can also serve to define full test plans with evaluations and domain space coverage. This required a fundamental paradigm change from an imperative towards a more declarative language.

Simulators

Several commercial tool providers provide solutions for intelligent vehicle simulation. In the following, the most important ones in the automotive industry are briefly introduced. Virtual Test Drive (VTD) [17] is a widely distributed and according to their website the world's most widely used simulator platform for intelligent vehicles. It provides a simulator with a visualization engine and is the main driver behind the OpenSCENARIO XML standard. The IPG CarMaker [18] is also a widely distributed simulator. It provides similar features such as VTD, however it is also highly known for its realistic vehicle dynamic and powertrain simulation. Besides these commercial tool providers, there exists also a magnitude of successful open-source projects. The Carla simulator [19] is with more than 2.2 thousand forks and 7.5 thousand stars one the most actively developed open-source simulator. It convinces with an Unreal based visualization and its open structure, as well as with its large community. The openPASS simulator [20] is mainly developed by the German automobile industry to serve as a standardization playground and exchange platform. It allows to generate and run several test suites and provides a simplistic visualization. Esmini [21] is another simulator which is mainly developed by Volvo. It is a lightweight simulator with a basic visualization. Mostly it is known for its OpenSCENARIO XML implementation, which makes it the standard implementation of the OpenSCENARIO XML standard. SUMO (Simulation of Urban MObility) [22] is a multi-modal traffic simulation package, which is intended to be used within other simulators. Its focus is on the generation on realistic traffic simulations.

Co-Simulation Frameworks

Besides simulators and standards, there exists a multitude of co-simulation frameworks, which aim to connect the different simulator solutions with system under tests. The Bosch developed Cloe framework [23] is an abstraction layer between vehicle simulators and system implementations. It allows to exchange simulators since everything is internally mapped to an internal data format. The Eclipse MOSAIC co-simulation framework is intended for connected and automated mobility [24]. Simulators can be connected here with different models and exchange via communication. In the paper [25] another approach is introduced to combine such multiple simulators. Like the Cloe approach, the framework allows to connect different simulators and systems via an internal data representation.

1.3.3 Simulation Specific Challenges

The current simulation landscape suffers from several challenges. The first one is the scattered landscape of standards. Even within the standardized input file formats there are contradictions. For example, one might expect that the OpenSCENARIO XML and OpenSCENARIO DSL versions are consistent, but with the version increment, a contradictory approach was chosen.

While the first version is built as an imperative language, the second one bases on a declarative approach. This magnitude of contradictory standards causes a lack of support by simulation tools. Since each standard requires integration and maintenance effort, simulation tools support either only a selected standard to a limited support or just an own defined input format language. Sharing this development effort collaboratively is difficult since each simulator introduces its own world data representation and architecture. This causes redundant development effort for each simulator solution. The co-simulation frameworks are here also of minor help. They circumvent the mentioned problems by connecting different simulators. This leaves the redundant effort for the simulators itself untouched. Also, the integration costs are shifted towards the integration of the simulators and systems against the framework specific world data format. Either this data format is solution specific, or they rely again on one of the many standards. As the last point, most of the solutions are still developed with a mental mind set of testing on a system and acceptance test level. Nearly all simulators provide many features but are too monolithic and slow to be used in deeper test levels such as in component or unit test levels. While this traditional testing approach is still important, the other levels are essential for modern software development.

1.4 Problem Statement

As it was shown, the field of intelligent vehicle development and testing is complex. A nearly infinite domain space needs to be covered on a reoccurring basis with reasonable effort. Simulation solutions are essential to tackle these problems. However, there is on the one side a wide range of simulators with a growing number of open and proprietary input standards on the other side, which are also getting more complex. This puts simulation tool providers in the unfavorable position to provide more and more combinations with increasing effort. Therefore, a solution is needed, which helps to solve the rising complexity of intelligent vehicle simulation and the practical problems within the simulation landscape. In the following this solution space is narrowed down systematically with the help of the arc42 architecture template [26]. First, the overall goal is defined, and its requirements are derived. Subsequent, quality goals to guide the implementations are determined. Last, constraints, restricting the solution space, are gathered.

1.4.1 Goal and Requirements

A framework is needed, which helps to solve the rising complexity of intelligent vehicle development and the practical challenges within the simulation landscape. Such a framework must enable minimizing redundant work by developing input file format compilers collaboratively for multiple target simulators. Also, it should be possible to use one or more simulators with different sub scopes and combinations as co-simulation if needed. To be holistic, the framework must be capable enough to represent all necessary intelligent vehicle simulation use cases. Furthermore, it must be usable in an incremental way so it can be used on each test level. Based on this goal, the following requirements are derived:

Req. 1: Agnostic development of input format compilers and simulator bindings

The redundant development of input format compilers for different simulators must be reduced. This concerns for example scenario formats, simulation and system configuration formats, and map formats. While all of them are important, the focus of this thesis will be on scenario formats. The solution must provide a way in which the integration of a new input scenario format can be split into two independent sub tasks: The development of an input format compiler, which processes all format specific tasks which are independent of the underlying simulator. And the development of simulator specific bindings, which allow to connect this with a simulator specific implementation.

Req. 2: Incremental simulation for all test levels

Since all test levels of the testing pyramid are important, the framework must be built up in such a way that it can be used for all levels. It must be possible to test the full intelligent vehicle stack with simulation, but also down scaled simulations are needed to test for example the perception unit in isolation. Such a holistic framework must allow to build up highly customized simulation solutions to match exactly the complexity of each individual test. For this it is necessary to be able to produce different simulation output quality, either by providing different simulation scopes or the usage of different simulators, even as co-simulation. In this thesis a simulator scope refers to a smaller but more lightweight subset of the full simulator set. Furthermore, the whole framework basis must be designed in such a way, that it is inherent to be used in unit tests. This concludes for example the avoidance of network or inter-process communication.

Req. 3: Ability to perform concrete, exploitative and exploratory searches

A holistic framework must be able to represent any kind of intelligent vehicle simulation use case. This includes the simulation of concrete scenarios for regression tests and basic development. Furthermore, it must enable more advanced domain space search approaches for homologation. This includes the exploitation of local scenarios spaces and exploration of large scenario areas in the domain space.

1.4.2 Quality Goals

To evaluate a software product quality, the ISO/IEC 25010 [27] introduces eight characteristics. They are Functional Suitability, Performance efficiency, Compatibility, Usability, Reliability, Security, Maintainability and Portability. Not all of these characteristics are relevant for every project. The arc42 reference guide [26] advises to choose the top three relevant characteristics for a project. Since the goal is to provide an easy-to-use framework to minimize development and maintenance effort, the following three quality goals are chosen for this project: Functional Suitability, Usability, and Maintainability. The concrete definitions of these characteristics are given in the appendix 7.1;

1.4.3 Technical and Organizational Constraints

While the requirements and the quality goals define the solution space, it is restricted further by technical and organizational constraints. In the field of intelligent vehicle simulation, the following constraints exist:

Constraint 1: Industry Adequate C++ Code

In the industry there are constraints for the code itself. On the one hand, it must be performant enough for real-time capable applications. On the other hand, it must support safety argumentation required by standards such as the ISO26262 [28]. The current language of choice in the industry for production code is hereby C++. The ISO driven standardization also guarantees long term API stability to a certain degree. This performant language allows to write real-time capable applications and the static type system also adds an extra layer of reliability. Furthermore, a rich eco system with libraries and commercial support such as for static code analysis allows to write better and safer code. While this could be achieved also with other languages, the interaction with other, mainly C++ based code is often an extra hurdle.

Therefore, C++ shall be used as the underlying language for the current work. Furthermore, the code must be of an industry adequate level, ensured by extensive unit testing, static code analysis and usage of style guides.

Constraint 2: Simplicity

As already pointed out, one overall goal is to reduce integration and maintenance costs. Therefore, only well-known designs and solutions shall be used so developers have a low barrier to entry when interacting with the framework. Furthermore, the usage of simulation standards and own developed world data representations shall be avoided. This is due the already discussed effect, that the simulation landscape is yet in a scattered state and no commonly accepted standard exist yet. Users should not be forced to use a specific standard, but they should always build in such standards on top.

1.5 The Yase Contribution

The field of intelligent vehicle simulation is fairly complex and even with the fact that a multitude of simulation solutions exists, the author is not aware of a solution which solves the specified goal above in a sufficient way. For this reason, the so called Yase (Yet Agnostic Scenario Engine/ Yet Another Scenario Engine) framework is presented in this thesis. It is an input format and simulator agnostic scenario engine. The three-stage approach allows to design and develop scenario format compilers (front ends) and simulator adapters (back ends) independent of each other. Internally it bases on Behavior Trees (BT) to link everything together. With its modular approach it is also applicable for holistic and incremental simulation of all testing levels. This work contributes to the state-of-the-art research regarding intelligent vehicle simulation for testing and validation in multiple ways. The specific contributions can be summarized as follows:

- **Development of a holistic and incremental simulation framework:** For the derived problem space, the concept a three-stage framework approach with a middle end, front ends, and back ends is developed. It is capable to perform holistic simulation while being incremental and agnostic.

- **Analysis of control architectures:** For the scenario representation independent of input file formats and simulators, the most important control architectures are elaborated. Furthermore, they are evaluated with the help of ISO quality goals.

- **Introduction to behavior trees:** The general concept of behavior trees is presented. In this context the most canonical implementation strategies, graphical representation, and terminology is introduced. Furthermore, the behavior tree characteristics are compared to the other control architectures.

- **Agnostic Behavior Tree (ABT) implementation:** Next, the Yase behavior tree implementation is introduced. This implementation is specifically designed for the intelligent vehicle simulation domain. The implementation serves as the basis for the further work and is also released open-source.

- **Demonstration of simulator back end implementation:** Within this context the implementation of a simulator back end based on the agnostic behavior tree is demonstrated. Hereby the implementation of behavior nodes is discussed, and a representative scenario is built up. The scenario is hereby used to exploit a two-dimensional scenario space. Furthermore, it is discussed how driver and traffic models can be build up with the same behaviors. With this as basis, a massive traffic simulation is simulated to demonstrate an exploratory search. Despite scenario simulation, the behavior tree can be used as a full simulation engine. It is shown that behavior trees are able to fulfill all needs of simulators such as scheduling, ensuring data consistency and more.

- **Demonstration of a compiler frontend implementation:** To demonstrate a Yase front end implementation, the implementation of such is demonstrated exemplarily for the OpenSCENARIO DSL language. Here, the general process from translating an input file to an executable behavior tree is pointed out. Furthermore, OpenSCENARIO DSL specific features are discussed, and implementation strategies are shown.

- **Open sourced implementation:** Next to theoretical concepts, the most important contribution of this thesis is the open-sourced implementation. This allows to collaborate among different companies. As a sub project under the openPASS repository, it is a well-established project with an active community [29].

Besides the noted contributions, the Yase concept already serves as the basis for more developments. Especially to mention is the Agnostic Type System (ATS), which is next to the Agnostic Behavior Tree (ABT) a central element in the Yase middle end. Furthermore, open sourced and company intern front ends and back ends are developed on this basis. However, these are not part of this thesis.

1.5.1 Document Outline

The following thesis is structured into five chapters. First, the overall Yase architecture with its separation into three logical stages is presented. It is shown how the concept is derived from the successful design pattern of computer language compilers. This results into a separation of the problem into an intermediate scenario representation as a middle end, scenario language specific front ends and simulator specific back ends. Secondly, the intermediate representation of scenarios as the underlying interface of the other stages is elaborated. Hereby potential approaches are discussed and compared against each other. For the subsequent work the concept of behavior trees is used as the intermediate representation of scenarios. As behavior trees are not a strictly defined concept, several implementation strategies and open-source projects are summarized and evaluated. Based on this, the Yase specific implementation is derived. In the fourth chapter it is shown how the Yase middle end with its behavior tree can be used to implement a simulation specific back end. First, several exemplary behavior node implementations are introduced, which serve as a basis for the following more complex behaviors. Hereby it is also discussed how the approach can be used to model the whole simulation as behavior tree. With so called MockupSimulator, the implementation of concrete scenarios is demonstrated exemplary with an Euro NCAP scenario. Secondly, it is shown how reactive behaviors such as driver and traffic models can be designed out of the same behavior nodes. The presented models are then evaluated regarding their capabilities of exploratory traffic simulation. Last, the concrete scenario is used for an exploitative search to find critical scenarios. In the next chapter, the stage of scenario language specific front ends is elaborated. Hereby it is presented for the OpenSCENARIO DSL standard, how a front end can compile such a scenario with the help of an Extended Backus–Naur form (EBNF) grammar and transpilers into an executable behavior tree. The last chapter recapitulates the outcomes of this thesis. Here, is summarized how the concept meets the defined requirements and quality goals. Finally, the current state of the Yase project and its open-source community is presented.

Chapter 2

Yase Architecture Design

In this chapter the overall architecture design of the Yase (Acronym for Yet Agnostic Scenario Engine / Yet Another Scenario Engine) framework is presented. First, the main design driver is identified, and the problem is compared to the design of programming language compilers. It is shown how the typical architecture of programming language compilers is built to fulfill the same requirement. Based on this, a three-stage architecture for the Yase framework is presented. These three stages separation also serves as the layout for the following thesis.

2.1 Main Design Driver

The goal of the Yase framework is to enable holistic simulation-based development and testing of intelligent vehicle systems in complex and changing software and tool environments. This requires all deduced requirements of section 1.4.1 to be fulfilled. However, some of these requirements are more influential on the overall architecture design then others. In the current case, the first requirement 1.4.1 stating agnosticism regarding input file formats and simulators and sub scopes is the most important one. Simulation in large industry and research cooperations is usually quite incompatible among the individual projects. This is caused by project specific needs, which often lead to the usage of different simulators, tools, and processes. Since the industry is lacking generally accepted scenario or simulation input file standards, many simulators and tools use their own propriety formats. This causes tremendous efforts when sharing and cooperating among projects. While in general combinations of multiple simulators (co-simulation) and sharing data with other simulation solutions is desired and encouraged, the benefits are often less than the effort to integrate the different simulation solutions. This is since for each simulation solution an adapter is necessary to read in a different format. The number of required adapters for complete compatibility among all input formats and tools rises in the worst case quadratic with the number of tool and input formats.

$$adapters_{max} = formats \times tools \tag{2.1}$$

Furthermore, writing adapters is usually not trivial and expensive. The initial high effort cultivates ongoing work in parallel and self-contained simulation solutions with little reuse for other projects, sometimes even in one common project. As another side effect, it increases lock-in effects in various degrees on tool vendors, since already existing data bases can often just be interpreted by one specific tool. The Yase concept aims to avoid this by enabling the development of parsers for input files independent and reusable for different simulators. On the one hand it should be possible to develop input file parsers agnostic of the underling simulator. On the other hand, simulators should be usable agnostic of the used input file format. This is the basis for joint development of shared and reusable tools.

A similar problem exists in the field of computer language compilers. Compiler programs consume files of a given high level programming language, mostly human readable languages. These are then translated into lower-level languages which usually is executed on a certain target hardware. Contrary to interpreters, which translate and execute programming languages line by line, compilers first parse the full file into an intermediate representation. This intermediate representation allows to realize performance optimizations and safety checks before translating the code further. Last, compilers translate this intermediate representation into executable instructions for the given target hardware. Compilers are also designed from the perspective that there exists a multitude of programming languages on the one side and multiple target hardware architectures on the other side. Most compilers alleviate the problem by using the intermediate representation as an abstraction layer between the input formats and the output format. This allows to separate the problem into tools which read in input files into the abstract intermediate representation and tools which translates this into a set of instructions for the target hardware.

2.2 Three-stage Compiler Pattern

Compilers are a well-studied research field in computer science and proven in praxis [30]. For these compilers a common design pattern evolved over the years with a clear separation of concerns. The compiler architecture is hereby separated into the three stages of front ends, a middle end and back ends. The most important stage is the middle end with the intermediate representation. It is a low-level representation of the given problem domain and serves as an abstraction layer. Front ends for specific programming languages parse and translate files into the intermediate representation of the middle end. This intermediate representation then serves as the interface for a back end. It takes the intermediate representation and translates it into the necessary instructions for a given target platform. In the Fig. 2.1 the LLVM (Low Level Virtual Machine) compiler [31] is illustrated as an example for the LLVM is together with the GCC (GNU Compiler Collection) [32] the most established compiler and has a special focus on modularity and extensibility. For the LLVM compiler there exist a magnitude of front ends and back ends. As an example, the C++ Clang frontend [33] parses and translates C++ files into the LLVM intermediate representation. From there on it can be directly executed on different target hardware such as ARM or X86 via back ends. The modular ith an intermediate representation as abstraction layer scales well with more combinations. The problem of parsing a specific input file is decoupled from the translation to a certain target hardware.

$$adapters_{max} = formats + target \tag{2.2}$$

While the direct translation has a quadratic complexity rise as shown in equation 2.1, the intermediate representation reduces it to a linear complexity rise for each new format or target architecture as shown in equation 2.2. As an example, the developers of the relatively new computer language Rust avoided to write their own compiler, but developed a Rust front end for the existing LLVM compiler instead [34]. With this front end they were directly able to compile Rust code on all target hardware which are already supported by LLVM.

Middle end: Abstraction layer for domain problem

The basis of a compiler middle end is an abstraction layer, which is capable to represent the full domain space as an intermediate representation without loss of information. Often, the code is represented as an Abstract Syntax Tree (AST). It represents the essential syntax of the code in a tree structure. Irrelevant syntax such as comments, empty lines can be pruned before. Furthermore, programming specific syntax is translated in the abstract format. Based on this abstract representation, it is possible to reason about further performance optimization independent of any given input file format. Typically, compilers eliminate instructions, which do not have any effect. This may be calculations of unused variables or removal of code, which can never

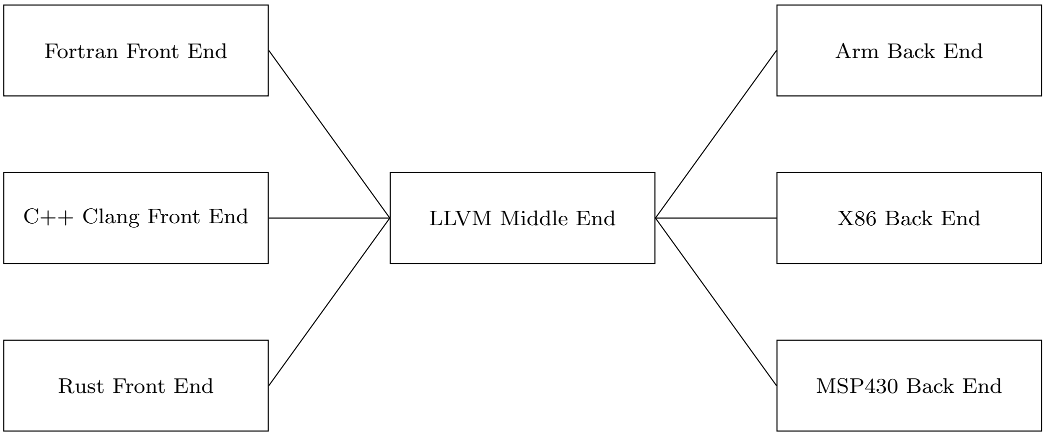


Figure 2.1: The modular three-stage LLVM compiler concept.

be reached. Such optimization is done under the strict “as if” rule. It states that any optimization is allowed under the constraint that the result is the same as if it would have been without optimization. Furthermore, middle ends can perform safety checks such as type checks and finding memory leaks.

Front end: Lexing and parsing

Compiler front ends are responsible to translate files of a certain input format into the AST structure of the middle end. These front ends are often unique since the language formats can differ strongly. Hereby the given language format is defined with the help of a grammar such as Backus-Naur-Form (BNF) or the Extended-Backus-Naur-Form (EBNF) [35]. The process of file parsing can be divided into the lexical analysis and the syntactical analysis. The lexical analysis is done with Lexers, which are sometimes also called Tokenizers. They split up the input file into a sequence of tokens. The tokens are keywords defined in the grammar and it allows to check for ambiguities or undefined tokens. Once the input file is translated into a valid stream of tokens, it serves as the input for the parser. Parsers perform syntactical analysis on the given input token stream. Hereby the relative occurrence of the tokens is checked against the grammar and transformed into the abstract syntax tree. This AST can then be processed further in the middle end by traversing the tree with the help of visitors. Since front ends are needed for many use cases, there exist multiple transpilers (also sometimes called compiler-compiler) to support the development of such.

Back end: Platform specific execution

In the last compiler stage, the back end translates the processed and optimized intermediate representation into processor specific instructions. Since processor architectures differ, for each processor architecture an individual back end is required. Besides just translating, the back ends can also perform further optimizations, as it has knowledge of the specific hardware architecture. Compilers usually support a variety of target hardware. As an example, the LLVM compiler can support many processor architectures such as X86, ARM or PowerPC.

2.3 The Yase Three Stage Concept

Programming language compilers solve a comparable problem as scenario engines. In both cases one or more input file formats must be parsed. For compilers it is computer language files and for scenario engines scenario description files. This is then translated into a logical sequence of instructions for a given target. Here, compilers produce sequences of hardware instructions for a certain hardware setup. Scenario engines on the other hand produce a sequence of action instructions for one or more given simulators. The only difference is the fact that the high-level language input format is translated to a sequence of simulation instructions instead of hardware instructions. As the established language compiler design scales well for supporting multiple input formats and targets, it is a promising architecture for Yase. Therefore, Yase is consequently build up as a three-stage approach with a middle end, front ends and back ends. In Fig. 2.2 the concept is presented which demonstrates how a back end and front end can be developed agnostic of each other. In a final project, the relevant back ends and front ends are imported and combined to a project specific scenario engine as shown In Fig. 2.3 . The three stages serve also as the layout of thesis. Hereby each stage with the specific design decisions is elaborated in individual chapters.

Middle end: Intermediate representation of scenarios

The most important stage of the design is the middle end with its intermediate representation. This serves as the abstraction layer between any input format specific front enddd and the simulator specific back ends. Therefore, the Yase middle end with its intermediate representation must be expressive enough to represent the full domain space of simulation. The most important content to express are simulation behaviors of multiple kinds: Scenario

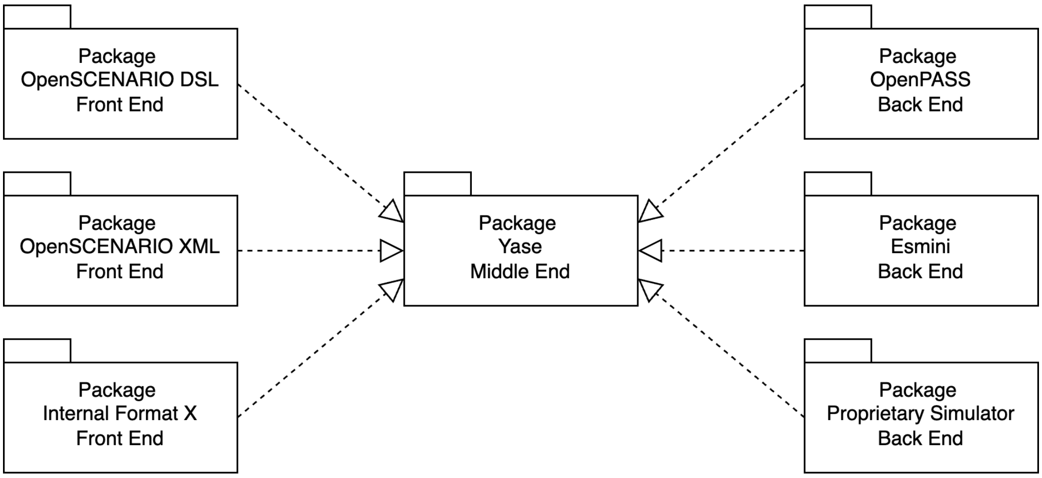


Figure 2.2: UML diagram of the modular three stage Yase concept.

behaviors, simulation runtime behaviors and sensor model behaviors. Therefore, the middle end must provide a behavior representation, which allows to model and combine such behaviors in a generic way. Some input formats instantiate these behaviors with additional static typed input arguments, therefore there must be possibilities to express behavior definitions in a type safe way. This can be done with the *Agnostic Type System* package which contains a static type system with type safe function expressions, arithmetic expressions, user defined type definitions, casts and conversions. This simulator and scenario file agnostic convenience tool is also included in the middle end. While the type system is also an integral part of Yase, the focus of this thesis will be solely on the representation of the dynamic behaviors.

Back end: Platform Specific Execution

A back end in the Yase context is any set of supported behavior primitives, which should be made available for front ends. These primitives must be modeled in the way as defined by the middle end. The scope of a back end is hereby unconstrained. It can be either the full set of all possible behavior instructions for a given simulator. Alternatively, this full set is split into multiple, self-consistent back ends for simulator sub scopes. This allows to slice the content of a simulator into different system cuts and provide only what is needed for a specific simulation. Furthermore, multiple simulators and simulator system cuts can be combined later on.

Front end: Input File Translation

Yase front ends provide the mechanisms to translate a specific input format into a consistent and executable simulation in the way defined by the middle end. This is highly input format dependent as they differ in semantics and syntax. As an example, the OpenSCENARIO XML standard defines a static scenario structure in a XML format, while the OpenSCENARIO DSL standard uses an own Domain Specific Language (DSL). The front end compiles everything which is independent of any simulator. At places in which an implementation specific behavior must be placed in, it uses a key value storage provided by the Agnostic Type System to look up and place actions. With this generic key value storage, the front end does not need prior knowledge of the supported content and can be developed independent of it. The key value storage is later filled by all used back ends, which introduce their scopes via dependency injection. While the focus of this thesis is on scenario execution, the same approach can be used to parse configuration files in the same way. This can include for example sensor setups with different sensor model behaviors.

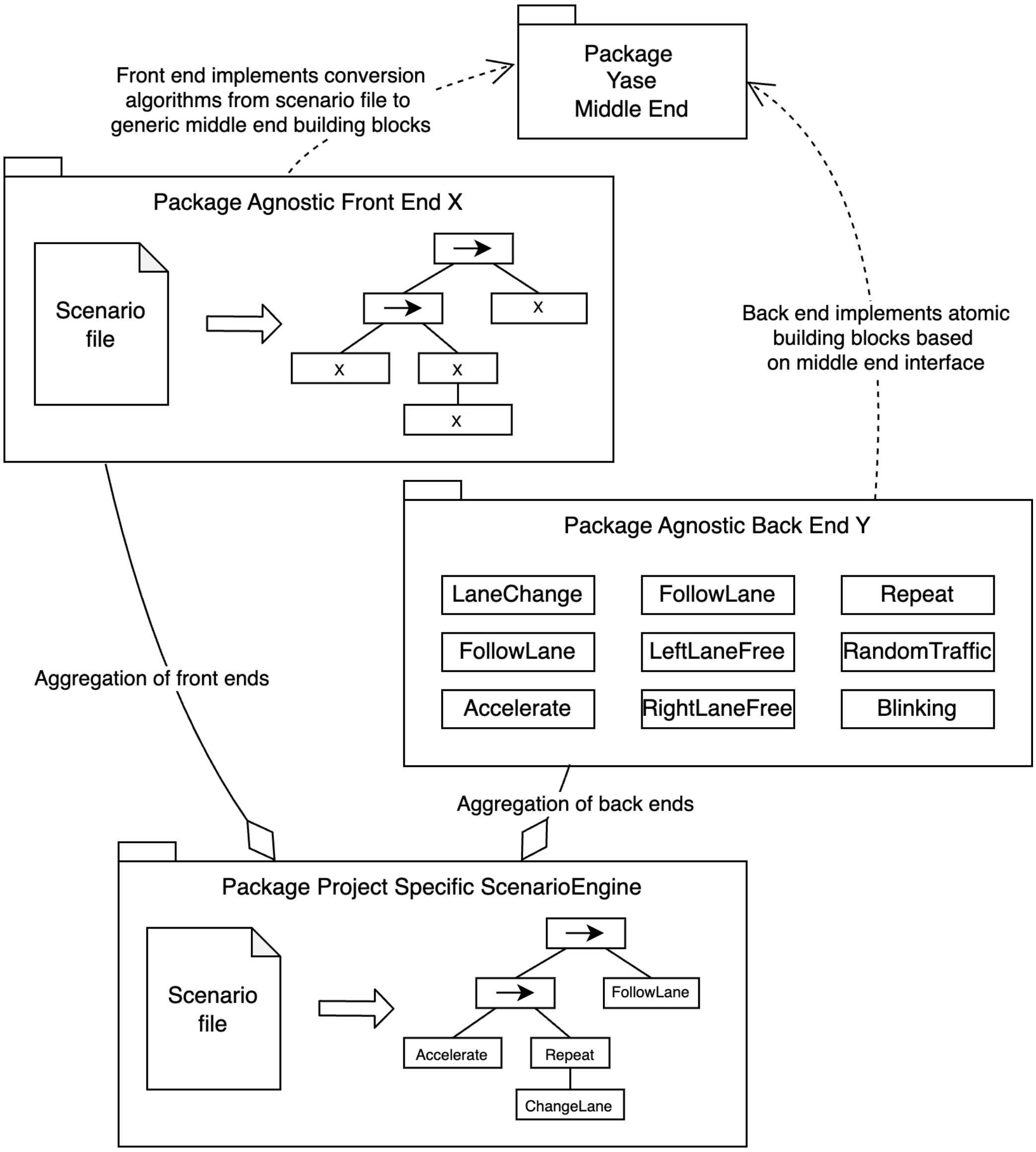


Figure 2.3: UML diagram of the Yase concept in usage in a final product.

Chapter 3

Middle End: Scenario Representation

The central element of the Yase concept is the middle end stage as the interface between front ends and middle ends. For this purpose, a powerful intermediate representation is needed, which is capable enough to represent all types of scenarios. While there are several aspects of such a scenario representation, the most important one is the representation of the dynamic content of a scenario. In this section the design of the middle end is elaborated. First, the most important control architectures are examined, which could serve as such an intermediate scenario representation. Subsequently, it is shown why and how behavior trees fit best for the task. After the comparison, the behavior tree concept is investigated further. Hereby the most common implementation strategies and open-source projects are presented and compared. Based on this, the agnostic behavior tree architecture to match the domain specific use case of Yase is presented. Hereby the terminology and design are compared with other implementations.

3.1 Survey of Control Architectures

Control architectures are mental models to structure actions in a meaningful way that they can express a behavior over time dependent on given input. Such models are elementary in computer science and robotics. The most common ones are hereby Finite State Machines (FSMs) and Hierarchical Finite State Machines (H-FSM). Also, Decision Tree (DT) similar approaches can help to model behaviors. Besides these well studied models, manifold lesser-known architectures exist. Examples in the AD/ADAS industry are scenario engines closely implemented after the OpenSCENARIO XML scenario structure. A further, relatively new concept from the gaming industry is Behavior Trees.

3.1.1 Finite State Machines

A Finite State Machine (FSM) or Finite State Automaton (FSA) is an elementary conceptual model in computer science to describe transitions between states [36], [37]. FSMs consist of a set of possible states with conditions for transitions between these states. A Finite State Machine can be seen as a graph, where the vertices represent the states and directed edges represent the transitions with their conditions. At any given time, FSMs can be only in exactly one state of the finite set of possible states. With a given initial state and all given inputs, the FSM and its trace is fully defined. As one of the first established models, automata theory became an essential part of the computer science. Due to its simplicity, this model is used in practical applications in many fields. However, while FSMs are easy to understand and implement for simple systems with low complexity, they often get overly complicated for larger systems. Adding, removing, relocating, or modifying states can potentially affect any other states within the graph. This is caused by the nature of the graph structure, where each state can potentially have a transition to any other state including itself. For a given FSM with N states, the number of potential transitions $transitions_{max}$ rise quadratic with the number of possible states:

$$transitions_{max} = N \times N \tag{3.1}$$

The potential quadratic rise of transitions worsens the ability to add or remove a new state since potentially all existing nodes may have to be revisited due to transition dependencies. An additional problem is that FSMs often lead in implementations to a tight coupling of states and the transitions with their trigger conditions. The transition to the next state is often integrated within the state itself, which is comparable to the problematic “goTo” expression. This mixture makes it hard to implement generic behaviors independent of the circumstances they are used in.

3.1.2 Hierarchical Finite State Machines

Since the complexity becomes more difficult to handle with growing number of states, the classical FSM concept was improved over time towards hierarchical FSMs (H-FSM) [38], [39], [40]. With this extension, self-contained parts of a FSM are packed into greater states, often called “SuperStates”.

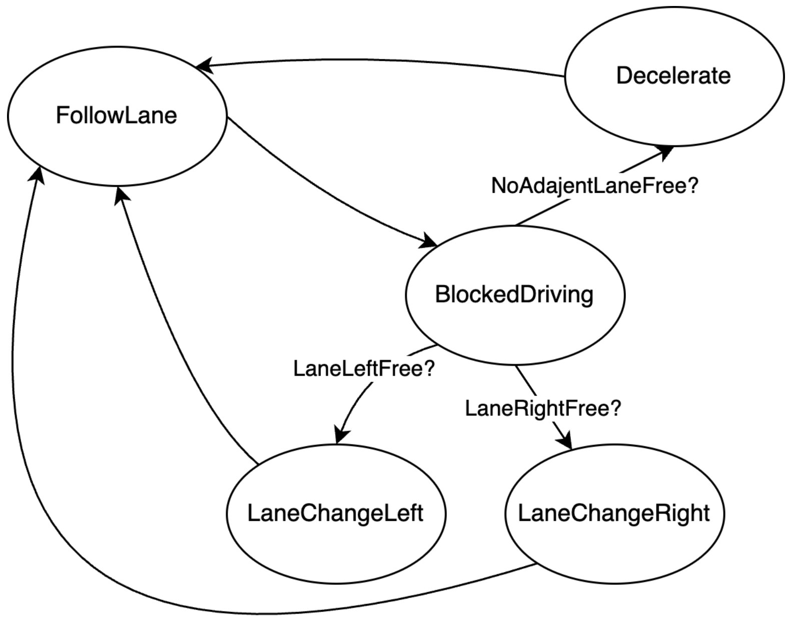


Figure 3.1: Example of a Finite State Machine.

Each “SuperStates” provides unique entry and exit points for the underling FSM and therefore allows to model more abstract FSMs with defined interfaces. Since these “SuperStates” can be nested again in arbitrary depth, the H-FSM exhibit a hierarchy which gives the extension its name. The hierarchical grouping of independent parts reduces complexity and eases the understanding by grouping sub behaviors. But it still does not solve the problem fully. At each hierarchy level the complexity can still have $N \times N$ complexity, whereas N is the number of states at the specific level.

3.1.3 Decision-Tree Related Architectures

FSMs and their extensions show good capabilities on processing procedural tasks, however often reactive systems are needed which can perform actions based on certain condition changes in the environment. To solve this problem, several approaches have been developed. In this thesis, they are summarized under the term “Decision-Tree Related Architectures” [41], but they are also sometimes refereed as Event-driven architecture (EDA), Trigger Engines or Event Engines. All these architectures perform actions as a reaction on certain events. The concrete implementation can hereby differ. Some implementations re-evaluate the events in every step, while others evaluate them only once. Some implementations check for all events equally in paral-

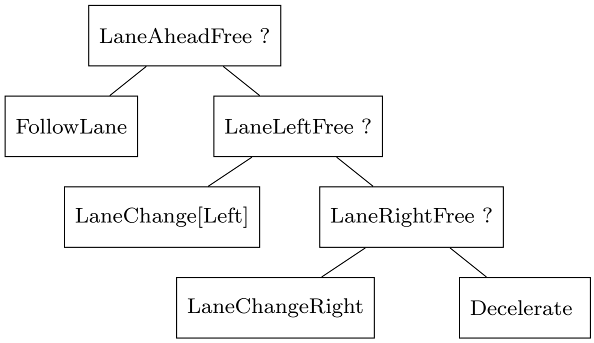


Figure 3.2: Simplified driver model modeled as decision tree.

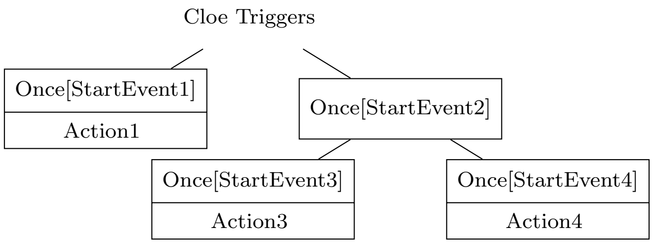


Figure 3.3: Example of two independent, partly nested Cloe Triggers.

lel, while others may have some sort of priority order. Some implementations even allow deeper nesting of events.

In Fig. 3.2 an exemplary decision tree implementation of a simplified driver model is presented. This decision tree can be evaluated in every step to enable reactive behavior. The conditions, such as LaneAheadFree?, are evaluated to choose either the left or right branch. Based on the current environment state, the decision tree can be traversed in every step to find the adequate driver model action. Similar solutions are implemented in many simulators and simulator frameworks to solve various use cases. As an example, the Cloe simulation framework provides a Trigger Engine [42], which allows to perform actions and evaluations within the simulations based on events. In the specific Cloe case, all triggers are evaluated in parallel until each trigger is evaluated once. Furthermore, the Cloe framework allows to combine multiple triggers into greater triggers. An exemplary setup can be seen in Figure 3.3. While such decision tree related approaches show good reactiveness, they have disadvantages. Procedural tasks such as sequences are not intended in the concept and require additional internal state variables to model it. Also, the execution of parallel tasks can be modeled only as separate and independent trees. This complicates the generation of large, partly reactive, partly parallel, and partly procedural scenarios.

3.1.4 OpenSCENARIO XML Architecture

Within the intelligent vehicle simulation community, first standardization efforts regarding scenario description and execution were taken with the OpenSCENARIO XML standard [15], [43],. The result is a mixture of a static hierarchical scenario structure with finite state machines at certain levels. In Figure 3.4 a simplified version of the scenario structure is visualized. It consists of the following levels:

- A Scenario contains one InitPhase and one Storyboard for the dynamic content.
- An InitPhase contains zero or more parallel InitActions.
- A Storyboard contains one StopTrigger, which runs zero or more Stories in parallel.
- A Story contains one or more Acts, which are executed in parallel.
- An Act contains one mandatory StartTrigger and an optional StopTrigger. It executes zero or more ManeuverGroups in parallel.
- A ManeuverGroup contains zero or more Actors references. For these Actors zero or more Maneuvers can be defined, which run in parallel.
- A Maneuver contains one or more Events, which run in parallel based on a FSM and EventPolicies.
- An Event consists of a StartTrigger and one or more Actions, which run in parallel.

While the structure is static and strictly hierarchical, some levels contain FSM logic. As an example, actors are represented as FSMs and also the selection of Events in the Maneuver level are chosen via a FSM logic and EventPolicies in a decision tree similar style. This standard has been influential for the execution model of many simulator implementations. As an example, in the simulator esmini [44] and the deprecated openPASS scenario engine implementation [45] this static scenario structure is implemented. Each level of the scenario tree is resembled with an equivalent C++ class, which contains the logic as defined in the standard.

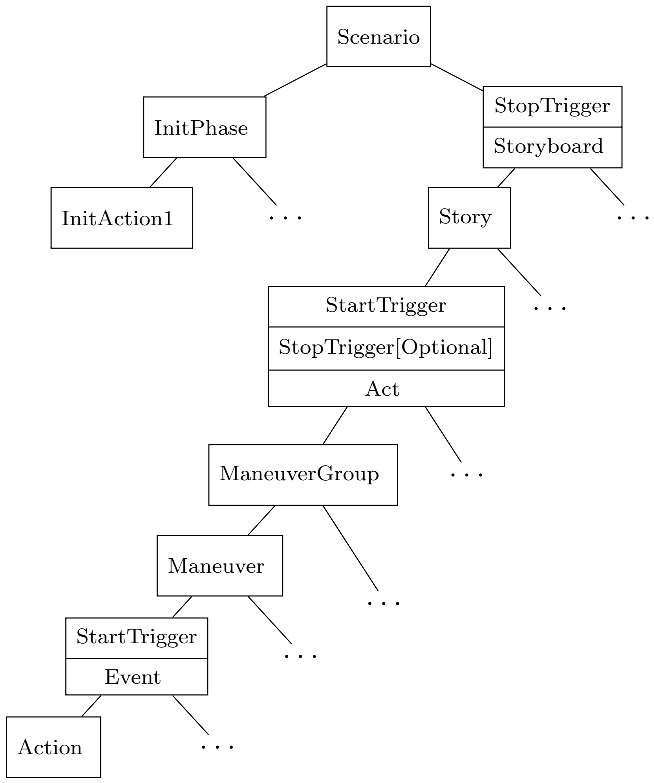


Figure 3.4: Simplified scenario structure of OpenSCENARIO XML.

However, the description and implementation of scenarios with the static OpenSCENARIO XML scenario structure has drawbacks. First, the static scenario structure obtains a high complexity for even simple scenarios. Even if only one action should be executed in a scenario, the whole scenario tree shown in Figure 3.4 has to be build up. This adds up to twelve partly superfluous hierarchy levels, which is also difficult to comprehend for users. Secondly, the static scenario structure impedes modularization and abstraction. The scenario structure can only grow in width, while the depth of the scenario is defined by the standard. Also, elementary operations such as the StopTrigger and StartTrigger are restricted to certain levels. These tight rules make separation of abstract sub scenarios difficult. Third, the execution model lacks elementary operators to combine and extend behaviors. Since all levels are intended to run in parallel, elementary composition operations such as sequences of actions can only be modeled with Start/Stop-Trigger and synchronized state variables. These synchronized state variables build up dependencies between different scenario parts and make the overall scenario difficult to maintain and extend.

3.1.5 Behavior Trees

In the recent past, modular, maintainable, and reusable development of Non-Player Character (NPC) in gaming became more important. Implementations with complex FSMs and H-FSMs have shown to be difficult to develop and maintain with the rising complexity. To overcome these problems, the concept of behavior trees was first developed in 2005 for the game “Halo” [46]. Since then behavior trees became the standard in gaming engines and robotics [47], [48], [49], [50]. As the concept is relatively new, there is no canonical design pattern yet and manifold implementation varieties exist. However, all behavior trees consist of three basic concepts, which are canonical among all major implementations. The representation of behaviors in an object orientated way as nodes, the stepwise execution and the general node primitive’s classification, which allow to compose trees of behaviors.

Behavior Nodes and Execution States

The general idea behind behavior trees is to encapsulate a task or any other arbitrary behavior as a self-contained behavior node class. The sole purpose of such a node is to express its specific behavior. As an example, a lane change of a vehicle in the simulation context can be modeled with a *ChangeLane* behavior node. To trigger the execution of one small step of the behavior (also refereed as to execute or tick a behavior), behavior nodes provide an API method, usually named *executeTick()*. Most intelligent vehicle simulators proceeded the simulation time with discrete time steps. While some behavior tasks can be executed immediately in full extent, others need to be stretched over a certain duration. For example the *ChangeLane* behavior needs multiple ticks to perform the full behavior, while a *ButtonPress* behavior could potentially finish within one step. To be able to track the execution process from the outside independent of the underlying behavior, the *executeTick()* method returns an execution state. All behavior tree implementations consist of the basic execution states *kRunning*, *kSuccess* and *kFailure*. A behavior which is executed, but is not finished yet, possesses the state *kRunning*. This state indicates that the behavior needs to be re-executed at least one more time in order to finish the intended behavior. Once a behavior is executed fully in the desired way, the status changes to *kSuccess*. Contrary, if it was not possible to express the intended behavior, a *kFailure* status is returned. These execution states are inherent to the behavior tree concept, as it allows to observe the current execution state from the outside and compose behavior nodes agnostic of their implementation in upstream logic.

In Fig. 3.5 an example for the temporal execution is given for the *ChangeLane* behavior. The simulation step width is set in the example with a time delta $\Delta t = 1s$. With the first tick, the behavior node starts expressing the intended behavior in the simulation environment by moving the vehicle towards the adjacent lane. As the lane change requires time, the node preforms only a small part of the behavior at each simulation step. During this process, the behavior node indicates the yet unfinished behavior progress with the execution status Running. This information is used by the simulator to re-execute the behavior in the subsequent steps. Once the lane change is successfully performed, the returned execution state changes to *kSuccess*. As the *ChangeLane* is the sole behavior to execute, the simulator can stop the simulation at this point and report the positive feedback of a successful scenario execution. In an alternative simulation situation with the circumstances of no adjacent lane, the *ChangeLane* behavior would return a *kFailure* to indicate that the desired behavior cannot be expressed. In this case the simulator can decide how to handle the execution state *kFailure*, for example with a simulation abort.

Behavior Node Primitves

To be able to treat all behavior nodes the same, they are all derived from one generic behavior node interface as shown in the UML diagram in Fig. 3.6. Hereby, this abstract behavior node is in most implementations derived fur-

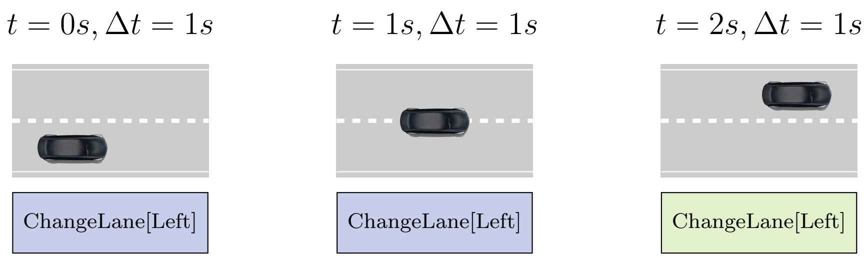


Figure 3.5: Temporal execution of a *ChangeLane* behavior with execution states *kRunning* (blue) and *kSuccess* (green).

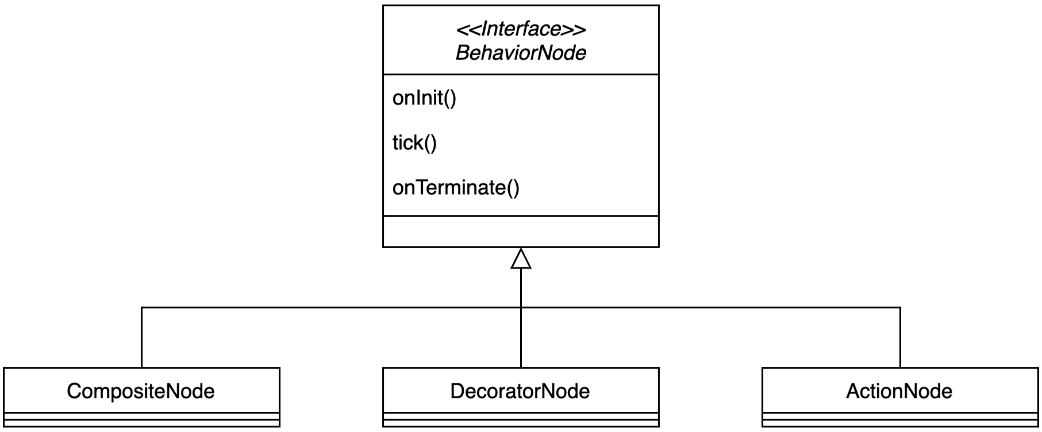
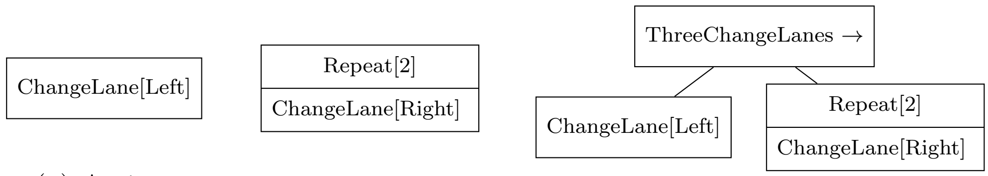


Figure 3.6: UML diagram of a simplistic behavior node implementation and its primitives.

ther into three primitives: *Actions*, *Decorators* and *Composites*. Behaviors, performing atomic tasks, are called *Actions* and are represented as rectangles (see Fig. 3.7a). Actions perform usually atomic tasks, for which it does not make sense to split the behavior further and therefore it can be fully expressed with one node. An example is the previously described *ChangeLane* behavior. But often such behaviors need to be used in slightly different contexts or it is necessary to extend the underlying behavior partly to a given situation. This can be done with *Decorator* nodes, which allow to extend the logic of other behaviors without modifying the underlying behavior itself. The *ChangeLane* action for example may be used in a driver model and here it should perform the lane change only if the adjacent lane is free. Or it may be used to create a critical situation, in which the lane change must be performed when a reference vehicle is approaching on the lane to change to. A third example is when the lane change should be performed twice over two lanes. Instead of creating specialized actions for each use case, decorator behaviors allow to extend the underlying behavior. Decorators can perform logic before and after the execution of the decorated child behavior and allow to modify the returned execution state of the child. Due to that reason, decorators can influence if, when, how long, under which circumstances and how many times an underlying behavior node is executed. As decorators can only decorate one child behavior, they are represented as a rectangle on top of the behavior to decorate as displayed in Fig. 3.7b. In the example the *ChangeLane* node is decorated with a *Repeat* decorator, which repeats the underlying behavior two times. With this approach the double lane change can be modeled together with a *Repeat* decorator instead with a partly redundant action *DoubleChangeLane*. Furthermore, the generic *Repeat* decorator can now also be reused to decorate any other behavior. Contrary to *Decorator* nodes, the *Composite* nodes (or lesser known as *Flow-control nodes*) allow to combine more than one behavior to a greater behavior. Here, any kind of composition of underling behaviors can be implemented. However, among most implementations three common composites are dominant as they cover the most elementary composition operations: The *Sequence* composite allows to execute multiple child behaviors in a row. Here, the first child is executed until it returns *kSuccess*, which causes the next child to be executed from the next tick on. The typical execution order is from the left child to the right until all children are executed successfully. If one of the children fails during the process, the sequence aborts also with a *kFailure*. In Figure 3.7c such a sequence is presented with the commonly used $\rightarrow$ symbol. With the *Parallel* composite it is possible to execute multiple behaviors in parallel. This is for example necessary to model the independent behaviors of two vehicles in parallel. Here, the *executeTick()* method call is passed to all child behaviors in each tick. Once a child behavior succeeds, it is not ticked anymore in the subsequent steps. The *Parallel* composite succeeds once all child behaviors succeed. Same as with the sequence, a *kFailure* of one child causes a *kFailure* of the whole composite. *Parallel* composites are represented with the $\rightrightarrows$



(a) Action. (b) Decorator. (c) Composite.

Figure 3.7: Graphical convention of behavior node primitives and their composition.

symbol. To allow reactive behaviors, most implementations provide also a *Selector* composite (sometimes also refereed as *Priority Selector* or *Fallback Selector*), which is labeled with the ? symbol. This composite executes the child behaviors subsequently within a tick until one child behavior can handle the situation with a *kRunning* or *kSuccess* state. Hereby it is allowed that a child behavior returns a *kFailure* as long there is still a fallback child behavior which is not failing. Besides these standard composites, any other specialized composite is allowed to combine multiple behaviors. These basic node primitives (composites, decorators, and actions) are the building blocks, which allow to build up any arbitrary behavior tree. Hereby there is no limitation regarding the tree structure in terms of depth or width.

3.1.6 Evaluation

To determine the best fitting control architecture for the Yase middle end, the presented architectures are evaluated in the following against each other. Hereby the ISO quality goals presented in 1.4.2 with their sub characteristics are used. Each architecture is given a point in a category if they stand out in comparison to the other architectures. The decisions are hereby based on the discussions of the previous sections.

Functional Suitability

This quality goal is divided into three sub characteristics. The functional completeness defines the degree to which specific tasks and user objects are covered. FSMs, H-FSMs and BTs are used in practice for a long time in nearly all fields. Therefore, it can be assumed that they could cover all aspects. OSC1 and DT are on the other hand restricted to certain domains such as decision making or lack of elementary operations such as sequences. Functional correctness defines the degree to which a product provides correct results. Here, all architectures are comparable except BTs, which show additional benefits in this field due to their explicit *kFailure* state. This execution state allows inherently build in failure management. Functional appropriateness is the degree to which the architecture facilitates the accomplishment of specified tasks and objectives. Since this prerequisites completeness, these architectures are within in the shortlist for appropriateness. However due to the hard to maintain complexity of FSMs, only H-FSMs and BTs are considered here as appropriate.

	FSM	H-FSM	DT	OSC1	BT
Completeness	+	+	0	0	+
Correctness	0	0	0	0	+
Appropriateness	0	+	0	0	+

Table 3.1: Comparison of the functional suitability.

Usability

The quality goal is divided into six sub characteristics, from which only three are considered here. To make conscientious evaluations for the categories appropriateness recognizability, user interface aesthetics and accessibility, more user studies would need to be done. Learnability on the other side is given for all broadly known architectures, as they are teached as basics in the field of computer theory. The exception is OSC1, which is a quite complex standard with a lot of peculiarities. In the field of operability, BTs and DTs are given a point since they provide easy entry points and a clear tree structure. For user error protection, the BTs get credit again for introducing failures as an inherent state.

	FSM	H-FSM	DT	OSC1	BT
Learnability	+	+	+	0	+
Operability	0	0	+	0	+
User error protection	0	0	0	0	+

Table 3.2: Comparison of usability.

Maintainability

The maintainability criteria define the ability to modify, improve, correct, and adapt a system to changes of its environment or requirements. In the subcategory modularity, all tree-based approaches have superior characteristics compared to graph structures, since they have minimal and clear inter dependencies as shown before. Since H-FSMs are a step towards this direction, they are given a point as well for this category. In terms of reusability and modifiability the tree-based approach is also beneficial. Furthermore, a second design pattern of behavior trees makes them superior to FSMs and H-FSMs in terms of reusability. In FSMs the implementation of states contains the transitions towards other nodes together with the actual state task.

This is can be compared to *goto* statements within code, which are considered harmful [51]. Furthermore, the tight coupling of transitions and tasks to perform is complicated to separate afterwards. In the concept of behavior trees, tasks and transitions are separated. Action nodes solely perform tasks, while the task of transitioning between nodes is handled specialized composite nodes. This decoupling makes the concept more modular in the sense of separation of concerns. Regarding analyzability, there is again a clear advantage for the tree-based approaches. Failures and impacts of changes can be traced back easily along tree branches. In the last category testability, none of the concepts exhibits noteworthy benefits.

	FSM	H-FSM	DT	OSC1	BT
Modularity	0	+	+	0	+
Reusability	0	0	+	0	+
Modifiability	0	0	+	0	+
Analysability	0	0	0	0	+
Testability	0	0	0	0	0

Table 3.3: Comparison of maintainability.

Decision For Yase

In table 3.4 the results of the quality goals are accumulated. It can be observed that the behavior tree approach shows superior characteristics in nearly all criteria. One reason for this good result is that behavior trees

	FSM	H-FSM	DT	OSC1	BT
Functional Suitability	1	2	0	0	3
Usability	1	1	2	0	3
Maintainability	0	1	3	0	4
Overall	2	4	5	0	10

Table 3.4: Comparison of the different architectures.

incorporate the best concepts of the other architectures while using proven good software practices such as separation of concerns. As discussed in the section 3.1.5, behavior trees advance on the one hand the hierarchical abstraction of H-FSMs by using a tree structure. On the other hand, they separate tasks and transitions between nodes into two independent problems. Furthermore, they incorporate the idea of decision trees. As shown in

[47] the FSMs, H-FSM and DT architectures can be expressed with behavior trees. Since the OpenSCENARIO XML approach is also a mixture of the previous mentioned architectures, it can be also modeled as behavior tree as well. This gives the best of all architectures. Another reason is the simplicity, which allows users to easily understand, visualize, and narrow down failures to sub trees. The effect can be observed since all gaming engines provide some sort of behavior tree packages for game designers to model their behaviors. With the superior characteristics and the widespread usage across many industries, behavior trees are the most promising candidate for the Yase middle end. Due to that reason, the following concepts will be built on behavior trees as basis. The appropriateness of behavior trees as an intermediate representation layer shall be underlined with the following quote:

> *"The behavior tree becomes a domain specific language (DSL) with efficient interpreter."[52]*

3.2 Behavior Tree - State of the Art

Since Behavior Trees are not a standardized concept, the most influential implementations and publications are presented first. The relevance criteria in this thesis are either influential publications in this research area, relevance for intelligent vehicle simulation or widely used implementations.

BehaviorTree.CPP Implementation

The BehaviorTree.CPP project [53] provides a C++ library with a behavior tree framework. With more than 350 forks and an active community it is the most popular behavior tree project for C++. It bases on the influential work "Behavior Trees in Robotics and AI" of Michele Colledanchise and Petter Ogren [47]. While its original focus is on robotics, it is also intended to be usable in other contexts. Since the implementation is provided as a lightweight C++ library, it can be used even in the smallest projects. This is essential for the usage in all levels of *incremental simulation*. Besides the core library with the behavior tree, it provides also a rich eco system with a GUI and convenience functionality such as logging. One downside is the XML based configuration of trees, which is tightly coupled in the implementation. Furthermore, it partly consists of outdated concepts (such as condition primitives) of the so called first generation of behavior trees.

Game AI Pro Book and Implementation

Since behavior trees are state of the art in the gaming industry, it is also elaborated in the Book "Game AI Pro: Collected Wisdom of Game AI Professionals" [52]. This book provides advanced insight in gaming development and presents implementation strategies for different degrees of performance. A whole section within the book deals with behavior trees. It elaborates several sophisticated implementations strategies and shows the evolution of implementations over time. A github repo with small examples for the different implementation strategies accompanies the book [54]. With more than 380 stars and 119 forks, it is also a widely accepted implementation. However, the repo is not actively maintained, and the last commit is older than 10 years. Compared to the BehaviorTree.CPP project, the provided features are limited. While the book provides deep insight in the implementation strategies, it provides code wise only an exemplary implementation without real eco system.

Unreal Engine - Behavior Tree Implementation

The Unreal Engine is a proprietary game engine to build sophisticated games. It also contains a sophisticated behavior tree implementation for game designers to create NPC logic [48]. To build such behaviors, an interactive GUI is provided, which allows an easy interaction with behavior trees. The documentation demonstrates the usage of modern behavior tree design of the so called "second generation" with the usage of decorators such as "Services". The actual implementation is built within the gaming engine, allowing to use a highly performant and deeply integrated implementation which is hidden behind an intuitive GUI. While the documentation demonstrates good practices and examples, the proprietary and performance orientated implementation contradicts the required usage as an open, modular, and light weight C++ library.

Carla Simulator: Scenario Runner

The Carla simulator is with more than 2.2 thousand forks and 7.5 thousand stars one the most actively developed open-source simulator in the intelligent vehicle domain [19]. Furthermore, it is also the first major simulator, which uses the behavior tree concept for scenario simulation. The implementation is available as a plugin called *scenario_runner* [55]. It allows to write and execute scenarios either via python or OpenSCENARIO XML. The implementation is based on the Python library *py_trees*, which was originally motivated by robotic use cases [56]. The fact that it is written in Python allows according to the *py_trees* developers [56] "rapid development of medium scale decision engines that do not need to be real time reactive". They estimate the typical latency of human interaction use cases with robots to 50-200ms and set this as a reference point. While the tool is beneficial such rapid prototyping, the performance is not sufficient for many intelligent vehicle simulation use cases such as HIL applications with real time requirements.

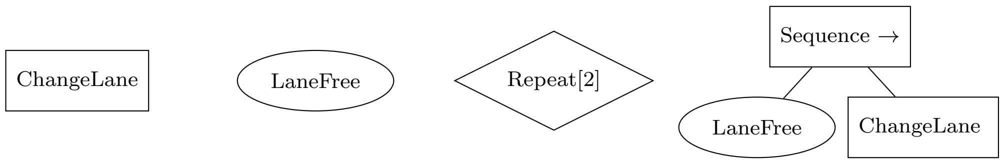
3.3 Behavior Tree Implementation Variants

Since the first occurrence of behavior trees in the game *Halo*, they have been used and evolved for many practical fields. This leads to different implementation variants beyond the common basics explained in the previous section. In the following, common variants of node primitives and their graphical representation are explained. Next, performance optimization with event driven behavior trees is presented. Last, a data sharing approach called blackboard is discussed, which is common in most implementations.

3.3.1 First and Second Generation of Behavior Trees

As Behavior Trees became an essential foundation of the gaming industry, great efforts were made to improve execution performance and simplify the practical usage. This led to an evolution of behavior trees, which is typically categorized into first and second generation behavior trees [52]. During the behavior tree evolution, different sets of behavior node primitives and graphical representations were invented.

The most famous representative for "first generation" behavior trees is the work of Michele Colledanchise and Petter Ogren [47]. Next to the previously present primitives in section 3.1.5, an additional node primitive is used. *Condition* nodes are here considered as an independent primitive next to *Actions*, *Decorator* and *Composites*. They are used as leaf nodes like actions with the sole purpose to check conditions. Contrary to actions, conditions are not allowed to perform actual behaviors. Such *Condition* nodes are used in their work to influence the logical execution flow of the behavior tree based on events. This is often needed to model reactive behaviors in which a behavior should be performed under certain circumstances. A typical use case can be seen in Figure 3.8d. In this example, a lane change behavior is delayed with the help of a sequence until the condition node becomes non-blocking. Besides its important theoretical contribution to this research field, the book presents the classical visual representation for behavior trees. The typically used graphical convention are rectangle shapes for actions and composites, diamond shapes for decorators and ellipse shapes for condition nodes as shown in Fig. 3.8.



(a) Action. (b) Condition. (c) Decorator. (d) Composite.

Figure 3.8: Graphical convention of first generation node primitives.

Later implementations aimed to reduce the implementation complexity and increase expressiveness and simplicity by removing superfluous logic and conventions. The so called second generation of behavior trees is characterized by the extensive use of decorators. Decorators can perform any task for the underlying subtree. One example are *Service* decorators, which can provide any services such as the calculation of resource heavy computation results for all underlying nodes. Furthermore, decorators are provided which allow to start or interrupt a behavior based on a generic condition. With this approach, the *Condition* node primitive is not needed anymore, since a decorator can check the condition under which a behavior should be executed. Therefore the first generation approach for a reactive behavior shown in Figure 3.8d can be redesigned with a decorator as shown in the Figure 3.9. This reduces the inherent tree complexity since it requires only two nodes instead of three nodes with composite branching. This eases the readability and reasoning, and it improves furthermore the execution performance.

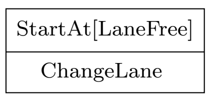


Figure 3.9: Condition check implemented as a decorator.

3.3.2 Event Driven Behavior Trees

To execute the specific behaviors of a behavior tree, the root node is executed at each step. From the root node, this execute call is passed down the tree to the corresponding behaviors. In this process composite nodes are traversed, and conditions are re-evaluated repeatedly, even there are no changes. Especially with larger behavior trees, this can cause computational overhead. To gain even smallest performance boosts, gaming engines avoid this by using event driven behavior trees as presented in [52]. Here, the last relevant tree node positions are stored for the next step to avoid superfluous tree traversal. Furthermore, conditions are not checked every time, but contain an observer, which is triggered only when the event occurs. This allows to determine in a performant way which relevant subtrees must be executed for each step. While this maximizes implementation efficiency, it increases the complexity of the code and its usage. Event driven behavior trees require the implementation of event observer patterns and the storage of sub tree entry points. This complex tree management makes it easily error prune to use and complex to understand. In [52], it is explicitly stated, that this complexity should only be introduced if the performance is really needed.

3.3.3 Blackboards

Behavior nodes typically perform their tasks by modifying data such as a simulation environment data container. As an example, the *ChangeLane* behavior may modify the vehicle position and orientation in the data container step by step during the execution. Moreover, this node may rely on further data, such as map data to decide how to perform the lane change based on road information. The distribution of such data is a non-trivial problem. First, not all data is always accessible in the moment of the behavior node instantiation. This data must be shared later consistently among multiple nodes in the tree. Secondly, each node has its own requirements on what data they need. To solve this problem, most behavior tree implementations provide a key value storage which allows to share data from a node on with its subtree. This so called “BlackBoard” allows to register data of any generic type with a unique key identifier from within a node. The blackboard instance is than passed down the tree and each sub node can then retrieve its required data with the key. Thus, it is easily possible to share data across nodes.

3.4 Agnostic Behavior Tree of Yase

In this section the design of the Agnostic Behavior Tree (ABT) library is presented. The package is designed as a pure C++ library, which is agnostic of any input language and simulator specifics. Due to the requirement of incremental simulation, it is designed to be usable in multiple levels. For this, it needs to be executable either as a standalone library or in combination with the other Yase stages as the intermediate scenario representation. In the following, first the reason for an own development is explained. Afterwards, all implementation details with the design decisions are elaborated. Hereby the implementation follows in general the most canonical way of second-generation behavior trees.

3.4.1 Reason for Yase Development

With plenty of other existing options, it is always the question if another implementation is needed. In the current case, especially the BehaviorTree.CPP [53] implementation is a promising candidate. This C++ library fulfills the constraint of C++ based code, which is unit test ready. However, three reasons favor a new and fresh implementation: First, the BehaviorTree.CPP already has other dependencies build within. With its built-in XML configuration, there is a dependency towards other libraries. Secondly, the library is based on the first-generation behavior tree concept. And last, the design is mainly designed for robotics. For this reason, an own development is presented, which has only C++ dependencies, is built with second generation design principles and contains specialization towards the needs of intelligent vehicle simulation.

3.4.2 Behavior Node Implementation

The behavior nodes are implemented in an object-orientated way. All behavior nodes need to be inherited from the abstract base class *BehaviorNode* shown in Fig. 3.10. This base class declares all necessary methods to interact with a behavior in terms of data distribution, initialization, behavior execution and termination. Based on this *BehaviorNode* class, the corresponding node primitives *ActionNode*, *DecoratorNode* and *CompositeNode* are derived (see example code in 3.11). The superfluous node primitive *ConditionNode* is omitted, since it can be expressed more efficiently via decorators as it was shown in section 3.3.1. The *ActionNode* class is the base class for behaviors, which perform atomic tasks and are not split further. Decorators can be derived from the *DecoratorNode* class. The implementation provides API methods to set a child behavior and access it again during execution. The child behavior handling ensures that only one valid child can exist for a decorator. Furthermore, the default implementation forwards necessary method calls to the child behavior node, such as the *distributeData()* method call. The *CompositeNode* base class allows to build composites with multiple child behaviors. Like the decorator base class, it ensures forwarding of necessary

```cpp
// Execution state of a node
enum class NodeStatus { kIdle,
                        kRunning,
                        kSuccess,
                        kSuccessRunning,
                        kFailure };

// Abstract base class for Behavior Tree Nodes.
class BehaviorNode {
 public:
  // Type alias
  using Ptr = std::shared_ptr<BehaviorNode>;

  // Start distribution of data among subtree from outside
  virtual void distributeData();

  // Inits/ resets the node before first tick
  virtual void onInit() = 0;

  // Method to invoke tick from the outside
  NodeStatus executeTick();

  // Terminates behavior after it finishes
  virtual void onTerminate();

 private:
  // Defines the logic of the node
  virtual NodeStatus tick() = 0;

  // Lookup and register data from blackboard for this node
  virtual void lookupAndRegisterData(Blackboard& blackboard);

  // Hidden Constructor, only accessible to friend classes
  BehaviorNode();
  friend class CompositeNode;
  friend class DecoratorNode;
  friend class ActionNode;
};
```

Figure 3.10: Simplified C++ extract of the behavior node base class.

```cpp
// Base class for action nodes
class ActionNode : public BehaviorNode {};


// Base class for decorator nodes
class DecoratorNode : public BehaviorNode {
   public:
    // Set the child behavior to decorate
    void setChild(BehaviorNode::Ptr child);

    // Provides access to managed child
    BehaviorNode& child();

    // Distributes data for this node and child node
    void distributeData() final;

   private:
    // Child behavior to decorate
    BehaviorNode::Ptr m_child_node{};
};


// Base class for composite nodes
class CompositeNode : public BehaviorNode {
   public:
    // Add an additional child at end
    void addChild(BehaviorNode::Ptr child);

    // Distributes data for this node and all child nodes
    void distributeData() final;

    // Provides access to specific managed child
    BehaviorNode& child(size_t index);

   private:
    // Child behaviors
    std::vector<BehaviorNode::Ptr> m_children_nodes{};
};
```

Figure 3.11: Simplified C++ extract of the three node primitives.

```
// Prepare: Distribute data and init tree
behavior_tree_node.distributeData();
behavior_tree_node.onInit();

// Simulate: Execute until behavior tree finishes
NodeStatus status = NodeStatus::kRunning;
while (status == NodeStatus::kRunning) {
  status = behavior_tree_node.executeTick();
}

// Cleanup: Terminate the behavior tree
behavior_tree_node.onTerminate();
```

Figure 3.12: Execution of a behavior tree in Yase.

method calls. Also, it provides methods to add child behaviors and access them later. The method *addChild()* is hereby by intention named differently than the decorator method *setChild()* to emphasize the difference of setting one child and adding multiple children. To prevent behavior node designers from deriving directly from the *BehaviorNode* base class, its constructor is made private. Only the derived three node primitives have as friend class access to the constructor. This enforces to derive nodes from one of the three node primitives.

3.4.3 Behavior Execution and Execution States

The typical interaction with a behavior tree is defined as shown in the code snippet in Fig. 3.12. The *behavior_tree_node* represents a behavior node with an underlying behavior tree. During the execution, any node encounters four method calls in the following order: First, the initial data distribution is triggered from outside via the *distributeData()* call, which causes the call of the node specific *lookupAndRegisterData()* method implementation. The *distributeData()* method is already implemented and ensures that the method call is passed forward recursively until each leaf node is reached. This data distribution is done only once as the first step before the actual execution of the behavior tree. Since the data is then distributed, the *onInit()* method is called next. The method call is always done just in time right before the first execution of the behavior. It allows to prepare the node state for execution and to reset the node state to a clean state. From this point on the node is ready to be executed. This is done from the outside with the public method *executeTick()*, which forwards (besides internal checks and state handling) the invocation to the node specific *tick()* method implementation. Usually, the behavior is then re-executed until it finishes. This can be observed with the returned execution state. However, there is no guarantee that a behavior is executed until it finishes successfully. In any case, the *onTerminate()* method is called last. The node specific method implementation allows to perform any cleanup such as the release of locks and resources.

The number of execution states (in the implementation named *NodeStatus*) of the behavior nodes is hereby extended from the classical three states as shown in section 3.1.5 to five states. The *kIdle* state is an internal start state only and it is not allowed to be returned via the *tick()* method. If it is returned, the state is captured within the *executeTick()* method and an exception is thrown. Since this state has no relevance, no color is used for this state in the visual representations. The *kRunning* state indicates a behavior which is not finished within this tick. It indicates furthermore that the node potentially changed the global data state by performing behavior tasks. As before, the color blue is used to visualize such running nodes. The *kSuccess* and *kFailure* state represent the end of a behavior, either a successful one or due to a failure. To follow the "return early" design pattern [57], these states should be returned directly before altering the global data state. After the return of these states, the *onTerminate()* method will be called immediately to clean up the node. From this point on the node is not executed any more, unless it is reset before with an *onInit()* call. In visualizations, the states are indicated with a green and red color. Besides the three classical states, there is one additional state which is needed due to simulation specific constraints. The *kSuccessRunning* state is, as the name indicates, a mixture of *kRunning* and *kSuccess*. On the one hand it is *kRunning*, meaning it still preform some tasks. On the other hand, it is also *kSuccess* at the same time, indicating that the behavior intent is already fulfilled. This is important to flag behaviors, which already fulfilled their purpose, but can continue to perform the behavior as long as it is executed. The need arises from technical constraints of different scenario languages. As an example, the OpenSCENARIO DSL standard defines a follow lane behavior, which runs as long as the scope it is placed in if the optional duration parameter is not given. In Figure 3.13 an example is shown for such a behavior. Here, one vehicle should follow the lane in parallel while the other vehicle performs a lane change behavior. According to the standard, the composed behavior must end once the lane change is finished successfully, since the follow lane behavior should only run as long as the parallel scope. This cannot be performed sufficiently with the classical three states. If the follow lane behavior would return *kRunning* all the time, then the parallel composite node could never detect when the behavior can be terminated. If the follow lane behavior would return *kSuccess* all the time, then the parallel composite node would terminate the behavior instantly after the first execution. The behavior would then not be executed any more in the subsequent steps and therefore the vehicle would potentially not follow the lane anymore. The *kSuccessRunning* state allows to solve this problem without creating interdependencies between the separate behavior branches. To visualize this execution state, a cyan color is used in the visualizations.

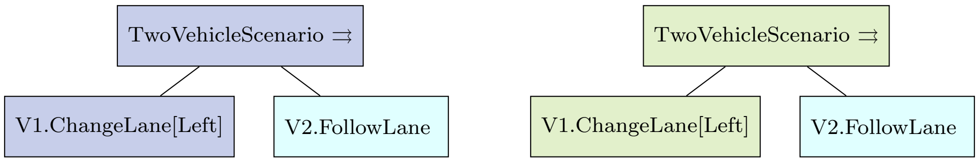


Figure 3.13: Exemplary execution of a parallel behavior with a SuccessRunning child behavior.

3.4.4 Tasks

Behavior nodes perform tasks, which are usually directly implemented within the nodes. However, the library provides additionally several generic Task classes with restricted scopes dependent of the tasks itself. This allows implementation agnostic reasoning on combination and execution of such tasks. It is especially important for writing agnostic decorators, as shown in the next section. The library comes with several generic task types: Many nodes need to perform behaviors based on condition evaluation. For this reason, a generic *ConditionTask* class allows to define condition checks. Like nodes, the task can access the blackboard to look up data needed for the condition evaluation. Furthermore the *onInit()* call allows to set initial states as well as to reset the condition from outside. To logically evaluate multiple conditions, logical operators are provided as *AndCondition*, *OrCondition* and *NotCondition* classes, which combine other conditions into one. By inheriting from the *ConditionTask* class, domain specific conditions such as *ElapsedTime* or *DistanceDriven* can be implemented. Such conditions can then be passed as an argument to special decorators such as the *StartAt[]* and *StopAt[]* decorator. The *ServiceTask* is on the other side a task, which provides a service such as a resource consuming calculation. This can be performed before or after the child behavior execution. As the other tasks, services have access to data shared via the blackboard. The results of the service are hereby made accessible for the subtree via the data shared with the blackboard. One or multiple service tasks can than be combined and passed to a special *ServiceNode*. The implementation of such tasks can be seen in Figure 3.14.

```
// Base class for condition tasks
class ConditionTask {
  public:
   // Lookup and register data for condition evaluation
   virtual void lookupAndRegisterData(const Blackboard& blackboard);

   // Inits & resets the condition
   virtual void onInit() = 0;

   // Evaluate condition
   virtual bool evaluate() = 0;
};

// Base class for service tasks
class ServiceTask {
  public:
  // Lookup and register data for service evaluation
  virtual void lookupAndRegisterData(Blackboard& blackboard){};

   // Inits & resets the service
   virtual void onInit(){};

   // Pre update method of the managed service, called before decorated child is ticked
   virtual void preUpdate(){};

   // Post update method of the managed service, called after decorated child was ticked
   virtual void postUpdate(){};
};
```

Figure 3.14: Simplified C++ extract of generic tasks.

3.4.5 Provided Set of ABT Nodes

The Agnostic Behavior Tree library comes with several behavior node implementations and convenience functions, which are independent of any project. Following composites are provided with the library:

- **Sequence:** Executes all child behaviors in sequence. For this it calls at the first tick the *onInit()* and *executeTick()* of the first child node. The execution invocation is forwarded until the child node returns *kSuccess* or *kSuccessRunning*. This invokes the *onTerminate()* of the finished child and the next child is initialized and executed. Once all child behaviors are finished successfully, the sequence finishes also. If one of the child behaviors fails, the whole sequence will fail as well.

- **Parallel:** The *Parallel* composite is used to run multiple behaviors in parallel. The *onInit()* call is directly forwarded to all child behaviors. Then, the child behaviors are executed until all children finish with *kSuccess* or *kSuccessRunning*. Behavior nodes which finish early with *kSuccess* before others are terminated and not executed in subsequent steps anymore. This composite expects as well that no child behaviors is failing.

- **Selector:** Reactive behaviors can be achieved with the *Selector* composite. This composite allows to try out in every tick several behaviors until one of them can handle a given situation. Hereby the behavior nodes are executed in priority from left to right. If a behavior returns *kFailure*, the next behavior is executed until one does not return a *kFailure*. If none of the behaviors can handle the situation, the composite returns *kFailure* as well. For resource efficiency, the behaviors are initialized and terminated just right in time.

Decorators are also essential building blocks, as they allow to extend existing behaviors in manifold ways. Together with the generic tasks presented in section 3.4.4, the following abstract decorators can be provided:

- **Service** A common problem is that some calculation results are needed redundantly in a multitude of nodes. An example is the prediction of all traffic participants in an environment, which is essential for driver models to base their decisions on. To avoid that each individual driver model node must calculate the prediction itself, such a service can be provided via *ServiceNode*. Here, the *ServiceNode* executes the *ServiceTask*, which provides then the information over the shared blackboard data for all other nodes in the underlying subtree.

- **Constraint:** This decorator allows to constraint a behavior to specific circumstances. As an example, a *ChangeLane* behavior shall only be executed, if the target lane is actually free. This can be implemented as a *ConditionTask*. If the passed *ConditionTask* does not evaluate to true, a *kFailure* is returned. The constraint condition can be checked either before the first child behavior execution, after the successful child behavior execution, or during every child behavior execution.

- **StartAt:** Allows to delay a behavior until the start *ConditionTask* evaluates to true for the first time. In this case, the child behavior is initialized and ticked from now without checking the condition anymore.

- **StopAt:** This decorator interrupts the underling behavior once the stop *ConditionTask* evaluates to true for the first time and returns *kSuccess*. The behavior is also interrupted even it is not fully finished yet.

- **Until:** Finishes an underlying behavior exactly when the *ConditionTask* evaluates to true. For this, either the underlying behavior must return *kSuccess* exactly when the condition evaluates to true or it must be *kSuccessRunning*, which can be promoted to *kSuccess*. If the underlying behavior finishes before or after, the decorator will return a *kFailure*.

- **Invert:** Inverts the execution state of its child node either from *kSuccess* to *kFailure* or vice versa.

- **Repeat:** This decorator repeats the underlying child behavior N times. Before each repetition, the decorator resets the behavior by calling *onTerminate()* and *onInit()*.

While the library provides multiple decorators and composites, the provided set of actions is quite limited. This is due to the fact that nearly all actions nodes are domain problem specific. Thus, the provided set of predefined action nodes is restricted to a minimal set to nodes for unit testing.

- **AnalyseNode** To mimic behaviors for tests, this node returns after a specified number of ticks a desired execution state, until then it returns *kRunning*. Furthermore it provides API methods to analyze later on how many times methods such as *onInit()* and *onTick()* were called. The *AnalyseNode* is the base class for the helper actions *AlwaysRunning*, *AlwaysSuccess*, *AlwaysSuccessRunning* and *AlwaysFailure*. With these behaviors it is possible to test the interplay of other decorators and composites. These nodes are also used for the library internal unit tests.

- **SleepNode** This node sleeps for a given duration before the defined execution state is returned. With this, realistic behavior trees can be mocked for testing.

3.4.6 Provided Set of Utility Methods

Additionally, to the provided sets of decorators, composites and actions, the library provides further convenience functions. First, the *runTree()* function assists to execute a behavior tree in a consistent way. The function is hereby implemented as already shown in Figure 3.12. Besides this tree execution function, there exists a multitude of tree print functions to enhance low level debugging. All of them allow to print the current state of the behavior tree on the terminal or in a logger format. The first is the *printTree()* function, which prints only the behavior tree structure overview. A more sophisticated analysis of the execution progress is possible with *printTreeWithStates()*. It prints additionally to the tree structure execution information such as the current execution state and the counter of when the node was ticked last. Furthermore, nodes ticked in the last step contain colorized states to identify currently active behaviors. To analyze the data distribution within the tree, the *printTreeWithData()* prints the tree and additionally the data which is registered by each node to the blackboard. This allows to reconstruct which data is shared at which level.

3.4.7 Implementation Guidelines

When a new behavior node is designed, there are four methods which need be considered for implementation. Furthermore, there are additional considerations to take into account when implementing decorators and composites. They are presented in the following.

Method lookupAndRegisterData()

The first interaction with a node is the distribution of data, which is done once at the beginning. A node can hereby hook in by implementing the *lookupAndRegisterData(Blackboard& blackboard)* method. The method is hereby optional to implement and gives read and write access to the passed Blackboard instance. This class allows to retrieve and register data with keys.

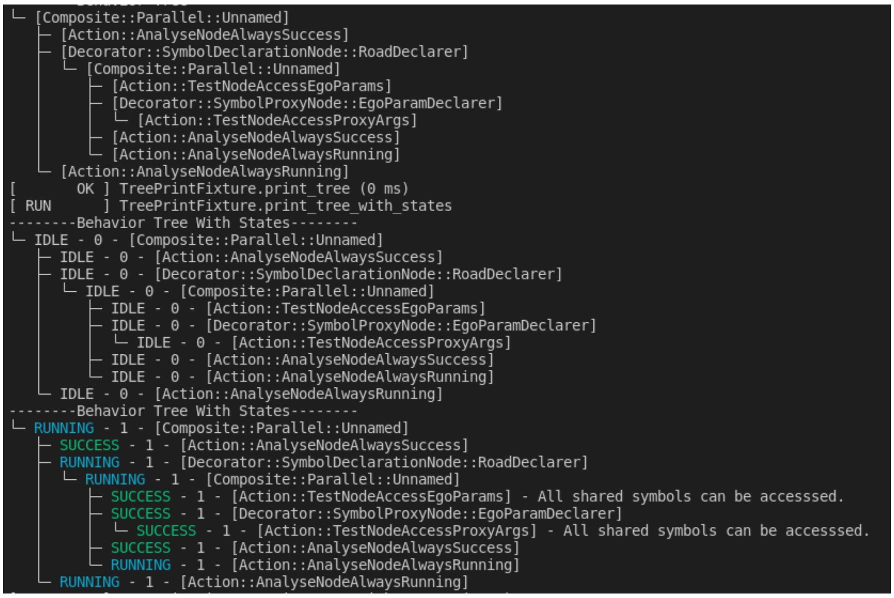


Figure 3.15: Different terminal convenience prints of a behavior tree.

The passed blackboard has not only access to inserted node local data, but also to the blackboard of the parent node. When data is searched with a key, then a recursive search beginning of the local blackboard is performed. If data is not found in the local blackboard, then it is searched in the parent blackboard until it is found, or the root node is reached. This way it is possible to register data within a node, which is then accessible for the whole subtree. Since the data is not accessible for the parent node, it furthermore ensures scoping of data. When new data should be inserted in the blackboard, a recursive search is first done to find potential key conflicts. If this is the case, an exception is thrown, otherwise the data is inserted in the local blackboard. Due to this process, unique keys can be guaranteed throughout the whole tree. The handling of the blackboard itself and the recursive search is hereby hidden for the user behind the API as presented in Fig. 3.16. This provides behavior node designers a convenient way to register new data for the subtree and find existing data.

Method onInit()

After data is distributed, the next method call of a node is the invocation of the *onInit()* method. It allows to perform any preparation tasks required for the behavior execution. This can include preparation calculations, setting initial states or allocating resources. For resource efficient execution, the method is called once just in time right before the actual execution. An es-

```cpp
// Blackboard interface
class Blackboard {
  public:
   // Checks if key exists and is accessible
   bool exists(const std::string& key) const;

   // Searches key and checks for expected type
   template <typename Type>
   bool exists(const std::string& key) const;

   // Get accessible data with key of given type
   template <typename Type>
   Type get(const std::string& key) const;

   // Declare new data with key of given type
   template <typename Type>
   void set(const std::string& key, Type value);
 };
```

Figure 3.16: Simplified extract of the C++ blackboard interface.

sential requirement when implementing the method, is that repeated *onInit()* calls reproduce the same clean node state. This state reset is required since a behavior can be repeated multiple times and the node must behave every time as it is a new node. To nudge users to implement the node state reset, the method is pure virtual by default. Therefore, it is enforced by design, that the node creator implements this function.

Method tick()

The most important method of a node is the implementation of the *tick()* method. In here, the node specific behavior logic is described. Since it is the most essential method, it is mandatory to implement for each node. The method does not pass in any input arguments; therefore the task is performed solely on the previously distributed data. A requirement for this method call is that it follows the "return early" design pattern [57]. Thus, a succeeding or failing behavior should return immediately without modifying the state of the global data. This prevents that a failing behavior leaves the data in a half modified and inconsistent state. Furthermore, it allows that an alternative behavior node can perform a behavior on the data. The state should only be altered if the execution states *kRunning* or *kSuccessRunning* are returned.

Method onTerminate()

Once a behavior should not be executed anymore, the node specific *onTerminate()* method is directly called. This can happen either because the node finished, or it is interrupted before. The method allows to clean up the node, such as by releasing allocated system resources. It has an empty default implementation and is therefore not mandatory to implement.

Noteworthiness on Decorators and Composites

Decorators and composites require additional attention when designing them. In both cases it must be considered that all elementary method calls are passed to the child behaviors. As an example, it is important that the *onInit()* and *onTerminate()* call does not only reset and cleans up the node itself, but the calls must be forwarded to the child behaviors just in time. The right timing depends on the specific node behavior. As an example, the sequence composite calls the *onInit()* of a child behavior just in the moment when the child behavior is in line. Since in most cases composites and decorators are implementation agnostic, it should be always checked if the desired behavior is already possible with the provided set of decorators and composites of the library.

3.5 Discussion

In this chapter possible control architectures candidates for the intermediate representation of the middle end were analyzed and compared. After an evaluation it was shown that behavior trees obtain the best characteristics regarding modularity and maintainability. Subsequent, several open-source behavior tree implementations and the most important architecture variants were presented. Hereby the implementation as well as terminology was compared and aligned to find the most canonical way. Since the existing projects are not able to fulfill all domain specific requirements, the Agnostic Behavior Tree (ABT) implementation of Yase was presented. For this, the design after the most canonical and modern design principles of the second-generation behavior trees is elaborated. Furthermore, the provided set of nodes and functions are highlighted, and implementation guidelines are given to ease the entrance for new developers. This serves as the foundation for the further work.

The presented implementation fulfills hereby the relevant requirements for the middle end. It was shown that the behavior tree concept fulfills best the defined quality goal 1.4.2 among all controller architectures. The deduced requirement of agnosticism of input formats and simulator bindings 1.4.1 is given by the fact that the concept is completely domain independent. Here, the behavior tree specific concept of blackboards has further benefits since the whole concept is independent of any world data representation. Since the library is written as a C++ library without any dependencies, it fulfills the requirement of simulation at each test level 1.4.1. The light way implementation and the fact that it does not rely on any network or inter-process communication and is therefore inherently unit test ready. During the development, the constraint of industry adequate C++ code 1.4.3 was taken into account with high test coverage of 97% code coverage and the usage of style guides and static compiler checks. The last constraint of simplistic code 1.4.3 is also fulfilled due to the fact that the concept is build up on the canonical and modern behavior tree concept. This eases the entrance level for new developers since the concept is well established.

Chapter 4

Back End: Holistic Scenario Modeling

In this chapter it is shown how a simulator specific backend is implemented with the introduced behavior tree and subsequent how it can be used to simulate all required use cases of holistic intelligent vehicle simulation. This is based on the prior work [58]. First, the architecture and the provided behaviors based on the ABT are elaborated. With this, the implementation of the Euro NCAP cut-in scenario is shown and further combinations with test evaluations are discussed. Subsequent, also the implementation of a driver and a traffic model is highlighted. With these scenarios and models, an exploitative and an explorative search is performed.

4.1 MockupSimulator Backend

For the following setups and experiments a so called *MockupSimulator* is build up to serve as a demonstrator. This simple simulator is designed to perform simple tests on a straight highway with only vehicles as traffic participants. The simulator itself is also build up consequently as a part of the behavior tree. Based on this simulator, a backend with vehicle action nodes is provided.

4.1.1 MockupSimulator Architecture

An intelligent vehicle simulator must provide several operating system similar services as shown in the paper [59]. The relevant ones are the scheduling, data management with access rights and system calls. While the Agnostic Behavior Tree library is mainly designed to represent scenarios, it is not restricted to this scope. All three simulator services can also be implemented with the help of a behavior tree node. The correct scheduling of tasks at the right time is taken care by the tree structure itself. Here, the behavior tree can be compared with the process tree of an operating system. The second important task is the data management with the possibility to restrict access rights. The scoped blackboard provides exactly this functionality as it allows to share data with restricted access only for its subtree. This again is like the data management within a process tree, where data access rights can be shared exclusively with a process sub tree. System call similar possibilities can be provided via a helper class, which is shared over the blackboard with all nodes. Since all of the required services of a simulator can be provided by the behavior tree approach, the *MockupSimulator* is build up as such. For this, the simulator is implemented as a decorator node named *Mockup-Simulator*, which is placed as a root node with the scenario subtree as its child.

The simplistic *MockupSimulator* decorator is designed as a simulator, which steps forward in time with discrete time steps. It provides a set of simulator data on which behavior nodes can perform their tasks on it. The *World* data container consists of the environment representation at the given time point. This container keeps track of all current objects in the environment and is provided as a read only container. Additionally a *Map* container provides the current map and the *Time* container provides the current time with the time delta to the next time step. Behavior tasks can be performed by staging environment changes for the next step in a *WorldDelta* container. All simulator data is hereby made available for the scenario behaviors in the underlying subtree via the blackboard. When the simulator node is executed, it first executes the child behavior with its underlying subtree. The scenario behaviors can then add the changes for the next step in the *WorldDelta* container. After the subtree execution is done, it is the responsibility of the simulator node to keep the data consistent for the next step. For this all staged changes of the *WorldDelta* container are applied into the *World*. Furthermore, the *WorldDelta* container is reset and the *Time* is incremented. Last, the retrieved execution state of the child node is returned. This allows to step a simulation forward until the simulation finished successfully or failed. While a simulator must not necessarily be implemented as a decorator node, the chosen approach has the benefit that all ABT utility functions shown in section 3.4.6 can be applied. This includes the print functions for debugging as well as the convenience function to run the behavior tree in a consistent way. In all following simulation examples, the *MockupSimulator* decorator is placed on top of the scenario behavior tree to execute shown in Figure 4.1.

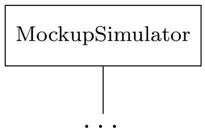


Figure 4.1: The MockupSimulator decorator node above a generic subtree.

4.1.2 Backend: Supported Vehicle Actions

To perform specific tasks on the simulator data, it is necessary to implement these behaviors as behavior nodes. The set of supported behavior nodes represents hereby the scope of the specific backend. It defines therefore what domain a simulator back end can support. For the following experiments a limited set of vehicle related action behaviors are provided. All these behaviors perform their tasks on the simulator data, which they retrieve from the blackboard.

In the simple demonstrator implementation, the necessary vehicle actions are restricted to three behavior actions for the straight highway use case. The first is the *FollowLane* action node with an optionally adaptable target speed parameter. This action follows the current lane and tries to hold the configured speed. The *FollowVehicle* action also follows the current lane, but adjusts the vehicle speed to maintain a safe distance to the vehicle in front. With the *ChangeLane* action, a vehicle can change its lane. The lane change direction is hereby a parameter. To combine these actions within more complex logic, a set of conditions is provided, which can be reused in the provided decorators of the ABT such as *StartAt* or *StopAt*. The first is the *[N]Seconds* condition, which evaluates to true after a certain duration is reached. The *LaneUnblocked* is a condition, which checks for a certain vehicle if the lane ahead is unblocked. Hereby it takes the velocity of the current vehicle and the vehicle in front into account. Similar to this the *Lane[Left/Right]Free* condition checks if a adjacent lane is safe to change to. Hereby also the velocities of the closest vehicles are considered to ensure an adequate safety zone. These actions and conditions serve as the basic set of building blocks for the following setups. If needed, this set can be extended any time for a specific setup with further specific actions or conditions.

4.2 Implementation of Concrete Scenarios

The *MockupSimulator* back end is now used to implement the Euro NCAP Cut-in scenario [11]. First the scenario itself is discussed and implemented

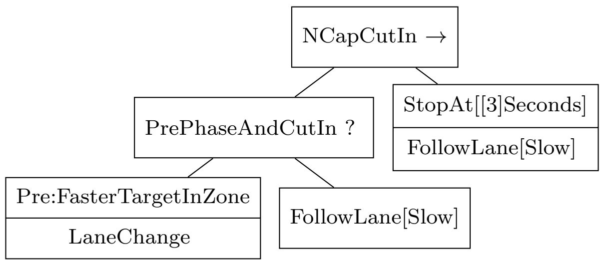


Figure 4.2: The scenario behavior Euro NCAP Cut-in.

with the provided set of nodes. Then an exemplary execution progress is explained in detail. Last, it is shown how the scenario can be directly extended with an evaluation of test and success criteria.

4.2.1 EuroNCAP Cut-in Scenario

In the following the Euro NCAP Cut-In scenario is used to demonstrate the implementation of a concrete scenario. In this scenario, a vehicle under test needs to proof a sufficient reaction regarding a sudden cut in of a slower vehicle. Hereby the vehicle under test drives with the allowed lane speed, while the challenging vehicle drives in front with a slower speed on the adjacent lane. Once the vehicle under test is close enough, the challenging vehicle performs an unexpected lane change towards the lane of the vehicle under test. As the vehicle under test drives with higher relative speed, it must react immediately to prevent a crash.

In Fig. 4.2 one possible behavior tree implementation can be seen. For simplicity reasons, it is assumed in this example that the vehicle under test is fully controlled by an external logic and only the challenging vehicle behavior needs to be modeled. In this particular implementation the scenario is divided into the combined *PrePhaseAndCutIn* phase and the post phase. The first phase is designed to react interactively on the vehicle under test. For this reason the *PrePhaseAndCutIn* selector first tries in every step to execute the critical cut-in. A decorator checks as a preconstraint with a test specific condition if the necessary situation of a faster adjacent target vehicle behind is given. If this preconstraint is not fulfilled and therefore fails, the *LaneChange* is not performed. Instead the *Selector* executes the fallback behavior *FollowLane[Slow]*. With this constellation it is ensured that the vehicle drives slowly until the vehicle under test is in the desired position. Once the critical cut-in was performed successfully, the root *Sequence* composite switch to the post phase behavior. This behavior keeps the vehicle in front with the slower speed until a certain duration is elapsed, which allows to observe the reaction of the vehicle under test.

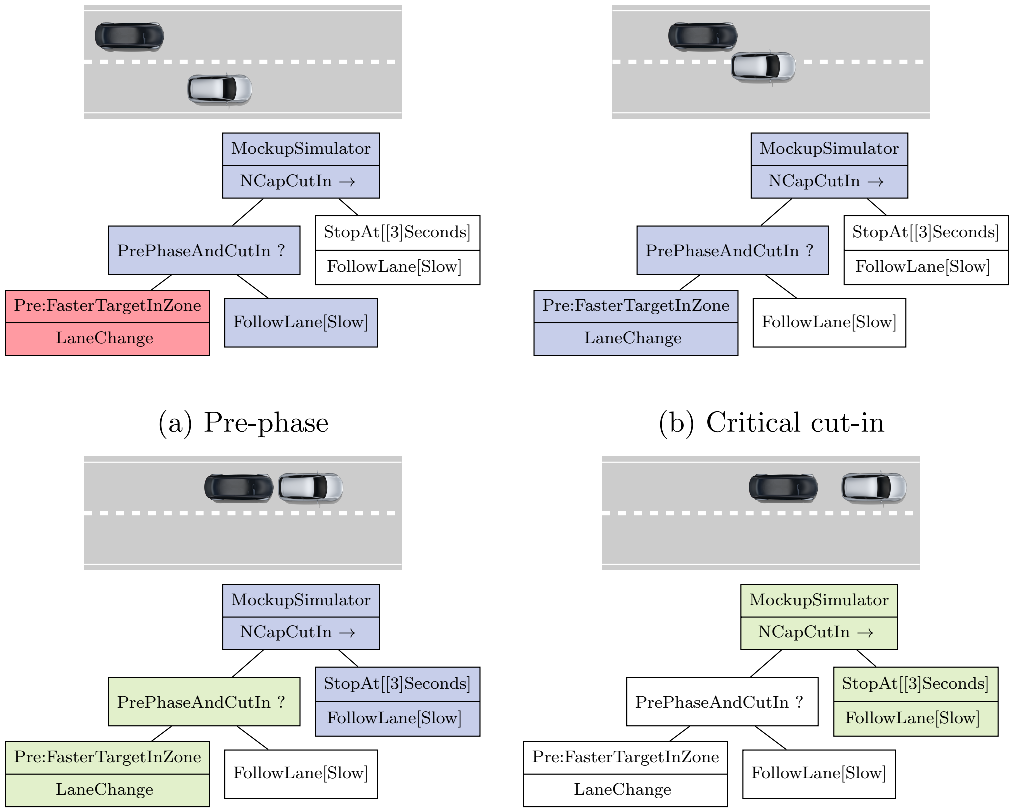
4.2.2 Simulation of the Euro NCAP Cut-in Scenario

Since the scenario behaviors require simulator data, it is appended under the *MockupSimulator* node. This node provides the necessary simulator data for the scenario tree and keeps it consistent after each step. Executing now the root simulation node simulates the scenario step by step. In Fig. 4.3 the progress of the behavior tree state of such an exemplary simulation is shown.

During the first steps the vehicle under test is yet too far away for the challenging vehicle to perform the critical lane change. In Fig. 4.3a it is shown how the adequate behavior is chosen in this situation. The execution of the *MockupSimulator* passes the execution tick to the scenario behavior *NCapCutIn*. This sequence passes the execution to the selector *PrePhase-AndCutIn*. As the preconstraint *Pre:FasterTargetInZone* fails, the selector ticks the fallback behavior *FollowLane[Slow]*. After it added the change to the simulator data, it returns the *Running* execution state, which is passed back through the tree to the root node. In the next phase in Fig. 4.3b the execution call is passed down again to the selector *PrePhaseAndCutIn*. Differently than before, the *Pre:FasterTargetInZone* constraint now evaluates to true, which in succession does not block any more the execution of the *LaneChange*. In the following steps the *LaneChange* action is executed again until it finishes with *Success*, as it can be shown in Fig. 4.3c. This returned successful execution state sets also the *PrePhaseAndCutIn* to *Success*, which in succession causes the sequence *NCapCutIn* to trigger the next behavior in the same step. This next behavior lets the vehicle drive again slowly with the *FollowLane[Slow]* action. The decorator *StopAt[3]Seconds* finishes hereby the scenario after enough time passed to observe the reaction of the vehicle under test.

4.2.3 Scenario Checks and Test Criteria

With the previous scenario a certain scenario logic for the challenging vehicle is described. However, executing this scenario does not ensure that the intended scenario flow will actually happen in the desired way. Furthermore, it is not tested if the vehicle under test meets the test criteria. For automated testing this is of special interest as large numbers of simulations need to be executed on regular basis without human oversight. This requires the automated check of test criteria and checks to avoid unintended deviation from the desired scenario flow. Such scenario checks and test criteria can also be



(a) Pre-phase

(b) Critical cut-in

(c) Post-phase

(d) Scenario end

Figure 4.3: Phases of the Euro NCAP Cut-in scenario with its behavior execution.

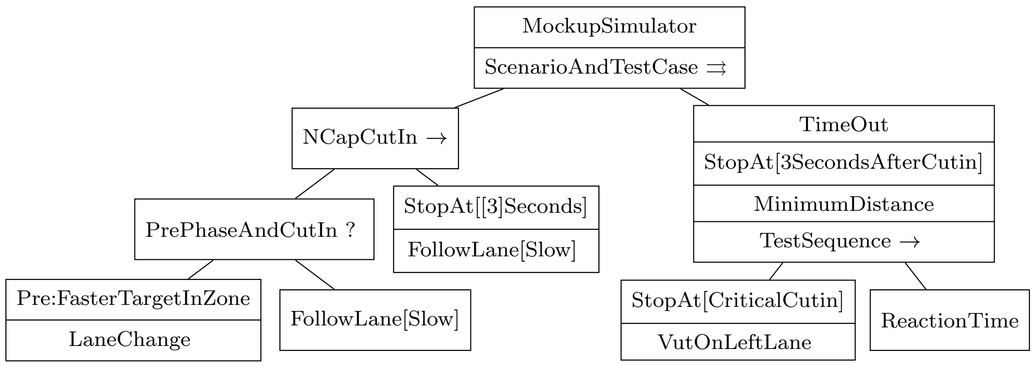


Figure 4.4: Simulation setup of the Euro NCAP Cut-in scenario with a separate test case.

integrated within the behavior tree. Of special help are here the *Constraint* decorators introduced in section 3.4.5. As an example, the following scenario checks could be adapted for the given Euro NCAP Cut-in scenario: To ensure that there even exists a suitable vehicle under test, the *VutOnLeftLane* constraint can check its existence on the left lane. Under the assumption that the vehicle under test can potentially circumvent the critical situation with a lane change, it can be checked that the vehicle under test also stays on this lane until the critical cut-in. A *TimeOut* constraint can limit the simulation to a certain duration. If the simulation does not succeed within this duration, this constraint causes a failure. Besides the scenario checks, the following test criteria can be used to test a valid behavior of the vehicle under test: The reaction of the vehicle under test can be evaluated with the *MinimumDistance* constraint, which checks if a certain velocity dependent distance between both vehicles is kept during the simulation. Also a *ReactionTime* constraint can check for a defined reaction time. Since all these checks and test criteria can be implemented as decorator nodes, they can be integrated at any place of the tree.

The traditional approach is to keep such scenario checks and test criteria in a separate test case. This test case is then kept independent of the scenario itself, which increases interchangeability of test cases and scenarios. In Fig. 4.4 such an implementation is illustrated. Below the *MockupSimulator* node the composite *ScenarioAndTestCase* combines the scenario and the test case behavior in parallel. In the test case subtree the constraints can be checked easily at the start, end or over the whole scenario duration. However, if test criteria need to be checked at specific scenario events, the logic of the scenario needs to be remodeled in the test case tree as well. As an example, it

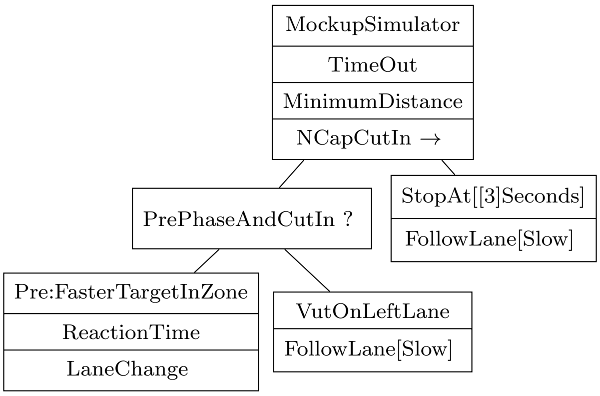


Figure 4.5: Simulation of the Euro NCAP Cut-in scenario with integrated test criteria.

requires the *TestSequence* composite, which checks first the *VutOnLeftLane* constraint until the critical cut-in starts. The subsequent constraint *ReactionTime* checks then the reaction time. To align the test case end with the scenario end, an additional *StopAt[3SecondsAfterCutin]* decorator is required to sync the scenario end. In the current example, the test case tree complexity is comparatively simple. However, at the worst case a separate test case must remodel the whole scenario flow again to check specific constraints at certain scenario events.

This redundant logic to determine certain scenario events can be avoided by integrating the constraints directly within the scenario behavior tree. In Fig. 4.5 such an integrated approach can be seen. As the scenario situations are processed by the scenario tree itself, the context specific constraints *VutOnLeftLane* and *ReactionTime* can be directly bound to the execution of the specific behavior parts. This reduces the complexity, as the context of a scenario (such as when a certain vehicle tries to perform a LaneChange) is known, while in post evaluation such a context is hard to identify later. On the downside, this approach reduces interchangeability of test cases. In summary it can be stated that the flexible behavior tree structure allows use both approaches and even combinations of both depending on the use case.

4.3 Implementation of Models

Exploratory searches are necessary to find unknown critical situations, which are not yet covered with concrete scenarios in regression tests. One approach is to generate natural traffic with realistic driver models to find unexpected

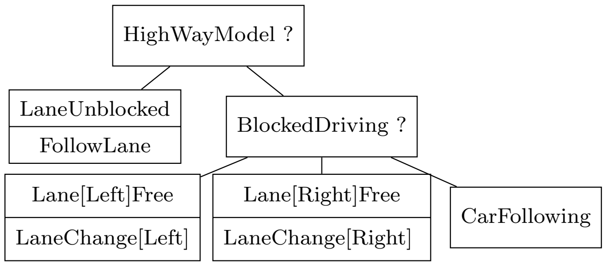


Figure 4.6: Highway driver model behavior composed out of basic behaviors.

traffic situations. In the following it is shown how such driver models can be build up with the basic behaviors of the backend. Afterwards this driver model is used to build up a traffic model.

4.3.1 Highway Driver Model Behavior

Driver models are one of the most complex models in intelligent vehicle simulation. Such driver models must be able to drive in a realistic and collision free manner within complex traffic situations. This task and its inherent complexity are similar to the complexity of autonomous vehicles. Behavior trees with their clear hierarchical structure allow to simplify this problem. This is demonstrated with the exemplary driver model presented in Fig. 4.6. This simple highway driver model is built up with the provided action nodes of the backend. It fulfills the design scope of the *MockupSimulator* with its restriction to highways without junctions. Thus, the root selector behavior *HighWayModel* must only differentiate between two situations: Either the lane in front of the vehicle is unblocked, or it is blocked, and it requires an adequate reaction. The first behavior is the *FollowLane*, which is decorated with a *LaneUnblocked* constraint. This enables the behavior only in situations in which the lane is unblocked in front. If the lane is blocked, the adequate reaction is handled by the *BlockedLane* selector. First, the *LaneChange[Left]* decorated with the *Lane[Left]Free* constraint tries to circumvent the vehicle ahead by changing the lane to the left if possible. The *Lane[Left]Free* constraint checks if the left lane has adequate space for a lane change. If the left lane is blocked and a lane change is not possible, the same behavior is executed for the right lane. As the last fallback in which it is neither possible to circumvent the vehicle ahead on the right nor to the left side, the last behavior *CarFollowing* is executed. This behavior adapts the speed to the vehicle ahead and even leads to a full emergency brake if necessary.

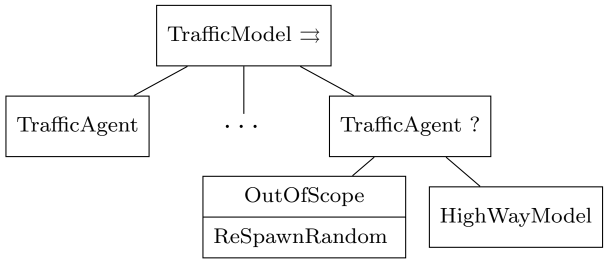


Figure 4.7: The *TrafficModel* behavior using the *HighWayModel*.

4.3.2 Traffic Model Behavior

The open structure of behavior trees allows to integrate behaviors and subtrees into greater trees. This characteristic is utilized in the following by reusing the previous *HighWayModel* in a traffic behavior. The implementation generates dynamic traffic within a certain radius around a vehicle under test. Hereby each traffic participant drives and interacts autonomously with its environment. This is modeled with the composite *TrafficModel*, which combines multiple *TrafficAgent* behaviors in parallel. The traffic density can be varied by the number of *TrafficAgent* instances. This setup allows to generate a generic number of vehicles which always drive within a certain radius around the reference vehicle. A *TrafficAgent* behavior consists of two fallback behaviors. The first child behavior checks with an *OutOfScope* constraint decorator if the traffic agent is not yet spawned or if its distance to the vehicle under test exceeds a predefined horizon distance. In such a case, the *ReSpawnRandom* behavior removes the current object in the environment if necessary and re-spawns it at another location with new random vehicle parameter settings. In situations in which the traffic agent is within the defined horizon of interest, the previously introduced *HighWayModel* behavior steers the vehicle autonomously.

4.3.3 Combination of Models and Concrete Scenarios

Besides the integration of models within other models, it is also possible to combine models with concrete scenarios. This is possible as everything is modeled as behavior trees. Next, one possible combination of the traffic model and the Euro NCAP Cut-in scenario is demonstrated as an example. This combined behavior allows to create less artificial test situations by performing the critical cut-in during random traffic situations. To enable these more complex and unpredictable critical situations, all traffic agents perform the critical cut-in if possible, in normal traffic situations. This requires to

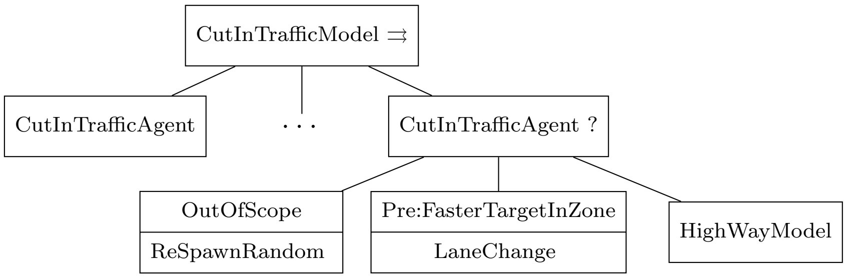


Figure 4.8: The combined *CutInTrafficModel* behavior.

extend the existing *TrafficAgent* to the *CutInTrafficAgent* behavior. All previous sub behaviors are taken over as is, however the behavior sub tree with the critical cut-in of the Euro NCAP scenario is placed between the re-spawn behavior and before the *HighWayModel* behavior. The overall *CutInTrafficAgent* behavior then leads to the same traffic behavior as before. However due to the higher prioritized critical cut-in behavior, each traffic participant performs the cut-in if the reference vehicle is behind on the adjacent lane with faster speed. This requires the system under test to monitor all traffic participants as they all can perform a critical cut-in.

4.4 Experiments

The presented scenario and the models are used in the following to demonstrate search-based testing approaches. First, an experiment with parameter variation for exploitative search is performed. Secondly, an exploratory search is shown with the help of the traffic model.

4.4.1 Exploitative Search with the Euro NCAP Cut-in Scenario

For the following experiment the Euro NCAP Cut-in Scenario of section 4.2.1 is used as the base scenario for scenario parameter variation. The initial start environment is instantiated on a straight three-way highway with a speed limit of $100km/h$. Hereby the vehicle under test is placed on the middle lane and the challenging vehicle $200m$ in front on the right lane. The dynamic behavior of the test setup is shown in Fig. 4.9. Below the *MockupSimulator* the *CriticalDistanceObserver* decorator observes continuously the distance of both vehicles. The *ScenarioBehavior* runs hereby the *NCapCutIn* and the *SystemUnderTest* behavior in parallel. This *SystemUnderTest* selector

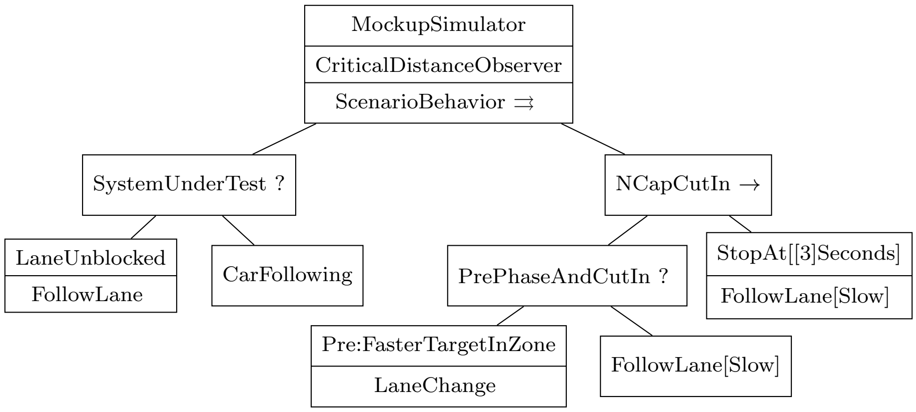


Figure 4.9: The test setup behavior for the exploratory search.

is a simplified lane keeping assistance mockup, which holds the lane and decelerates for objects in front.

The two-dimensional parameter space to exploit is defined by two parameters of the challenging vehicle. First, the speed of the vehicle is set from 30 to $45km/h$ in steps of $1km/h$. The second variation parameter is the distance at which the critical cut-in is started. It is decreased from $105m$ to $10m$ in $5m$ steps. The observed test criterion is hereby the smallest distance between both vehicles, further also called critical distance. In the plot in Fig. 4.10 the exploited parameter space is presented. The parameter with stronger impact on the critical distance is the cut-in distance. The shorter the distance is, the less time remains for the system under test to react on the other vehicle. With cut-in distances with less than $60m$ this leads to collisions between both vehicles. However, if the cut-in distance gets too small, the cut-in happens too slow as the much faster vehicle under test already passes the challenging vehicle. The increase of the second parameter leads to a shift of this collision pattern towards smaller cut-in distances. The reduced delta speed allows a timeous reaction even at smaller cut-in distances.

4.4.2 Exploratory Search with the Traffic Model

In the following experiment an exploratory search with random traffic is performed, which produces natural traffic situations. Such traffic can be used to search for critical situations, which were not defined explicitly.

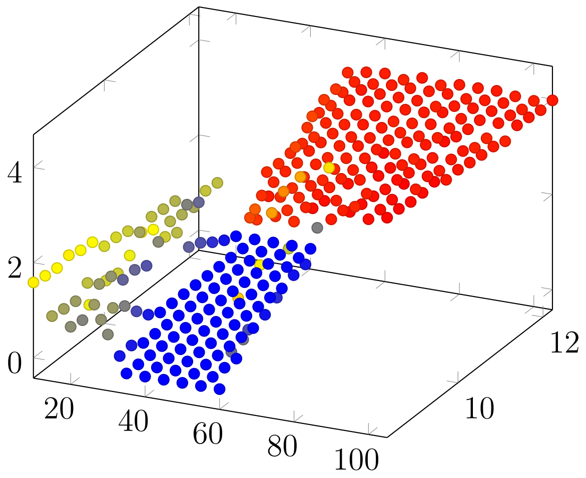


Figure 4.10: Critical distance map of eploitative search with the Euro NCAP scenario.

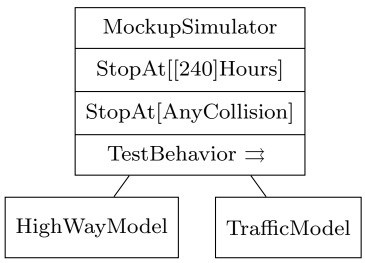


Figure 4.11: Exploratory test setup with random traffic.

Test setup

The behavior tree of this test setup is pictured in Fig. 4.11. Hereby an instance of the *HighWayModel* is used to mock an autonomous vehicle under test. Around this vehicle under test the *TrafficModel* creates in parallel random traffic. Thus, the driver models drive in this test setup against themselves. The end of an individual simulation run is defined either by a collision or a timeout. The decorator *StopAt[AnyCollision]* stops as soon as any of the vehicles are involved in a collision. If no collision occurs after ten simulated days (240 simulated hours), the simulation is stopped with the timeout decorator *StopAt[[240]Hours]*.

The driver models are parametrized as follows: The default driver model tries to keep the advisory lane speed and accelerates and decelerates with $5m/s^2$. Furthermore, it tries to hold a time to collision distance of 2 seconds to vehicles in front. These default model parameters are used for the system under test. Contrary to that, the parameters of the traffic participants within the traffic model are randomized each time they are (re-)spawned. Hereby the default parameters such as the desired lane speed as well as the acceleration and deceleration are varied with a unique value within a bound of ±25%. This unique randomization creates diverse and fluid traffic situations with more and less aggressive traffic participants.

To cover more representative traffic situations, the traffic experiment is tested with different traffic environments. First, the advisory lane speed is changed between 50, 70, and $100 km/h$. Secondly, the traffic agent density setting is changed between 10, 12, 14, or 16 vehicles in total within a $100m$ radius of interest around the vehicle under test. This traffic is instantiated on a straight three-way highway map with infinite length and without any junctions. For an expressive evaluation with averaged results, each parameter combination is simulated 100 times with a new random seed.

Evaluation of the traffic and driver model

All 1200 simulations were executed in parallel on a high-performance consumer PC with 20 CPU cores within less than a week. The results of the evaluation are available for further research [60]. Over all simulations the driven distance of the vehicle under test accumulates to more than 8.8 million simulated traffic kilometers with an accumulated driving duration of 15.8 simulated years. In average, the vehicle under test drove 7358 kilometers within 115.7 simulated hours per simulation. During these simulations, 12.6 vehicles were simulated in average simultaneously. All traffic participants (including the vehicle under test) combined drove a distance of 110.8 million kilometers with an accumulated driving duration of 199.2 simulated years in total. From all 1200 simulations, 361 simulations were stopped due to the maximum duration of 10 simulated days and did not contain collisions. The other 839 simulation runs were aborted because of a collision. Since the last seconds before a collision were recorded from the perspective of one of the collision partners, it is possible to analyze these crashes. During analysis it was conspicuous that nearly all the crashes were related to a spawning problem. In these cases, a vehicle close to the radius of interest border around the vehicle under test performs a lane change towards an empty lane. Exactly in this moment a vehicle is (re-)spawned at this lane and the other vehicle crashes into it. With an automatic analysis the collision cause of the majority with at least 468 out of 839 collisions can be definitely traced back to problems within the spawning zone. Hereby it was checked if at least one frame of the recorded GIF is fully black, as this indicates that the vehicle does not exist at this frame and is in the process to a (re-)spawned. Not all of these collision types can be traced back with this simplistic automatic analysis, as this analysis works only if the collision was recorded from the perspective of the (re-)spawned vehicle. In cases in which it was recorded from the lane changing vehicle perspective, there is sometimes no black frame and therefore it could not be counted in. However, it is assumed that nearly all collisions are related to this collision type. When taking all 839 collisions into account, the average driver model drove 132 thousand kilometers without collisions. If the clearly associated spawning crashes are abducted, the average distance increases to 298 thousand kilometers.

This spawning problem is the biggest influencer on the collision behavior. Figure 4.12 and 4.13 illustrate the driven distance and duration of the vehicle under test over the parameter variation. For lower lane speeds (50 and $70km/h$) and low target traffic densities (10-14 traffic agents) a trivial correlation can be observed. With denser traffic and higher speeds, the risk of collisions increases and therefore the driven distance and duration decreases. However, at a certain point the maximum traffic density for the given lane speed and radius of interest is reached and no more additional traffic agents can be spawned, as all spawning positions are blocked. This leads to less (re-)spawning. The effect is visible in Fig. 4.14. For the lowest lane speed with $50km/h$ the actual traffic density has a strong correlation with the target agent density. This is also true for the sample set with $70km/h$, however the last sample point with a target agent density of 16 traffic agents has already a significant deviation to the actual density with 14 traffic agents. In the sample set of $100km/h$ the significant deviation can be already observed at lower target densities. Furthermore, this sample set converges to maximum density of 14 traffic participants within the $100m$ radius around the vehicle under test. In such dense traffic situations, all traffic participants converge to a uniform common speed while leaving no room for lane changes or re-spawning. Thus, the traffic participants drive in a laminar flow with less chaotic situations. This laminar traffic flow and especially the reduced spawning activity has a major impact on the collision probability. The correlation of the maximum density and reduced collision probability can be seen in Fig. 4.12 and 4.13. The parameter set with a lane speed of $70km/h$ deviates from the previously linear curve at the sample set in which a deviation between actual and target density can be measured. In the parameter set with lane speed $100km/h$ the average collision free distance and duration is at the beginning very low but converges towards the maximal possible duration/distance. For the last parameter setting ($70km/h$, 16 agents) all 100 simulations drove collision free until the 10 simulated days were reached.

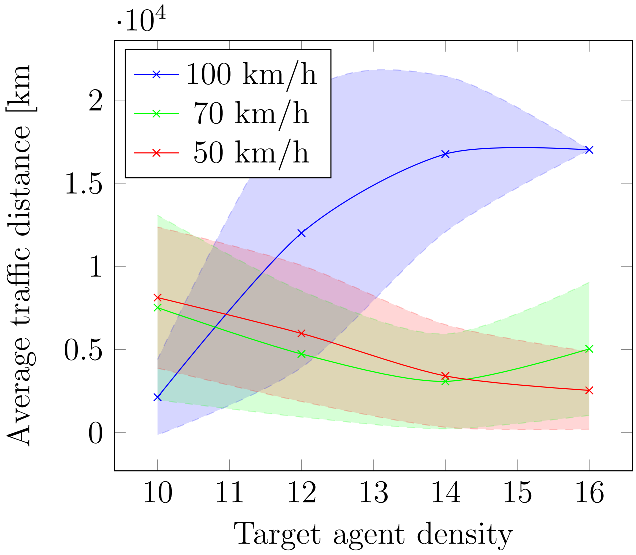


Figure 4.12: Average traffic distance with confidence intervals.

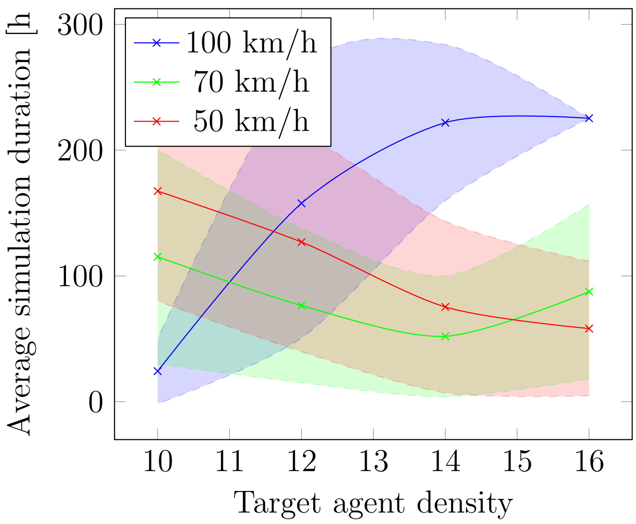


Figure 4.13: Average simulation duration with confidence intervals.

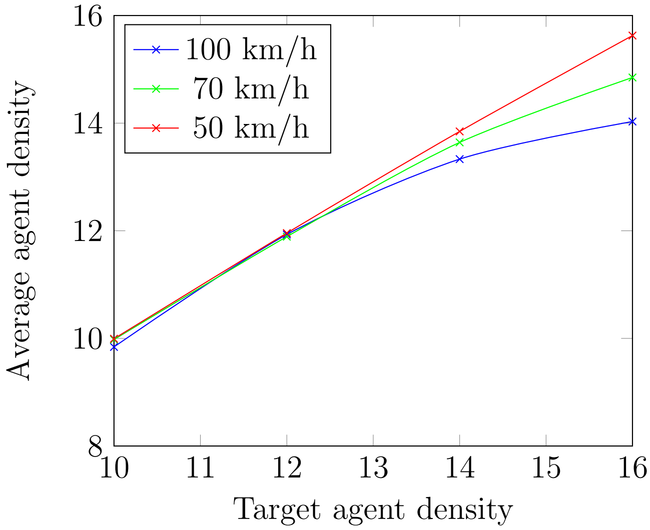


Figure 4.14: Actual traffic participant density during simulation run.

4.5 Conclusion

In this chapter it was shown how a back end based on the Yase middle end behavior tree can be created and how all elementary use cases of holistic simulation can be performed. It was proven on the one side, that the deduced requirement of agnostic development of simulator back ends 1.4.1 is possible with the provided Yase middle end. On the other side it was shown with a concrete scenario as well as with exploitative and exploratory searches that the requirement of simulation of all relevant use cases is possible 1.4.1.

First, the back end with several nodes were build based on the simple *MockupSimulator* which served as a basis for the following behaviors and experiments. Based on the supported behavior set of the back end, it was demonstrated how it is possible to create complex scenarios out of these simplest behavior nodes. First, the implementation of a concrete scenario with its integration in the simulator was highlighted. It was also shown how test criteria can be integrated in multiple ways. Secondly, an implementation of a driver and traffic model was illustrated. Hereby the great extensibility characteristics were demonstrated by reusing the driver model behavior in the traffic model behavior. In the last section it was shown with two experiments how exploitative and exploratory searches can be performed. Hereby also the test setups were fully implemented as behavior trees. Especially the large scaled traffic simulation with more than 8 million driven kilometers showed that efficient driver models can be build up with basic behaviors.

Chapter 5

Front End - OpenSCENARIO DSL Compiler

In this chapter the implementation of a language specific front end for the Yase framework is presented for the state-of-the-art scenario description language OpenSCENARIO DSL. First, the language and its characteristics, features and problems are introduced. Secondly, the instantiation of behavior trees out of the scenario description with a transpiler is shown for a representative feature subset. Here, several language features are explained, and implementation strategies are discussed.

5.1 The OpenSCENARIO DSL Language

The OpenSCENARIO DSL scenario description language, which was released in the year 2022 [16], is a domain specific language designed for intelligent vehicle testing. While it is the successor version of OpenSCENARIO XML, there is no continuity between both versions since it is a complete new and differently designed language. The main use case is still scenario execution in simulation, but it is also intended to be used for other use cases such as test plan descriptions and as an input for scenario searches in recorded databases. The language is designed consequently as a programming language, especially with elements of aspect-oriented programming. It uses well known programming principles, which allow to compose scenario behaviors into greater scenarios. With this approach it is possible to write concrete as well as high-level abstract scenarios. Furthermore, it comes with programming language typical features such as a type system with type safety, inheritance, casting, operators, functions expressions and more. While it is per definition a declarative language, it also provides an imperative set of action behaviors. This domain model consists of a basic set of primitive and compound types, actors, and action behaviors, which can be used as building blocks to model scenarios. With introduced extension mechanisms, the domain model can be extended to fit for more use cases. However, with the multitude of new features there comes the drawback of an increased implementation complexity in comparison to the previous version. Furthermore, the declarative description allows to define under constrained scenarios, which leaves much interpretation to the simulator. This can lead on the one side to different results and on the other side it requires the simulator to find valid solutions for constraint-satisfaction problems. However, such constraint-satisfaction problems are in general NP-complete [61]. Therefore, the focus of this work is on the imperative subset of the domain model.

5.2 Implementation

The OpenSCENARIO DSL front end is designed as described in chapter 2. It is independent of any simulator and has only the middle end as dependency. The front end is hereby responsible to compile a scenario file format into an executable behavior tree. This makes it a two folded problem of scenario language specific file parsing and secondly the translation into a behavior tree.

File Parsing With Transpilers

The compilation of the scenario files is done with the help of a transpiler, also sometimes called compiler-compiler. In this concrete implementation the ANTLR (ANother Tool for Language Recognition) tooling is used [62]. The transpiler parses input files with the help of a context-free grammar in the Extended Backus–Naur Form (EBNF) [35] and generates an Abstract Syntax Tree (AST) out of it. First, the input files are parsed with a syntactical lexer, which splits up the incoming file into a stream of tokens. This token stream is then evaluated against the grammar if everything is used in the right context. From there on an Abstract Syntax Tree is build up, which can be traversed later with visitors. Since the EBNF grammar of OpenSCENARIO DSL language is defined in the standard, the lexer, parser and an abstract visitor base class can be auto generated with the help of ANTLR. Figure 5.1 shows a simplified extract of the official grammar. As discussed in the previous section, OpenSCENARIO DSL requires many features to be implemented. In the following, only the most important topics regarding scenario behavior representation are addressed.

```
# Defines start of the scenario behavior
do_directive: 'do' do_member;

# Behavior definition
do_member: composition | behavior_invocation;

# General composition
composition: composition_operator '():'
                NEWLINE INDENT
                do_member+
                DEDENT;

# Composite behavior
composition_operator: 'serial' | 'parallel';

# Behavior invocation
behavior_invocation: [actor_expression '.']
                        behavior_name
                        '(' [argument_list] ')';
```

Figure 5.1: Simplified extract of the OpenSCENARIO DSL grammar with aspects of behavior description.

Visitor Implementation For Behavior Tree Generation

With the ANTLR auto generated lexer and parser, any valid scenario file can be translated into an Abstract Syntax Tree. For the further translation into a behavior tree, the AST must be traversed via visitors. ANTLR also autogenerates an abstract visitor class, which already contains for each grammar statement a visitor method. For the compiler front end, a project specific visitor is inherited from the auto generated base visitor. Here, each overwritten visitor method is responsible to parse the given input, translate it into the corresponding behavior and return the result. For some of the grammar statements in the given grammar extract 5.1 visitor pseudo code is shown in 7.2, which shows how the behavior tree is assembled. While the example concerns only with the behavior tree instantiation, the actual visitor class is far more complex with more than 80 visitors since it has to handle much more features. The behavior definition of an OpenSCENARIO DSL scenario starts with a *do_directive* statement, which is translated with the *visitDoDirective* visitor method dependent on the given situation. This visitor invokes the *do_member* statement with the *visitDoMember* visitor, which passes itself the call to either the *visitComposition* or the *visitBehaviorInvocation* visitor. Within the visitor method *visitComposition*, a composite node with its children is build up and returned. It first calls the underlying visitor *visitCompositionOperator*, which returns either a sequence or a parallel composite node. At this composite node all traversed child nodes are appended. Since the child nodes are instantiated with the *visitDoMember*, the result can be again a full subtree. Once the visitor methods are traversed down to a leaf action node, the *visitBehaviorInvocation* is called. This visitor must return the corresponding action node. However, the visitor can only rely on the dependencies to the middle end with the Agnostic Behavior Tree and the Agnostic Type System. The front end does not have any information of the actual supported action behavior set of the simulator, since this varies for each back endd. However, the Agnostic Type System provides a *SimulatorSupport* class, which allows to retrieve a behavior node declaration for a behavior name. With this, the visitor class can search with the parsed action behavior name for the declaration and return the instantiated behavior node. This way the visitor can be implemented completely independent of any backend, since it is decoupled via the *SimulatorSupport* class. In final usage, the front end is instantiated and the supported action behaviors of the specific back ends are then passed in via dependency injection with the *SimulatorSupport* class. The fact, that action behaviors can be also instantiated with the help of actors is ignored in this pseudo code, but the principles are the same.

For the scenario in Figure 5.2 the translation process of the dynamic behavior part is shown exemplarily. The translation for the actor instantiation is ignored here as well as the aspect of input argument parsing. First, the scenario file is translated with the lexer and parser into the Abstract Syntax Tree shown in Figure 5.3. This tree is then traversed by the implemented visitors, which build up step by step the whole behavior tree presented in Figure 5.2. With this process any OpenSCENARIO DSL scenario behavior can be translated into a behavior tree.

Actor handling

Previously it was shown how the dynamic scenario behavior part is translated into a behavior tree. However, OpenSCENARIO DSL uses extensively the concept of actors. In the previous example 5.2, the scenario started with such a definition of two vehicle actors. An actor is an entity, which can perform assigned behaviors but has also default behaviors to execute if nothing is assigned. If no behavior influences the actor, the vehicle actor may act as a normal vehicle. It can follow the lane, adapt the speed, or overtake slower traffic participants. If an actor specific action is active, the degree of freedom of the specific actor is restricted. As an example, the vehicle actor

```
# Definition of an example scenario
scenario simple_example_scenario:
  actor_1: vehicle
  actor_2: vehicle

  do parallel():
        serial():
            actor_1.follow_lane(duration: 1s)
            actor_1.change_lane(lane_change_side::left)
        actor_2.follow_lane()

```

Figure 5.2: Simple OpenSCENARIO DSL scenario example with two vehicle actors.

can perform an action such as $follow\_lane$. This action restricts the lateral control of the vehicle to follow the lane. However, in such a case the vehicle is not restricted regarding its longitudinal control, which allows the vehicle to still adapt the speed to the lane speed limit or to obstacles ahead. This could be constrained further by executing in parallel behaviors such as *keep_speed* or *change_acceleration*. The OpenSCENARIO DSL standard advises to use actor bounded actions over standalone behavior nodes. Like the compound data type *struct*, it is also possible to add data members as well as methods to the actor. Furthermore, inheritance can be used to extend the scope of actors. The definition and handling of such actors during the file compilation phase is handled with the help of the Agnostic Type System.

During the scenario execution, the vehicle actor default behavior, and the handling of incoming (potentially conflicting) action behavior requests must be handled explicitly in the behavior tree. For this, the actor instances are placed within an *ActorMaintainer* decorator node. If in the scenario file an actor is instantiated, then this decorator is placed on top of the scenario behavior tree to ensure that the actor can either perform requested tasks or its default behavior. The realization can be seen in the generated behavior tree in Fig. 5.2, where the actors are maintained in the top *ActorMaintainer* node. The action behavior nodes request specific behavior tasks from the actors. If the requested task is valid, the actor executes at the end all behavior tasks in a consistent manner and fills definition gaps with its default behavior. However, in case of inconsistencies, the actor reports this back to the requesting behavior node, which can then in succession return a Failure as execution status. This can be the case if for example one behavior node requests a $follow\_lane$, while another node request a *change_lane* in the same execution tick.

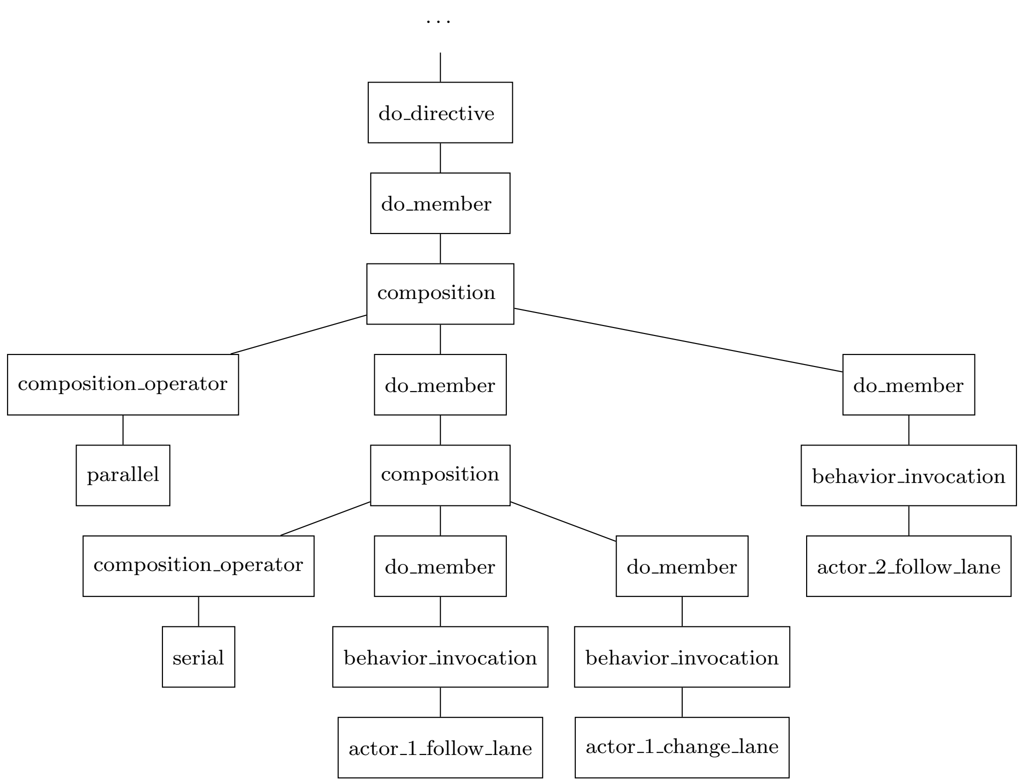


Figure 5.3: Generated Abstract Syntax Tree (AST) of the dynamic behavior part of scenario 5.2.

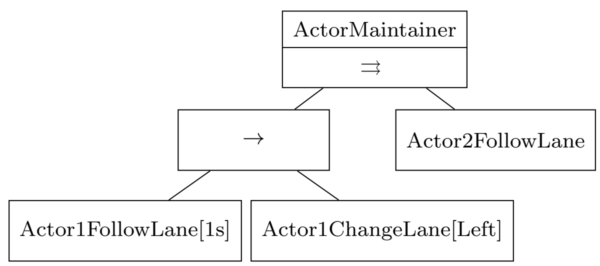


Figure 5.4: Generated behavior tree of scenario 5.2.

Abstraction Via Scenario Sub Trees

One crucial feature of OpenSCENARIO DSL is the modularization of scenarios by writing abstract and reusable sub scenarios. Scenarios can be instantiated within greater scenarios to develop more complex behaviors. In the Fig. 5.5 an OpenSCENARIO DSL example is given, which imports the previous scenario 5.2 and instantiates it two times. The behavior modeling is trivial since behavior trees always allow to append other behavior trees within a behavior tree. In the given example, at the places of the imported scenario invocation a new instance of the sub scenario tree is generated and attached to the tree. The only difference is here the instantiation with the different input arguments. Implementation-wise, the actor maintenance must be handled differently for an imported scenario behavior. Actors, which are passed via an argument to the sub scenario are already maintained by an *ActorMaintainer* at a higher tree position. Due to that fact, it is checked while scenario instantiation if an actor is passed as argument and already exists, or it instantiated at this level for the first time. Only if the actor is not passed in and therefore not yet maintained, the actor is placed in a new *ActorMaintainer* node at this level. This effect can be seen in the instantiated behavior tree in Fig. 5.8. Instead of having *ActorMaintainer* nodes in each sub scenario, the actor maintenance is done once for both actors in the root node.

```
import ./simple_example_scenario.osc

# Sequence of two different imported scenario instances
scenario extended_scenario:
  actor_1: vehicle
  actor_2: vehicle

  do serial():
        simple_example_scenario(actor_1, actor_2)
        simple_example_scenario(actor_2, actor_1)

```

Figure 5.5: Example in which the imported scenario 5.2 is instantiated with different actor assignment.

Modifiers

The OpenSCENARIO DSL standard allows to extend actor behaviors with modifiers. Such modifiers can constraint a behavior even further during its

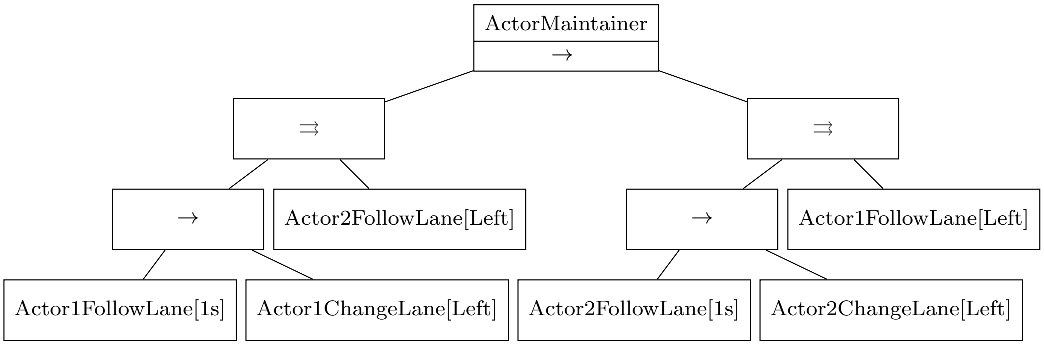


Figure 5.6: Generated behavior tree of scenario 5.5.

lifetime. In Fig. 5.7 a very simple scenario is shown, in which the $follow\_lane$ behavior is extended with a speed modifier. This modifier restricts the speed of the actor between 10 and 20 meter per seconds during the lifetime of the $follow\_lane$ behavior. This can be modeled with decorators, which constraints the actor during the lifetime of the behavior. While in OpenSCENARIO DSL these modifiers are written below the behavior to modify after a *with* statement due to syntactical reasons, the modifiers are placed on top of the behavior node to modify. This allows to extend behaviors such as the $follow\_lane$ node, which are leaf nodes. The modifier decorator passes the execution signal through to the child behavior. As long as this child behavior is running, the modifier task such as the speed restriction is delegated to the actor. With this approach it is also possible to cascade multiple modifiers for the same behavior.

```
scenario scenario_with_modifier:
  actor_1: vehicle

  do actor_1.follow_lane(duration: 1s) with:
        speed(10m_s..20m_s)

```

Figure 5.7: Simple scenario with one behavior and a modifier.

5.3 Discussion

In this chapter it was shown that a complex and state-of-the-art scenario language can be translated with the help of the Yase middle end into an

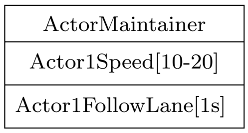


Figure 5.8: Generated behavior tree of scenario 5.7 with a modifier.

executable behavior tree. Hereby the language with its characteristics and features, but also its difficulties was elaborated. First the usage of transpilers was explained, which help to convert any given input file into an Abstract Syntax Tree. From there on it was shown how project specific visitors are used to translate such an AST into an executable behavior tree. Based on this, OpenSCENARIO DSL specific implementation details such as actor handling, scenario abstraction and the usage of modifiers were elaborated further. With this chapter it was proven that the OpenSCENARIO DSL language can be translated into the Yase middle end with behavior trees as their representation. As this was already shown for the previous version OpenSCENARIO XML in section 3.1.4, the first deduced requirement 1.4.1 with agnostic development of multiple front ends is fulfilled.

Chapter 6

Conclusion and Outlook

For intelligent vehicle development and validation, simulation is a crucial element to sufficiently cover the test domain space with reasonable effort. With this thesis, several contributions were provided to bring the state-of-the-art research in intelligent vehicle development towards more holistic and incremental simulation. In the following, the content of this thesis with its most important contributions are summarized. Subsequent, the accomplishments are evaluated regarding the deduced requirements, quality goals, and constraints. Last, an outlook on the open-source project is given and further research topics are presented, which are either already under research or potential future research fields.

6.1 Contributions

This thesis provides multiple contributions to handle the increasing complexity with respect to the occurring problems in industry scaled intelligent vehicle development. The contributions can be summarized into three main categories. First, a state-of-the-art research with respect to practical problems in the industry was given. Secondly, solutions for these problems were presented with the Yase concept. And last, the sustainability and applicability were proven with (open sourced) implementations. As a result, the Yase framework with its three-stage approach was made available, which decouples the development of scenario language compilers from the actual integration to a specific simulator or its sub scopes. This concept allows to build highly customized simulators out of sub scopes in an incremental way for each test level down to the lowest level of unit tests.

The first main contribution is an extensive research about the state of the art of intelligent vehicle simulation and the current problems under the aspect of practical industry development:

- **Elaboration of current simulation use cases, solutions and limitations.** An extensive elaboration of the current situation in the broad field of intelligent vehicle simulation was presented. Here, the current problems arising when developing more and more complex intelligent vehicle systems were discussed. From this perspective, existing simulation approaches were presented. Furthermore, the problem with the inconsistent standardization landscape in this domain field was presented.

- **Deduction of current requirements.** Based on this analysis, three main requirements were deduced: The need for incremental simulation for all test levels. The importance of agnostic development of input format compilers and simulator bindings to reduce redundant work. And the necessity to be able to simulate concrete scenarios, as well as to perform exploitative and explorative searches of scenario spaces.

- **Presentation of a standardized development approach.** For the deduced problem statement, it was exemplarily shown how such a problem can be approached systematically with state-of-the-art software development standards. With the help of the arc42 reference guide, development quality goals and constraints were derived. The quality goals were deduced from the software product quality standard ISO/IEC 25010.

- **Exhaustive analysis of control architectures in the intelligent vehicle simulation domain.** A specialized analysis of existing simulators and their underlying control architectures was contributed. Additionally, the most important control architectures were presented and evaluated regarding the derived quality goals. To the knowledge of the author, such an extensive analysis of control architectures was never done in this detail for the given domain field. Furthermore, a deeper analysis of behavior trees with their common implementation strategies and projects was presented.

This theoretical groundwork with the analysis of the state of the art, current solutions and architectures, as well as the derived requirements were the basis for new developed concepts. Here, the thesis contributed two new important concepts to the state-of-the-art research:

- **New Yase concept with three-stage approach.** Development of the Yase framework for holistic and incremental simulation with a computer language compiler orientated three-stage approach. To the knowledge of the author, this concept has never been transfered into the context of intelligent vehicle simulation. The presented approach allows to develop scenario language front ends independent of the concrete simulator back ends. This enables improved collaboration across companies and a simplified split of simulator monoliths into smaller simulator sub scopes.

- **Domain specific Agnostic Behavior Tree (ABT) concept.** For the central middle end, an improved version of a behavior tree implementation was contributed. This so called Agnostic Behavior Tree is designed to fulfill the needs of the intelligent vehicle simulation domain. A main improvement of this concept in comparison to existing implementations is the development of the new node state *kSuccessRunning*, which enables more simulation specific use cases.

Besides the new introduced concepts, the thesis contributed with several implementations to prove the practical feasibility. The specific contributions are here as follows:

- **Demonstration of a simulator back end implementation.** With this implementation it was shown how a scenario engine can be based on the introduced Agnostic Behavior Tree. The implementation includes examples of scenarios, driver models, and traffic models. Further implementation strategies were discussed, such as the design of a full simulator with behavior trees as well as implementations of constraints. To the authors knowledge, no such holistic approach with behavior trees exists in this extent by now.

- **Demonstration of a compiler front end implementation.** For this, an implementation for the most complex and state-of-the-art scenario language standard OpenSCENARIO DSL was presented. Here it was shown how a front end is able to compile such scenarios into a generic behavior tree. Furthermore, a discussion on how to implement certain aspects of the scenario standard was contributed. To the authors knowledge, it is the first concrete and detailed discussion of an implementation for this standard.

- **Experiments to prove applicability.** With the presented implementations, several experiments with exploitative and exploratory scenario space searches were performed. From special interest is hereby the massive traffic simulation, with more than 8 million driven kilometers. This traffic simulation results were contributed for further research.

- **Open sourced implementation.** The last contribution is the open sourced implementation. The Yase code was published with an open license within the OpenPASS project. The active usage and integration in further projects underlines the practical usability of the presented concept.

6.2 Evaluation of Contributions

The presented contributions have been proven by experiments and in practice. Furthermore, a list of quality goals and technical constraints was derived next to the defined goal within the problem statement in section 1.4. The fulfillment of such is assessed in the following:

Req. 1: Agnostic development of input format compilers and simulator bindings

The foundation for independent development of scenario language compilers and simulator back ends was set within chapter 2. Here, the three-stage approach with input language specific front ends, an interface middle end, and simulator back ends was presented. In the subsequent chapter 3, with behavior trees a powerful controller architecture was identified as intermediate scenario representation. Based on this, the possibility of agnostic development of back ends (chapter 4) and front ends (chapter 5) was proven with exemplary implementations.

Req. 2: Incremental simulation for all test levels

The Agnostic Behavior Tree (ABT) presented in chapter 3.4 as the interface between front ends and back ends is the essential building block on which everything is based on. This library was designed as a pure C++ library without any dependencies. Due to the avoidance of any network and interprocess communication the library is inherently unit test ready. Therefore, the Yase middle end allows to build up simulations, which can run on all test levels - from unit tests up to full system tests.

Req. 3: Ability to perform concrete, exploitative and exploratory searches

With the back end implementation and the experiments in chapter 4.4 it was proven how it is possible with a simple set of behavior tree nodes to simulate all relevant intelligent vehicle simulation use cases. First, the concrete Euro

NCAP Cut-in scenario was used as the basis to exploit a two dimensional scenario space 4.4.1. Furthermore, an exploration of a larger domain space with a massive traffic simulation with over 15.8 simulated years was shown in section 4.4.2.

Quality Goals

The quality goals are mostly relevant for the middle end implementation since the middle end serves as the central interface between the interchangeable front and back ends. For this reason, multiple controller architectures were analyzed and compared against the derived quality goals 3.1.6. Here the behavior tree approach was evaluated as the best fitting architecture, which caused the Yase implementation of the Agnostic Behavior Tree (ABT) presented in chapter 3.4.

Constraint 1: Industry Adequate C++ Code

Like the quality goals, this constraint affects again mainly the middle end as the central element. The implementation of the C++ Agnostic Behavior Tree was done with the help of modern software practice. This includes the usage of consistent style guides and formatting, but also the survey with Clang and GCC compiler checks. Furthermore, the library is tested extensively with unit tests.

Constraint 2: Simplicity

The design constraint of a straightforward architecture was met by the reduction of overly complex and self-developed concepts. The architecture relies mostly on known and proven concepts. This helps to lower the entry level for new users. Each design choice resembles existing well-known concepts and follows the most canonical way when possible. For this, the general Yase design was resembled after computer language compiler design with their terminology presented in chapter 2. Furthermore, the central middle end is based on the widely used concept of behavior trees. For both concepts, extensive background information is available.

6.3 Outlook

Besides the presented implementations of the Yase framework, the Yase concept already serves as the basis for further development. Furthermore, additional research fields can be investigated further with the help of the presented Yase contribution. In the following, the most important ones are listed.

Further Development in the Open Source Community

During this thesis, the Yase code was released with an open-source license. It exists under the umbrella of the OpenPASS project as an independent repository with its own project structure. Based on this work, contributions were made by other companies, such as the integration of multiple build systems. Furthermore, Yase was integrated in the main OpenPASS project for the OpenSCENARIO XML scenario engine [63]. Additional to that, the Yase concept is developed actively within Bosch as a full simulation engine and lead to a rich ecosystem. This company internal work contributes to the open source community with ongoing release of features and improvements. Namely the Agnostic Type System package within the middle end and the OpenSCENARIO DSL front end are in complexity and in terms of lines of code at least comparable to the Agnostic Behavior Tree package, if not more complex (See lines of code analysis in Fig. 7.3, 7.4, 7.5).

Scenario Monitoring

For safety argumentation it is often necessary to prove coverage of certain test domain spaces. For intelligent vehicle validation this can be done by proving that a certain amount of scenarios are tested, both in a simulated environment and in real-world driving. Besides simulation of defined scenario sets, this also requires to search for existence of scenarios in a given data base. Such a data base can be either filled with recordings of exploratory simulations or of real drives. This use case is also mentioned as a valid use case of OpenSCENARIO DSL. In a sense, such scenario searches can be interpreted as reverted scenario execution. The scenario to search can be as well implemented as behavior tree. Instead of executing a behavior on simulation data, the action nodes in this tree can then observe if the desired behavior occurs in the environment data. Since such action search nodes can be separated into a self contained back end, it is possible to use the Yase concept for simulation and scenario monitoring in parallel with maximum code reuse. This topic is currently under research and may be published in future.

Test Plan Description

With OpenSCENARIO DSL the scenario language format was designed to enable full test plan descriptions. Despite the declarative description of the scenario behavior, it also contains descriptions for evaluation and coverage. This requires to extend the functionality of executing only one simulation per scenario file to executing a full test campaign with a multitude of scenarios. For this, the simulator itself must be capable to run multiple scenarios. This may include more sophisticated approaches such as the distribution of individual scenarios to different simulation workers. A further research area is the execution of declarative scenarios. Since the constraint satisfaction problem can not be solved directly, one could use randomization to generate a multitude of scenarios to find a solution in a brute-force manner. After the execution of each scenario, tooling such as the described scenario monitoring could be used to search for a fitting scenario in the set of simulated scenarios. For the scenario generation, simple brute force approaches such as random traffic could be used. Also more sophisticated approaches with constraints solvers can be developed, taking the given constraints into account.

Chapter 7

Appendix

This chapter contains all additional material for a better understanding of this thesis.

7.1 ISO/IEC 25010 Quality Goals

In the following, the definitions of the three relevant quality characteristics are copied from the ISO standard 25010 [27]:

Functional Suitability

This characteristic represents the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions. This characteristic is composed of the following sub-characteristics:

- Functional completeness - Degree to which the set of functions covers all the specified tasks and user objectives.
- Functional correctness - Degree to which a product or system provides the correct results with the needed degree of precision.
- Functional appropriateness - Degree to which the functions facilitate the accomplishment of specified tasks and objectives.

Usability

Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. This characteristic is composed of the following sub-characteristics:

- Appropriateness recognizability - Degree to which users can recognize whether a product or system is appropriate for their needs.

- Learnability - Degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use.

- Operability - Degree to which a product or system has attributes that make it easy to operate and control.

- User error protection. Degree to which a system protects users against making errors.

- User interface aesthetics - Degree to which a user interface enables pleasing and satisfying interaction for the user.

- Accessibility - Degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.

Maintainability

This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements. This characteristic is composed of the following sub-characteristics:

- Modularity - Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.

- Reusability - Degree to which an asset can be used in more than one system, or in building other assets.

- Analysability - Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.

- Modifiability - Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- Testability - Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

7.2 Further Material

```cpp
// Example node
class ChangeLaneNode : public ActionNode {
  public:
   // Lookup data for this node
   void lookupAndRegisterData(Blackboard& blackboard) {
      m_map = blackboard.get<Map>("unique_map_key");
   };

   // Inits/ resets the node state before first tick
   void onInit() override {
      m_vehicle.attach();
      m_start_lane = m_map->getCurrentLane();
   };

   // Logic of the node
   NodeStatus tick() {
      // In Failure case, stop without altering state
      if(!hasAdjacentLane()) {
         return NodeStatus::kFailure;
      }
      // In Success case, stop without altering state
      if(reachedAdjacentLane()) {
         return NodeStatus::kSuccess;
      }
      // Perform behavior step
      steerVehicleTowardsAdjacentLane();
      return NodeStatus::kRunning;
   };

   // Terminates behavior after it finishes
   virtual void onTerminate() override {
      m_vehicle.detach();
   }

  private:
   bool hasAdjacentLane() const;
   bool reachedAdjacentLane() const;
   void steerVehicleTowardsAdjacentLane();

   Map* m_map{};
   VehicleActor* m_vehicle{};
   Lane m_start_lane{};
  };
```

Figure 7.1: Simplified C++ example of a lane change node.

```
// Yase vistitor implementation of the generated abstract visitor
class Osc2YaseVisitor : public OscParserVisitor {
 public:
   Osc2YaseVisitor(std::unique_ptr<const SimulatorSupport> simulator_support)
       : OscParserVisitor(),
         m_simulator_support(std::move(simulator_support))

     Any visitComposition(CompositionContext* ctx) override {
       yase::CompositeNode::Ptr composite = visitCompositionOperator(ctx->compositionOperator())
       for (auto child : ctx->children) {
           composite.push_back(visitScenarioNode(child));
         }
       return composite;
     }

     Any visitCompositionOperator(CompositionOperatorContext* ctx) override {
       if(ctx->serial != nullptr) {
           return std::make_shared<yase::SequenceNode>();
       } else {
           return std::make_shared<yase::ParallelNode>();
       }
     }

     Any visitBehaviorInvocation(BehaviorInvocationContext* ctx) override {
       return m_simulator_support->getFromScope<Declaration<yase::BehaviorNode::Ptr>>(ctx->behavior_name)
           .instantiate(input_arguments);
     }
 private:
   // Supported cope of simulator
   std::unique_ptr<const SimulatorSupport> m_simulator_support;
};
```

Figure 7.2: Pseudo code of a visitor implementation.

```
% cloc ./agnostic_behavior_tree
      95 text files.
      92 unique files.
       3 files ignored.

github.com/AlDanial/cloc v 1.98  T=0.03 s (3146.8 files/s, 290873.5 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                             57           1066           1220           3820
C/C++ Header                    31            497            639            892
Markdown                         2             50              0            139
CMake                            1             21             16            130
Starlark                         1              2              0             12
-------------------------------------------------------------------------------
SUM:                            92           1636           1875           4993
-------------------------------------------------------------------------------
```

Figure 7.3: Code line count of the Agnostic Behavior Tree package (with the cloc tool).

```
% cloc ./agnostic_behavior_tree
      95 text files.
      92 unique files.
       3 files ignored.

github.com/AlDanial/cloc v 1.98  T=0.03 s (3146.8 files/s, 290873.5 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                             57           1066           1220           3820
C/C++ Header                    31            497            639            892
Markdown                         2             50              0            139
CMake                            1             21             16            130
Starlark                         1              2              0             12
-------------------------------------------------------------------------------
SUM:                            92           1636           1875           4993
-------------------------------------------------------------------------------
```

Figure 7.4: Code line count of the Agnostic Type System package (with the cloc tool).

```
% cloc ./agnostic_behavior_tree
      95 text files.
      92 unique files.
       3 files ignored.

github.com/AlDanial/cloc v 1.98  T=0.03 s (3146.8 files/s, 290873.5 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                             57           1066           1220           3820
C/C++ Header                    31            497            639            892
Markdown                         2             50              0            139
CMake                            1             21             16            130
Starlark                         1              2              0             12
-------------------------------------------------------------------------------
SUM:                            92           1636           1875           4993
-------------------------------------------------------------------------------
```

Figure 7.5: Code line count of the OpenSCENARIO DSL front end package (with the cloc tool).

Bibliography

[1] A. Ziebinski, R. Cupek, D. Grzechca, and L. Chruszczyk, “Review of advanced driver assistance systems (adas),” in *AIP Conference Proceedings*, vol. 1906, no. 1. AIP Publishing, 2017.

[2] R. Okuda, Y. Kajiwara, and K. Terashima, “A survey of technical trend of adas and autonomous driving,” in *Technical papers of 2014 international symposium on VLSI design, automation and test*. IEEE, 2014, pp. 1–4.

[3] N. Kalra and S. M. Paddock, “Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?” *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, 2016.

[4] H. Winner, W. Wachenfeld, and P. Junietz, “Validation and introduction of automated driving,” *Automotive Systems Engineering II*, pp. 177–196, 2018.

[5] “Article - tesla autopilot crash 2016,” https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-death-self-driving-car-elon-musk, accessed: 2023-05-22.

[6] B. E. T. Automation and J. Jain, “Learn api testing.”

[7] A. Ngo, M. P. Bauer, and M. Resch, “A sensitivity analysis approach for evaluating a radar simulation for virtual testing of autonomous driving functions,” in *2020 5th Asia-Pacific Conference on Intelligent Robot Systems (ACIRS)*. IEEE, 2020, pp. 122–128.

[8] ——, “Deep evaluation metric: Learning to evaluate simulated radar point clouds for virtual testing of autonomous driving,” in *2021 IEEE Radar Conference (RadarConf21)*. IEEE, 2021, pp. 1–6.

[9] ——, “A multi-layered approach for measuring the simulation-to-reality gap of radar perception for autonomous driving,” in *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2021, pp. 4008–4014.

[10] “Waymo safety report 2021,” https://downloads.ctfassets.net/sv23gofxcuiz/4gZ7ZUxd4SRj1D1W6z3rpR/2ea16814cdb42f9e8eb34cae4f30b35d/2021-03-waymo-safety-report.pdf, accessed: 2023-05-22.

[11] “Euro ncap - aeb car-to-car system test protocol 2023,” https://cdn.euroncap.com/media/75439/euro-ncap-aeb-c2c-test-protocol-v411.pdf, accessed: 2023-05-22.

[12] “United nations - automated lane keeping system approval,” https://documents-dds-ny.un.org/doc/UNDOC/GEN/G20/087/82/PDF/G2008782.pdf?OpenElement, accessed: 2023-05-22.

[13] “Website - asam,” https://www.asam.net/, accessed: 2023-05-22.

[14] J. Ma, X. Che, Y. Li, and E. M.-K. Lai, “Traffic scenarios for automated vehicle testing: A review of description languages and systems,” *Machines*, vol. 9, no. 12, p. 342, 2021.

[15] “Website - asam openscenario xml,” https://www.asam.net/standards/detail/openscenario/#, accessed: 2023-01-30.

[16] “Website - asam openscenario dsl,” https://www.asam.net/standards/detail/openscenario/v200/, accessed: 2023-01-30.

[17] “Website - vires vtd,” https://vires.mscsoftware.com/, accessed: 2023-05-22.

[18] “Website - ipg carmaker,” https://ipg-automotive.com/en/products-solutions/software/carmaker/, accessed: 2023-05-22.

[19] “Carla simulator,” https://github.com/carla-simulator/carla, accessed: 2023-01-30.

[20] “Website - openpass simulator,” https://projects.eclipse.org/projects/automotive.openpass, accessed: 2023-05-22.

[21] “Website - environment simulator minimalistic (esmini),” https://github.com/esmini/esmini, accessed: 2023-05-22.

[22] "Website - sumo simulator," https://www.eclipse.org/sumo/, accessed: 2023-05-22.

[23] "Website - cloe framework," https://github.com/eclipse/cloe, accessed: 2023-05-22.

[24] "Website - eclipse mosaic framework," https://www.eclipse.org/mosaic/, accessed: 2023-05-22.

[25] D. Salles, L. Lang, M. Kehrer, and H.-C. Reuss, "A modular co-simulation framework with open source software and automotive standards," in *22. Internationales Stuttgarter Symposium: Automobil-und Motorentechnik*. Springer, 2022, pp. 207–223.

[26] "Website - arc42 architecture template," https://arc42.org/overview, accessed: 2023-05-22.

[27] "Website - iso/iec 25000: Systems and software engineering — systems and software quality requirements and evaluation (square)," https://www.iso.org/standard/64764.html, accessed: 2023-01-30.

[28] "Website - iso 26262," https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en, accessed: 2023-05-22.

[29] "Website - yase repository," https://gitlab.eclipse.org/eclipse/openpass/yase, accessed: 2023-05-22.

[30] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques and tools*, 2020.

[31] "Website - llvm compiler," https://llvm.org/, accessed: 2023-05-22.

[32] "Website - gcc compiler," https://gcc.gnu.org/, accessed: 2023-05-22.

[33] "Website - llvm clang compiler," https://clang.llvm.org/, accessed: 2024-01-20.

[34] "Website - rust compiler," https://rustc-dev-guide.rust-lang.org/overview.html, accessed: 2023-05-22.

[35] "Iso/iec 14977:1996 standard for extended bnf grammar definition," accessed: 2023-04-24.

[36] J. Wang, *Formal methods in computer science.* CRC Press, 2019.

[37] S. M. Yacoub and H. H. Ammar, "Finite state machine patterns." in *EuroPLoP*, 1998, pp. 401–428.

[38] V. Sklyarov, "Hierarchical finite-state machines and their use for digital control," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 2, pp. 222–228, 1999.

[39] M. Risler, *Behavior control for single and multiple autonomous agents based on hierarchical finite state machines.* tuprints, 2009.

[40] A. Kurt and Ü. Özgüner, "Hierarchical finite state machines for autonomous mobile systems," *Control Engineering Practice*, vol. 21, no. 2, pp. 184–194, 2013.

[41] S. B. Kotsiantis, "Decision trees: a recent overview," *Artificial Intelligence Review*, vol. 39, pp. 261–283, 2013.

[42] "Website - trigger cloe engine," https://cloe.readthedocs.io/en/latest/reference/triggers.html, accessed: 2023-01-30.

[43] H. Chen, H. Ren, R. Li, G. Yang, and S. Ma, "Generating autonomous driving test scenarios based on openscenario," in *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 2022, pp. 650–658.

[44] "Environment simulator minimalistic (esmini) - scenario code," https://github.com/esmini/esmini/blob/master/EnvironmentSimulator/Modules/ScenarioEngine/SourceFiles/Story.hpp, accessed: 2023-01-30.

[45] "Openpass simulator - old scenario engine," https://gitlab.eclipse.org/eclipse/simopenpass/openscenario1_engine/-/tree/main/existing_open_scenario_engine/src/Storyboard, accessed: 2023-01-30.

[46] D. Isla, "Managing complexity in the halo 2 ai system." Game Developers Conference, 2005.

[47] M. Colledanchise and P. Ogren, *Behavior Trees in Robotics and AI: An Introduction*, 07 2018.

[48] "Unreal engine - behavior trees," https://docs.unrealengine.com/5.0/en-US/behavior-trees-in-unreal-engine/, accessed: 2023-01-30.

[49] "Website - unity behavior designer asset," https://assetstore.unity.com/packages/tools/visual-scripting/behavior-designer-behavior-trees-for-everyone-15277, accessed: 2023-01-30.

[50] L. Ruifeng, W. Jiasheng, Z. Haolong, and T. Mengfan, "Research progress and application of behavior tree technology," in *2019 6th International Conference on Behavioral, Economic and Socio-Cultural Computing (BESC)*. IEEE, 2019, pp. 1–4.

[51] E. W. Dijkstra, *Letters to the editor: go to statement considered harmful*. Communications of the ACM, Volume 11, 03 1968.

[52] S. Rabin, *Game AI pro: collected wisdom of game AI professionals*. CRC Press, 2013.

[53] "Website - behaviortree.cpp," https://www.behaviortree.dev, accessed: 2023-01-30.

[54] "Game ai pro: Behavior tree starter kit repository," https://github.com/aigamedev/btsk, accessed: 2023-01-30.

[55] "Carla simulator - scenariorunner," https://github.com/carla-simulator/scenario_runner, accessed: 2023-01-30.

[56] "Website - py-trees - a python based behavior tree implementation," https://py-trees.readthedocs.io/en/devel/introduction.html, accessed: 2023-01-30.

[57] "Article - return early pattern," https://medium.com/swlh/return-early-pattern-3d18a41bba8, accessed: 2023-01-30.

[58] M. P. Bauer, A. Ngo, and M. Resch, "The yase framework: Holistic scenario modeling with behavior trees," in *2021 IEEE 94th Vehicular Technology Conference (VTC2021-Fall)*. IEEE, 2021, pp. 1–7.

[59] ——, "Usage of operating system kernel paradigms to build modern simulation platforms for autonomous driving," in *2020 5th Asia-Pacific Conference on Intelligent Robot Systems (ACIRS)*. IEEE, 2020, pp. 80–85.

[60] "Github repo - yase highway traffic experiment results," https://github.com/MaxPBauer/YaseHighwayExperiment, accessed: 2023-05-22.

[61] M. Y. Vardi, "Constraint satisfaction and database theory: a tutorial," in *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2000, pp. 76–85.

[62] "ANTLR: Another tool for language recognition," accessed: 2023-04-24.

[63] "Openpass simulator - openscenario1 engine with yase," https://gitlab.eclipse.org/eclipse/openpass/openscenario1_engine, accessed: 2023-01-30.