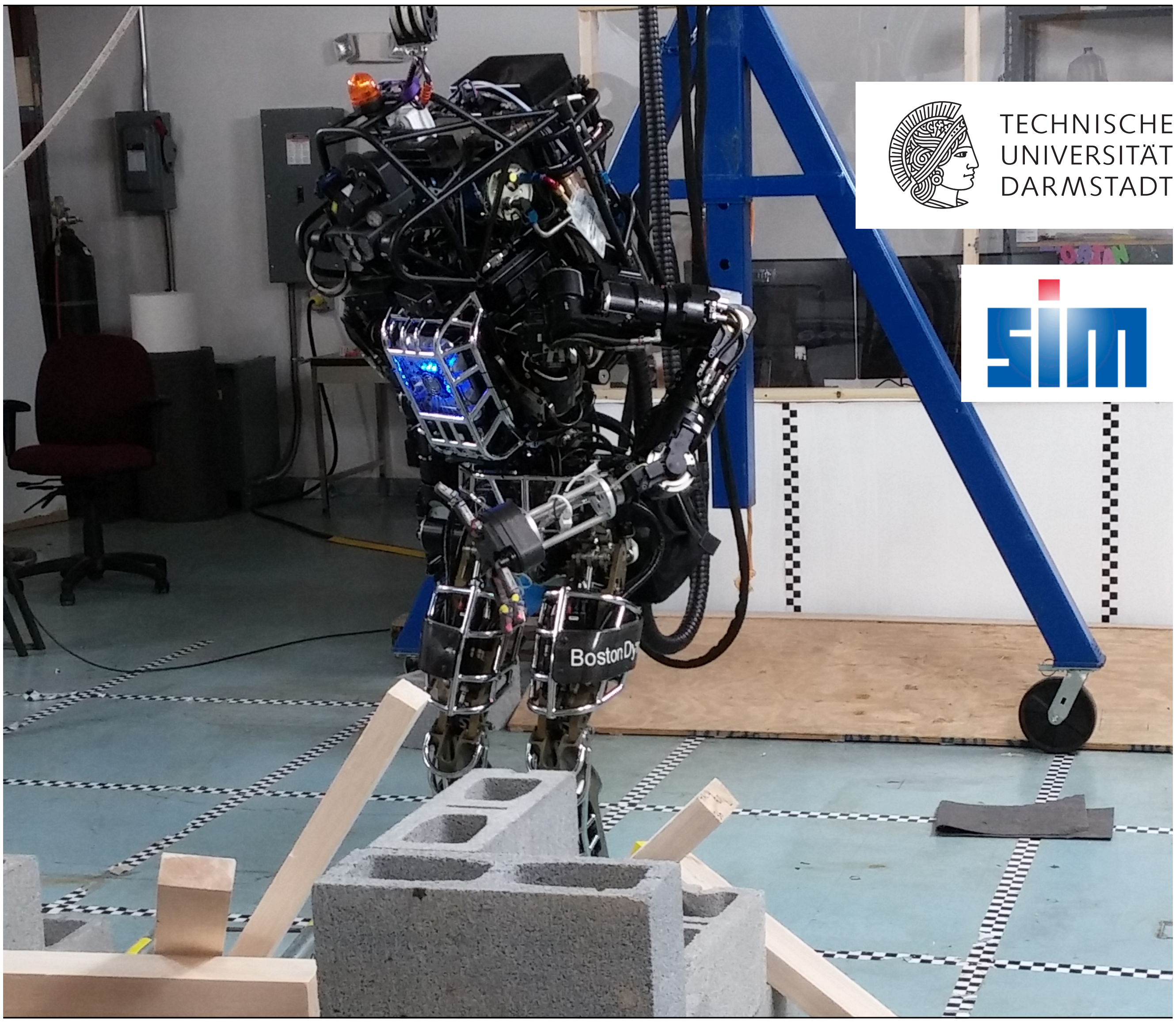
An Approach for Runtime-Modifiable Behavior Control of Humanoid Rescue Robots

Ein Verfahren zur Steuerung humanoider Rettungsroboter durch zur Laufzeit modifizierbare Verhalten

Master-Thesis von Philipp Schillinger

Tag der Einreichung:

1. Gutachten: Prof. Dr. Oskar von Stryk
2. Gutachten: Dipl.-Inform. Stefan Kohlbrecher



TECHNISCHE UNIVERSITÄT DARMSTADT

An Approach for Runtime- Modifiable Behavior Control of Humanoid Rescue Robots
Ein Verfahren zur Steuerung humanoider Rettungsroboter durch zur Laufzeit modifizierbare Verhalten

Vorgelegte Master-Thesis von Philipp Schillinger

1. Gutachten: Prof. Dr. Oskar von Stryk
2. Gutachten: Dipl.-Inform. Stefan Kohlbrecher

Tag der Einreichung:

Technische Universität Darmstadt
Fachbereich Informatik

Fachgebiet Simulation, Systemoptimierung und Robotik (SIM)
Prof. Dr. Oskar von Stryk

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 14. April 2015

(Philipp Schillinger)

Abstract

This thesis describes a novel approach for modification of robot behaviors during runtime. Existing high-level robot control has very limited adaptability regarding unexpected disturbances. Possible uncertainties have to be known and explicitly considered in advance by defining strategies of how to react to these. This requirement confines the development of robust robots as for example required in complex and unstructured disaster mitigation scenarios. In order to overcome this limitation and to facilitate the development of more flexible high-level robot behavior control, the approach developed in this thesis enables to change the whole structure of behaviors even while they are executed. Therefore, the operator is able to incorporate situational knowledge gained during execution, and thus to compensate insufficient a priori knowledge about the status of the environment. The approach is proposed based on modeling behaviors as hierarchical state machines, allowing for modular composition and intuitive specification in different levels of abstraction. Detailed monitoring of the state of execution and occurred errors assists the operator when giving commands and adjusting the level of autonomy, utilizing the individual capabilities of both robot and operator in a cooperative manner. Nevertheless, the developed framework is able to cope with severe restrictions on the communication channel to the robot and is robust regarding runtime failure. In addition, verification of specified behaviors greatly reduces the risk of failure. As part of this thesis, the behavior engine *FlexBE* has been developed in order to implement and refine the promoted concepts. It comes with a comprehensive user interface and behavior editor and is practically applied in the upcoming *DARPA Robotics Challenge Finals*.

Zusammenfassung

Die vorliegende Arbeit beschreibt einen neuartigen Ansatz zur Anpassung des Roboterverhaltens während der Laufzeit. Aktuelle Verfahren zur Steuerung von Robotern sind nur sehr eingeschränkt in der Lage, auf unerwartete Störungen zu reagieren. Mögliche Unsicherheiten und Strategien, um diese zu überwinden, müssen im Voraus bekannt sein und explizit berücksichtigt werden. Diese Anforderung schränkt die Entwicklung robuster Roboter, wie sie beispielsweise in komplexen und unstrukturierten Krisenszenarien erforderlich sind, deutlich ein. Um diese Einschränkung zu überwinden und die Entwicklung flexiblerer Verhaltenssteuerung von Robotern zu erleichtern, ermöglicht es der in dieser Arbeit entwickelte Ansatz, die komplette Struktur von Verhalten zu ändern, selbst während diese bereits ausgeführt werden. Das erlaubt es dem Bediener, zur Laufzeit gesammeltes Wissen über die Situation des Roboters einzubringen und so unzureichendes Ausgangswissen auszugleichen. Der verfolgte Ansatz basiert auf der Wahl von hierarchischen Zustandsautomaten zur Modellierung von Verhalten, was eine modulare und intuitive Definition in verschiedenen Abstraktionsschichten erlaubt. Detaillierte Informationen über den aktuellen Laufzeitstatus und aufgetretene Fehler unterstützen den Bediener dabei, Befehle zu senden und den Autonomiegrad des Roboters anzupassen. Dadurch können die individuellen Fähigkeiten von Roboter und Bediener für eine kooperative Herangehensweise genutzt werden. Dennoch ist das System in der Lage, mit starken Beschränkungen hinsichtlich der Kommunikation mit dem Roboter umzugehen und robust auf Laufzeitfehler zu reagieren. Das Risiko solcher Fehler wird zudem durch die Verifizierung von Verhaltensdefinitionen deutlich reduziert. Als Teil dieser Arbeit wurde auch die Verhaltensengine *FlexBE* entwickelt, um die erarbeiteten Konzepte umzusetzen und zu verbessern. *FlexBE* beinhaltet eine umfangreiche Benutzeroberfläche sowie einen Verhaltenseditor und wird im bevorstehenden Finale der *DARPA Robotics Challenge* eingesetzt.

Acknowledgements

In the course of this thesis, many people provided support, contributed with helpful discussions about the concept, and practically used the implemented framework *FlexBE* already in early stages of development. First and foremost, I would like to thank my advisors Prof. Dr. Oskar von Stryk and Stefan Kohlbrecher, not only for their feedback and suggestions regarding this thesis, but also for being co-authors of the submitted research paper presenting some of the concepts developed as part of this thesis.

Furthermore, I thank Spyros Maniatopoulos and the *Verifiable Robotics Research Group* lead by Prof. Hadas Kress-Gazit for their hospitality during my stays at Cornell University, and especially Spyros Maniatopoulos for the joint work on combining the developed framework with behavior synthesis and for extensively using *FlexBE*, helping to knock out all those little hidden flaws in early versions.

Moreover, thank goes to David Conner and the DRC project team of TORC Robotics for giving us a solid infrastructure for software development and doing their best to provide a working robot during all of the three test phases at Blacksburg, despite the technical difficulties introduced by switching to the new version of *ATLAS*.

Additionally, I would like to thank all participants of the regular behavior control meetings for the helpful and leading discussions regarding design choices and required features. Also, I thank Moritz Schappler, Jonathan Vorndamme, and the rest of their team at the *Institute of Automatic Control* at Leibniz University Hanover for extensively using *FlexBE* for their system identification tests.

Finally, I would like to thank all members of *Team ViGIR* providing functionality for the implementation of behavior states, especially Stefan Kohlbrecher, Alberto Romay, and Alexander Stumpf at *SIM*, TU Darmstadt, and all members developing and assisting development of behaviors based on *FlexBE* and relying on the previously mentioned states.

Intelligence is the ability to adapt to change.

— Stephen Hawking

Contents

1 Introduction **1**

1.1 Motivation . . . 4

1.2 Contributions . . . 5

1.3 FlexBE . . . 7

1.4 Thesis Overview . . . 8

2 State of Research **9**

2.1 Behavior Control Approaches . . . 9

2.1.1 Autonomous Behaviors . . . 10

2.1.2 Human Robot Interaction . . . 12

2.1.3 Cooperative Behaviors . . . 14

2.2 Behavior Verification . . . 16

2.2.1 Behavior Synthesis . . . 17

2.3 Runtime Modifications . . . 18

2.4 Existing Behavior Frameworks . . . 19

2.4.1 XABSL . . . 19

2.4.2 RoboCup Frameworks . . . 20

2.4.3 SMACH . . . 21

2.4.4 Flor Behavior Engine . . . 22

3 Concepts **24**

3.1 Definitions . . . 24

3.2 Operator Interaction . . . 27

3.2.1 Behavior Mirror . . . 28

3.2.2 Behavior Input . . . 29

3.3 Runtime Modifications . . . 30

3.3.1 Consistency . . . 31

3.3.2 Behavior Update . . . 33

3.4 Behavior Development . . . 33

3.5 State Constraints . . . 34

4 Behavior Engine **36**

4.1 Foundations . . . 37

4.1.1 Behavior Specification . . . 37

4.1.2 Operator Interaction . . . 38

4.2 Onboard Executive . . . 39

4.2.1 Inheritance Chain . . . 41

4.2.2 State Implementation . . . 45

4.3 Code Transfer . . . 46

4.4 Behavior Switch . . . 47

4.5 Monitoring . . . 49

5 Behavior Operation **51**
5.1 Approach 51
5.1.1 Infrastructure 52
5.1.2 Operator Views 55
5.2 Statemachine Editor 57
5.2.1 User Interaction 58
5.2.2 Integration of States 60
5.2.3 Code Generation 62
5.3 Runtime Control 64
5.3.1 Control Interface 67
5.3.2 Communication 70

6 Evaluation **73**
6.1 Operator Experience 73
6.1.1 Behavior Development 74
6.1.2 Behavior Execution 75
6.2 Behavior Performance 76
6.3 DARPA Robotics Challenge 79
6.3.1 ATLAS 81
6.3.2 THOR-MANG 82
6.4 Future Work 83
6.4.1 Synthesis Integration 83
6.4.2 Web-based Solution 84
6.4.3 Parallel Execution 85
6.4.4 Multiple Robots 86

7 Conclusion **87**

Bibliography **89**

Appendix **93**
A.1 List of Developed States 94
A.1.1 flexbe_states 94
A.1.2 flexbe_atlas_states 95
A.2 Evaluation of Behavior Logfiles 97
A.3 Example State Implementation 98
A.4 Exploration Demo Behavior 100
A.5 Communication Channel Topics 101
A.6 FlexBE Tutorials 102
A.6.1 Creating a New Behavior 103
A.6.2 Using the Statemachine Editor 106
A.6.3 Execution of a Behavior 110

1 Introduction

Robotics research showed significant progress during recent years. Individual robot capabilities such as grasp planning for enhanced manipulation, reliable bipedal locomotion, or advanced techniques to perceive the environment, have been significantly improved and enable applying robots to an even greater extend to tasks where they can support humans or replace them in dangerous situations. However, most of the current robot applications are very domain specific. They are tailored to specific tasks and are not arbitrarily applicable, while often based on one or two main capabilities. For example, a common service robot is focused on human robot interaction, but it is only able to navigate on flat ground. A robot equipped with a multitude of sensors might be able to detect objects of interest in its vicinity, but lacks in autonomous manipulation, and thus can only report its findings.

The assumed scenarios are typically well defined. It is known in advance what is required to solve a task, and also possible sources of failure are known in detail. Furthermore, if the problem is too complex to be precisely modeled, teleoperation of the robot agents is a common approach. Unfortunately, scenarios in rescue robotics are often much more severe. Besides typically performing on tasks of critical importance, rescue robots face much harder restrictions regarding the ability to model their expected environment and to communicate with them during task execution. While the former restriction limits the applicability of approaches relying on autonomous robot behavior, the latter prevents conventional teleoperation. In order to address such tasks a hybrid approach is required instead. The robot and its human operator need to jointly work together and support each other when possible. In addition, precise a priori knowledge of the circumstances of such tasks is rarely available, which makes it difficult to design an approach in advance.

In order to encourage research in this important field of robotics, the *United States Defense Advanced Research Projects Agency*[1] (DARPA) proposed the *DARPA Robotics Challenge*[2] (DRC).

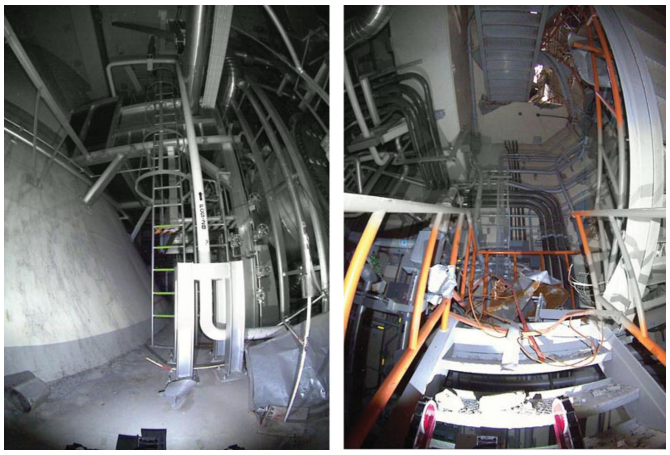


Figure 1.1: High-resolution photos taken by one of the robots at Fukushima [30].

[1] http://www.darpa.mil *(03/21/2015)*
[2] http://www.theroboticschallenge.org/ *(03/21/2015)*

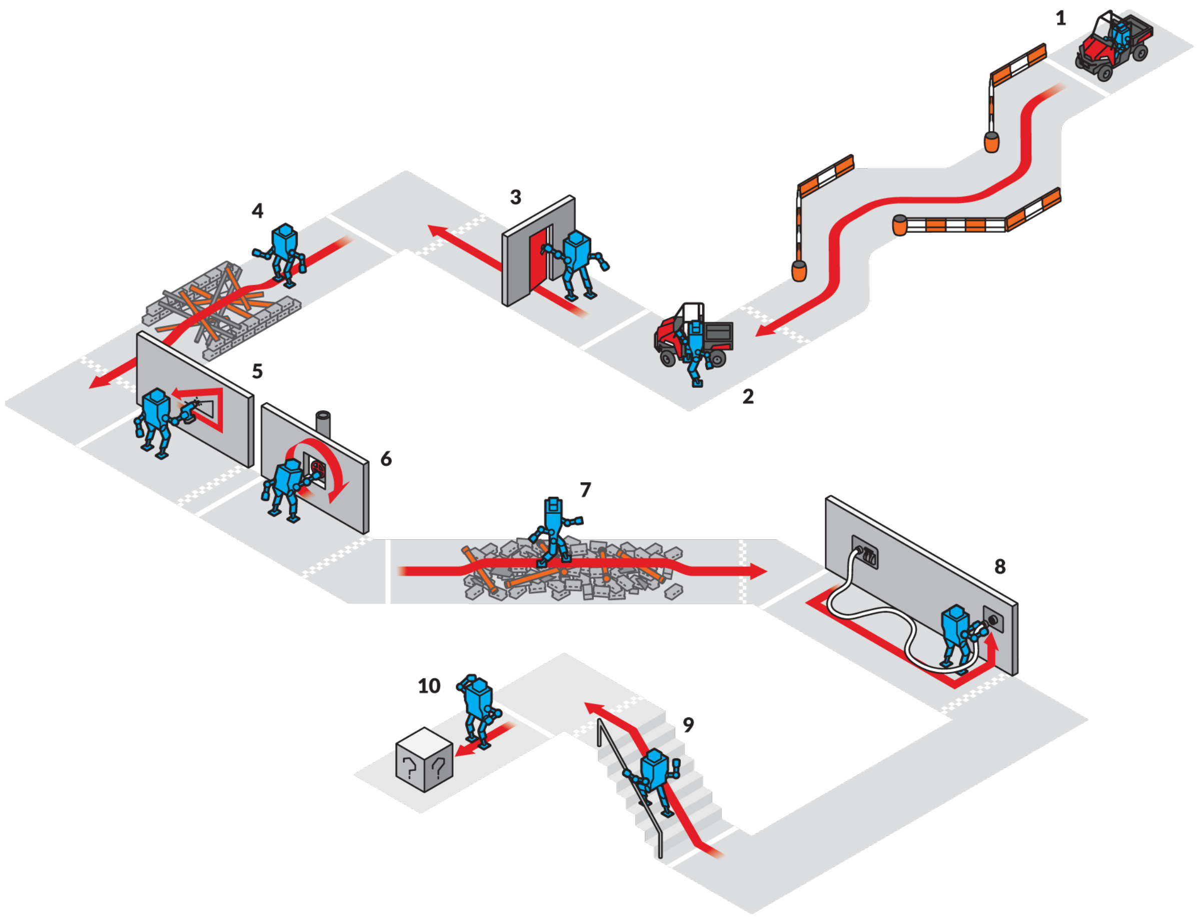


Figure 1.2: Overview of the tasks in the DRC Finals, illustrated by James Provost.

Motivated by the earthquake and tsunami which caused significant damage to the Fukushima Daiichi nuclear plant in 2011, this challenge defines a sequence of tasks required to mitigate similar disasters. At this time, robots were deployed at the destructed nuclear plant with the goal to investigate the site and to avert further danger. Despite all effort, unexpected limitations such as narrow stairs, and disturbances like debris blocking the way prevented successful execution of the conducted missions.

An overview of the tasks of the DRC Finals is given in figure 1.2. After approaching the site by driving a vehicle and egressing it, the robot has to enter the building by opening a door. Inside, the robot will face a number of different tasks such as negotiating debris, cutting a hole in a wall and opening a valve behind it, or plugging in a fire hose. Finally, the robot will have to climb up stairs and solve a previously unknown manipulation task. The whole course has to be solved in a single run with a time limit of one hour. During the task, no wired connection to the robot is allowed, neither for communication nor as power supply. As further restrictions, the wireless connection to the robot is strictly limited to incorporate the expected conditions of a disaster response scenario and to prevent teleoperation and the availability of detailed environmental data. Especially indoors, communication is barely available and may occasionally face blackouts.

Team ViGIR[3] (standing for Virginia-Germany Interdisciplinary Robotics) is one of the *Track B* teams participating at the *DARPA Robotics Challenge*. While *Track A* teams receive DARPA

[3] `http://www.teamvigir.org/` *(03/31/2015)*

funding to develop an own robot including software, *Track B* teams are funded for sole software development and are provided an *ATLAS*[4] robot developed by *Boston Dynamics*[5]. In addition, further teams can participate by relying on external funding. The challenge has been proposed in early 2012 and started in October 2012. The first competition, the *Virtual Robotics Challenge* (VRC) was held in June 2013 between the teams without an own robot in order to determine the top teams to receive an *ATLAS* robot. *Team ViGIR* was ranked 6th out of the 126 registered teams and thus qualified to continue participation at the DRC and got an *ATLAS* robot. In December 2013, the DRC Trials formed the first competition of the *DARPA Robotics Challenge* including the real robots and the *Track A* teams. *Team ViGIR* ranked 10th out of the remaining 16 teams and is now facing the DRC Finals, held in June 2015.

The team was initially formed by TORC Robotics[6], the *Simulation, Systems Optimization and Robotics Group*[7] at TU Darmstadt, and the *3D Interaction Group*[8] at Virgina Tech. After the VRC, the *Robotics and Human Control Systems Lab*[9] at Oregon State University joined the team, and on track to the DRC Finals, the *Verifiable Robotics Research Group*[10] at Cornell University and the *Institute for Automatic Control*[11] at Leibniz University Hanover joined as well. The software of *Team ViGIR* is based on ROS[12] and leverages several open-source software frameworks. The main part of the software runs on the robot while the operator interacts with the part running at the *Operator Control Station* (OCS). An overview of the system as used for the trials is given by

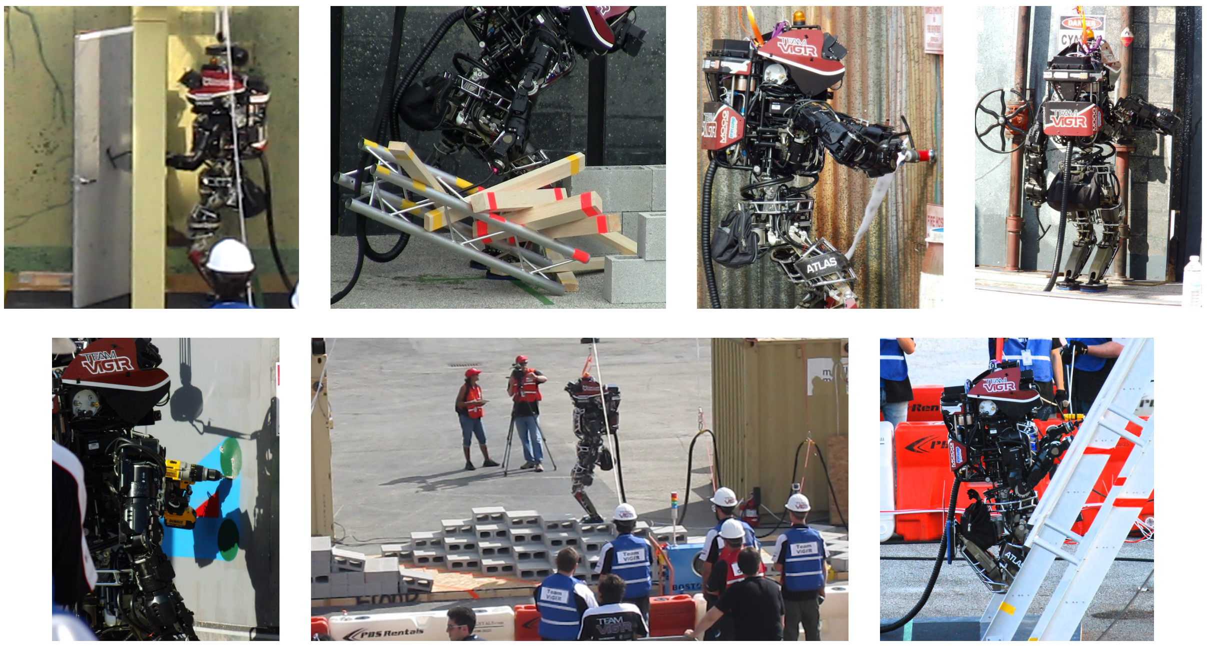


Figure 1.3: Florian, the *ATLAS* robot of *Team ViGIR*, at the DRC Trials [22].

[4] http://www.bostondynamics.com/robot_Atlas.html *(03/31/2015)*
[5] http://www.bostondynamics.com/ *(03/31/2015)*
[6] http://www.torcrobotics.com/ *(03/31/2015)*
[7] http://www.sim.informatik.tu-darmstadt.de/ *(03/31/2015)*
[8] https://research.cs.vt.edu/3di/ *(03/31/2015)*
[9] http://mime.oregonstate.edu/research/rhcs/ *(03/31/2015)*
[10] http://verifiablerobotics.com/ *(03/31/2015)*
[11] http://www.irt.uni-hannover.de/ *(03/31/2015)*
[12] http://www.ros.org/ *(03/31/2015)*

[22]. The behavior control framework developed in this thesis incorporates the team's approach towards a more operator-independent operation of the robot and to support high-level human robot cooperation when autonomous execution is not possible. Previous work already targeted this issue and provided an approach for specifying high-level behaviors of the robot, as well as a way to adjust the level of autonomy during execution. But as approaching the DRC Finals, these capabilities are not flexible enough for such complex tasks and the provided operator interaction is not sufficiently detailed. While the robot had to solve eight different tasks in separate runs with a time limit of half an hour for each run in the DRC Trials, the upcoming DRC Finals require the robot to solve ten tasks, one of them even not specified in advance, in a single run with a time limit of only one hour.

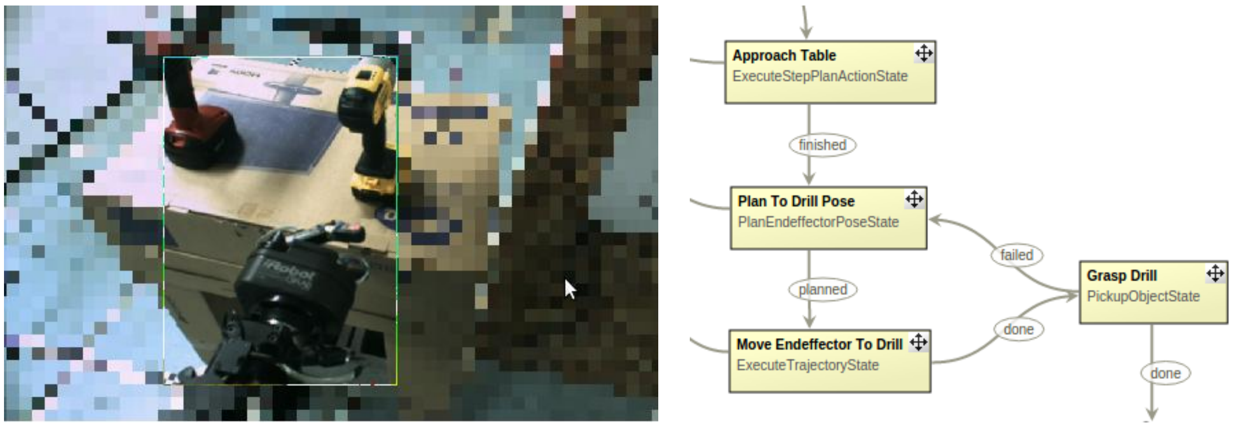
1.1 Motivation

Considering the disaster response scenario as previously described, main focus of this thesis is the development of a framework applicable on a single robot with advanced locomotion and manipulation abilities. It is assisted by at least one operator in order to solve complex tasks in an unstructured environment while facing tough restrictions regarding communication. The exact details of a task are not necessarily known in advance and unforeseen failure during task execution has to be taken into account carefully. Therefore, high adaptation of behavior control at runtime is required.

Adaptation to gained knowledge during task execution is generally possible in two different ways. On the one hand, abstracting from details and trying to detect these during runtime is a data-based way of adaptation. For example, when grasping an object, instead of commanding the robot to close its fingers by a certain value, the command can be to close the fingers until an increased force to the sensors indicates that the object is safely grasped. On the other hand, modifications to the control flow of a behavior might be required in order to successfully adapt to disturbances. Closing the fingers of the robot, for example, could be the wrong decision at this time. If the operator detects that the object to be grasped is completely different from what was expected, the grasping strategy developed and specified in advance may not be applicable.

Existing approaches only allow for limited adaptation by targeting the data-based way. Recent research, especially work in the field of assisted or supervised autonomy, starts also focusing on decision-based adaptation. However, these approaches mostly aim at preventing wrong decisions by utilizing operator interaction and reducing the autonomous capabilities of the robot. Although this is an important aspect for preventing task failure, high involvement of the operator for the rest of a task just because the scenario does not meet the previously assumed specifications close enough, is neither a robust approach, nor is it always possible.

In order to overcome this limitation and to be able to flexibly adapt to arbitrary disturbances, this thesis not only proposes a way to allow for closer cooperation between the operator and the robot in case of insufficient autonomous capabilities of the robot due to its behavior specification not matching the faced scenario. But also, the developed approach enables re-specification of the robot's behavior based on the gained situational knowledge without interrupting or aborting task execution, and without being limited in any way by previous specifications.



(a) (b)

Figure 1.4: Possible scenario where the control flow has to be adjusted: Instead of using the yellow drill as planned in advance, a more powerful drill is detected and preferred by the operator. The behavior defined in (b) does not account for choosing between drills. (a) taken from [22].

Motivated by the complex scenarios including a high level of uncertainty as faced by rescue robots, the ability to modify behaviors flexibly during runtime is a powerful feature to improve task performance and to be able to develop significantly more adaptable robots. However, changing the specifications of a behavior during its execution also makes it prone to failure by possibly introducing new errors. This is especially likely considering the high pressure situation in which the operator has to specify the changes, and his limited possibilities to test the new specifications in advance because the robot is already in the field. Thus, it is an important aspect of this thesis to reduce the sources of failure by assisting the operator in developing the behavior, and to automate as much as possible while also incorporating verification checks.

Finally, further challenges of modifying behaviors during runtime have to be addressed. It is very important to not distract the operator from supervising the actions taken by the robot. Considering that the circumstances in which a modification of the robot behavior is required typically also require careful supervision and high attention by the operator, specification of the modifications has to be as easy and intuitive as possible. In addition, choosing a multi-operator approach is very advisable for such situations. Furthermore, even small adjustments of the behavior require time. Although the scenario itself may put time constraints on task execution, modifying the behavior of the robot should not add further time pressure. For this reason, the operator needs to be able to specify a point in behavior execution where the robot will pause until the changes are applied, while keeping to execute the correct behavior as long as possible in parallel to defining the changes in order to still make progress on the task.

1.2 Contributions

As discussed in the previous section, this thesis mainly addresses the support of runtime modifications of robot behaviors. But while these adaptations offer great flexibility when considering the capability of reacting to unforeseen difficulties or imminent failure, they come along with a considerably high risk regarding reliability. When changing a behavior specification during runtime, the new version of the respective behavior is very prone to errors since it can not be tested before applying it to the robot. Furthermore, occurring execution failure may directly lead to task failure, or at least have a significantly negative impact on task performance. For this reason, the thesis also addresses assistance for the operator when specifying behaviors and takes mechanisms regarding behavior verification into consideration. Therefore, this thesis includes contributions in the following three domains:

Runtime Modifications

The realization of runtime modifications to behavior code without interrupting the execution of the current task and without having to specify certain points for just making predefined decisions represents the main contribution of this thesis. At first, several ways of how to define and execute behaviors will be discussed and evaluated regarding their capabilities to incorporate runtime changes. Subsequently, one approach is selected and a concept for the realization of changes is developed. Finally, the implementation of this concept is presented and evaluated in scenarios motivated by tasks of the *DARPA Robotics Challenge*.

Human Robot Cooperation

In order to enable the specification of behavior changes and to leverage the operator's capability of reacting to unexpected situations, this thesis discusses ways for the realization of cooperation between the robot and its operator. Requirements for a suitable user interface are derived and their realization is demonstrated. Based on the special conditions imposed by the situation in which behavior modifications have to be specified, as well as on the required complexity of behaviors in order to solve the faced tasks, an innovative modeling tool is developed. This not only allows to automatically generate executable behavior source code, but also to monitor and control behaviors during their execution.

Behavior Verification

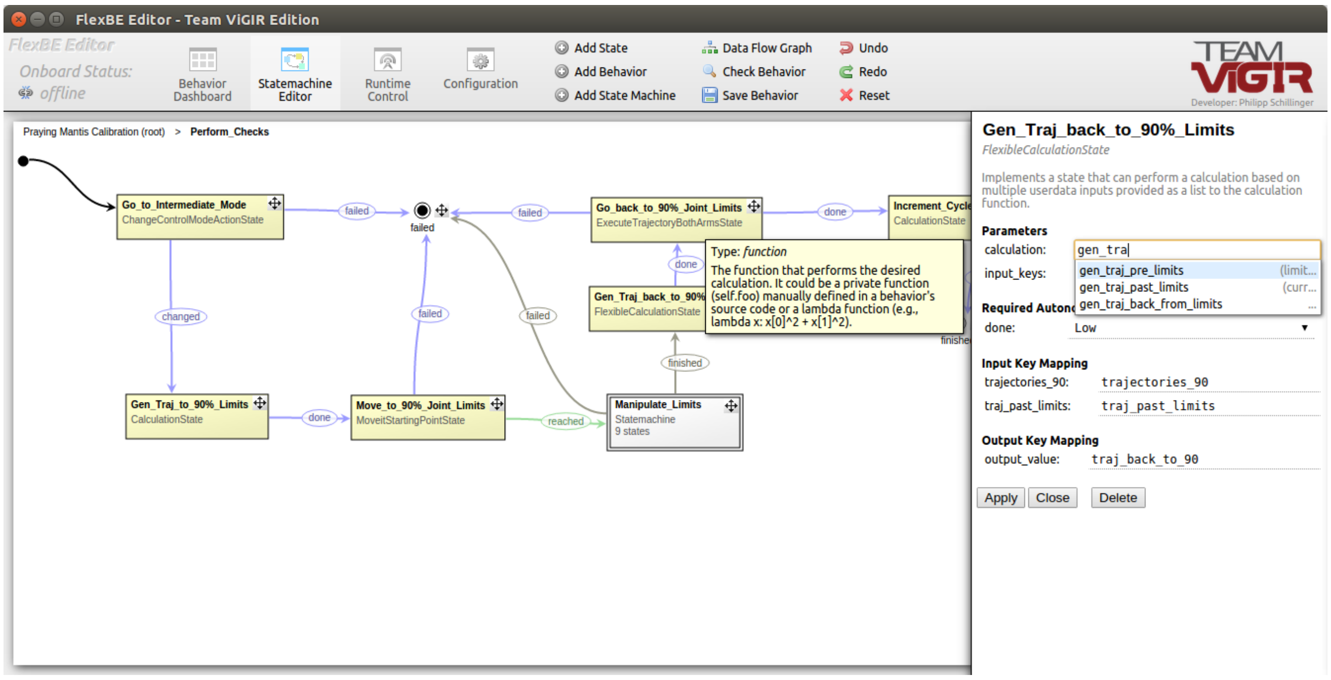
Considering the high reliability required for behavior specifications due to the missing possibility of performing extensive testing during an ongoing mission, this thesis investigates ways to efficiently and robustly verify complex behaviors. This not only refers to structural consistency, but also to the dataflow implied by the structure. Furthermore, ways to define custom conditions for actions triggered by the behavior and integrating behavior synthesis for provably correct behaviors will be discussed, though not yet provided by the implemented behavior framework, in order to form a basis for future extensions.

Although originated from the *DARPA Robotics Challenge*, this behavior framework is applicable in a variety of scenarios under different circumstances. Systems can benefit a lot from the possibility to flexibly combine individual capabilities in order to form high-level task behaviors. The tight integration of a human operator into the decision process of the robot enables even application in complex, unstructured, or unknown environments. Especially applications in the field of rescue robotics, but also similar disciplines such as space robotics or deep-sea exploration, take advantage of the tolerance to impeded communication as well as of the ability to flexibly react to unforeseen situations without needing to reset the robot.

1.3 FlexBE

As result of this thesis, the behavior engine *FlexBE* (standing for *Flexible Behavior Engine*) has been developed, based on the discussed concepts and defined requirements. It includes a user interface combining an advanced behavior editor and a behavior control station for supervising behaviors during runtime. While the engine itself is applicable for even underspecified tasks in unstructured environments, the interface is designed for use by non-expert operators and developers as well. Especially when specifying complex behaviors, even a developer familiar with robotics will typically not have detailed knowledge regarding all the individual capabilities involved for solving the task. Thus, a modular and abstract composition of behaviors is possible using *FlexBE*, with a strong focus on assisting the developer by providing relevant documentation. Even advanced behaviors can be specified and executed without manually writing a single line of code, although editing source code is possible in order to not limit expert users.

In addition to solely providing a way for specifying complex behaviors, *FlexBE* also manages their execution on a remote robot agent. It includes operator interaction for close human robot collaboration by design and deals with issues such as keeping track of the remote execution, or sending commands and monitoring their execution. Switching flexibly from the currently executed version of a behavior to a new one is possible without restarting the robot or interrupting task progress, in order to take gained knowledge regarding the task into account. Even if a behavior unexpectedly fails during its execution, the robustness of *FlexBE* allows the operator to correct it on-the-fly or to select a different behavior to be executed instead.



Type: *function*
The function that performs the desired calculation. It could be a private function (self.foo) manually defined in a behavior's source code or a lambda function (e.g., lambda x: x[0]^2 + x[1]^2).

Gen_Traj_back_to_90%_Limits
FlexibleCalculationState

Implements a state that can perform a calculation based on multiple userdata inputs provided as a list to the calculation function.

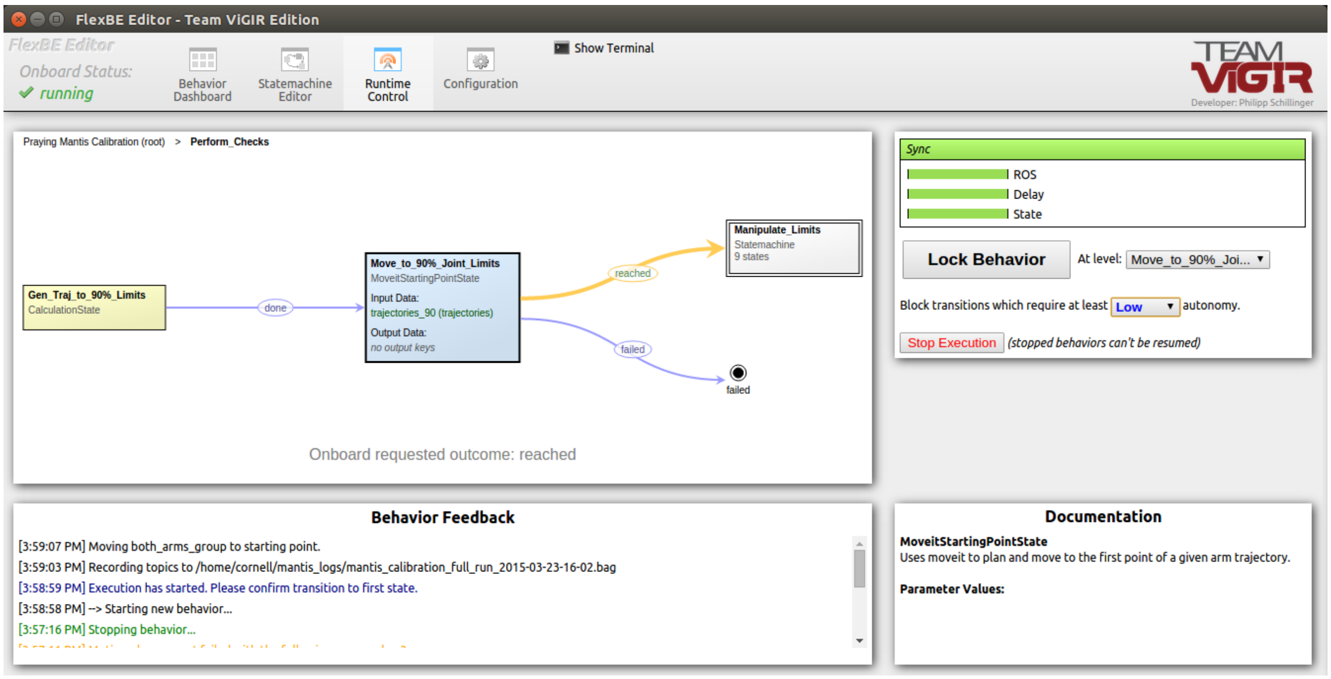
Parameters
calculation: gen_tra
gen_traj_pre_limits (limit...
gen_traj_past_limits (curr...
gen_traj_back_from_limits ...
input_keys:

Required Autono
done: Low

Input Key Mapping
trajectories_90: trajectories_90
traj_past_limits: traj_past_limits

Output Key Mapping
output_value: traj_back_to_90

Figure 1.5: Defining a behavior state machine in the *FlexBE* editor.



```
[3:59:07 PM] Moving both_arms_group to starting point.
[3:59:03 PM] Recording topics to /home/cornell/mantis_logs/mantis_calibration_full_run_2015-03-23-16-02.bag
[3:58:59 PM] Execution has started. Please confirm transition to first state.
[3:58:58 PM] --> Starting new behavior...
[3:57:16 PM] Stopping behavior...
```

MoveitStartingPointState
Uses moveit to plan and move to the first point of a given arm trajectory.

Parameter Values:

Figure 1.6: Supervising a behavior during its execution using the *FlexBE* runtime control view.

1.4 Thesis Overview

After having given a first impression of the topic and the motivation behind this thesis in this chapter, the thesis is divided into four parts, presented in chapters 2-6. At first, chapter 2 gives an overview of the current state in relevant fields of research. Main emphasize is placed on behavior control, but approaches of behavior synthesis and verification, as well as runtime modifications are touched in order to be considered for the newly developed approach contributed by this thesis. Next, chapter 3 focuses on the underlying concepts and discusses the theoretical background in an abstract manner. After summarizing the available basis provided by previous work in a uniform way, concepts regarding operator interaction and runtime modifications are added on top. Finally, consequences for behavior development are discussed.

The third part, chapters 4 and 5, presents various aspects of the implemented software based on the developed concepts. Chapter 4 targets the onboard behavior engine and shows how execution of behaviors is solved by *FlexBE* and how the process of behavior switching during runtime is integrated. Subsequently, after an initial discussion regarding the detailed approach specific to the user interface, chapter 5 presents the behavior control station, including code generation of behaviors and controlling their execution.

Chapter 6 finally concludes this thesis with an evaluation of the developed behavior control framework. Both, the development of new behaviors and the provided support during their execution, is investigated, and strengths as well as weaknesses are identified. Following, the application of this framework in the context of the *DARPA Robotics Challenge* on the robot platforms *ATLAS* and *THOR-MANG* is presented. Finally, an outlook regarding further work is given.

2 State of Research

When speaking of behavior control, there are many ways to realize such a system. Nonetheless, there are core characteristics of a behavior control system which all approaches have in common, regardless of their different ways of realization. The main purpose of a behavior control system typically is to coordinate high-level decisions, select appropriate actions, and trigger recovery from failure while assuming that the implementation of each single capability of the robot is solved separately and can be accessed by a common, known interface. Behaviors, in this context, refer to specific representations of strategies to address at least one of the three listed problems.

This chapter presents and discusses different approaches of such behavior control systems, especially varying with regard to the way how decisions are taken and how behaviors are represented. Next, going towards runtime-modifiable behavior control, current work regarding behavior synthesis and verification is presented, and it is discussed how these techniques can support the implementation of complex behaviors during runtime, considering the specific circumstances as discussed in chapter 1.1 as well as their possible use to ensure consistency and correctness of defined behaviors.

Furthermore, ways to enable runtime modifications as required for the implementation of behavior updates are discussed, and relevant research is presented. The latter, due to the innovative character of this topic, is not specifically focused on robot behavior control. Last, selected existing frameworks are introduced. These can be used as a foundation for the framework developed in this thesis and are discussed regarding their applicability.

2.1 Behavior Control Approaches

Classic behavior control approaches mainly focus on either teleoperation of single robots or as much autonomy as possible. Unfortunately, each of these characteristics comes along with its unique pitfalls and disadvantages. In order to overcome this conflict and incorporate the strengths of each approach, recent work starts focusing on solutions to combine both paradigms in an intelligent way, as for example summarized in [6].

This section presents the most relevant approaches for this thesis and discusses their advantages as well as their applicability for the given context. Furthermore, approaches which are not directly applicable are discussed in order to evaluate some of their aspects and to learn from their results and experience. But runtime-modifiable behavior control also strongly depends on the ability to monitor and influence the execution of behaviors. Therefore, not only behavior control systems are presented, but also concepts regarding human robot interaction are discussed and it is evaluated how these paradigms can be incorporated by advanced assisted autonomy approaches of behavior control. This is especially relevant because the commands an operator can give the robot have a high impact on task execution when speaking of runtime modifications, and thus have to be given on a reliable basis.

2.1.1 Autonomous Behaviors

The main goal of autonomous behaviors is to remove the need for human operators by providing decisions depending on available situational data. This includes the internal state of the robot as well as sensor data of its surroundings. The availability and reliability of this data is one of the factors which characterizes the requirements for a behavior control system. Especially the reliability has to be above a certain level in order to enable autonomous behaviors at all. If not, the human cognition is much better able to identify certain situations. Further characteristics of a task which determine the type of behavior control are for example the complexity of the task, the number of tasks to be solved in parallel, the amount of robots involved, and the probability of failure or external interruptions. As already presented in chapter 1, the characteristics considered in this thesis target a single robot solving a sequence of complex tasks while facing a high probability of failure in executing the desired actions.

In the context of the *DARPA Robotics Challenge* (DRC), many teams like IHMC [18] and WPI [10] used scripts to execute a series of predefined motions or relied on templates, as for example done by team HUBO [44], to provide known endeffector poses. These approaches are simple and require a lot of knowledge and preparation in advance, at the same time struggling to deal with variations, uncertainty and failure. For example, WPI prepared a motion script for picking up a piece of debris and moving it out of the way. In order to score the first point in the corresponding task of the DRC Trials, it is required to remove at least five pieces. The script worked well for the first four pieces, but since the motion was predefined, the robot was unable to consider its environment and collided with a wall while trying to execute the script for the fifth time.

Another, more reactive approach, are the behavior-based systems as presented in [26] and illustrated by figure 2.1. Behavior, in this context, refers to relatively simple policies which are used to preserve or achieve a single property of the system. In order to solve the given tasks,

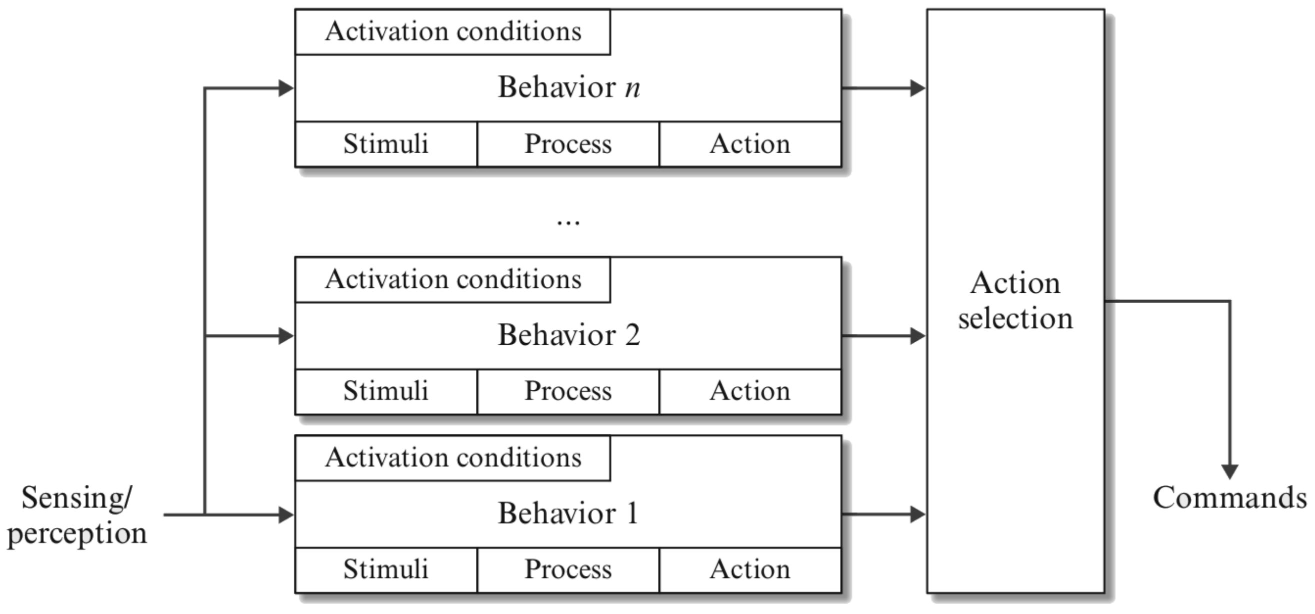


Figure 2.1: A typical behavior-based system, taken from [26].

multiple of these policies are active in parallel and their output is combined in order to form the overall behavior of the agent. As stated by [9], the reactive nature of these systems makes them less appropriate for complex scenarios. Especially when a sequence of actions has to be executed, meaning the goal to be achieved by the behavior control system changes over time, a purely behavior-based approach is hard to apply.

This yields the need for more advanced strategies of how to combine single behaviors depending on the state of task execution and the environment of the agent. [25] uses a Bayesian framework for estimating the current state of the agent and selecting the appropriate behavior when facing uncertainty regarding the environment. This approach not only helps to increase the likelihood that the desired goal is achieved, but also keeps the complexity of action selection relatively low. Action selection can also be targeted by approaches based on fuzzy logic for combining multiple applicable policies as done by [23]. [15] proposes a fuzzy discrete event system in order to coordinate between single policies.

When facing complex tasks, it is a trending and reasonable approach to use machine learning for determining and parameterizing behaviors to achieve the desired goal. It is often combined with a behavior-based approach as described above. The advantage of such an approach is the remarkably low amount of required knowledge with regard to the details of the environment which enables behaviors to be applicable under various circumstances. However, the influencing aspects of a task have to be reliably identified and taken into consideration. Finally, the system has to be trained. This is often very time-consuming and typically requires multiple attempts on the task. [42] uses modular reinforcement learning for navigation, while the modular characteristic of this approach is used to reduce the learning time. [33] relies on hierarchical abstraction in order to achieve the same goal. Given the scenario of rescue robotics, multiple attempts to solve a task while steadily learning and improving task performance is not desirable. It may even be the case that task failure leads to completely losing access to the robot, as already discussed in chapter 1. Therefore, machine learning approaches are not considered as applicable for the given context.

A different, widely-used approach for specifying the top-level behavior of an agent is the use of state machines. The active state of the state machine somehow represents the situation currently faced by the robot, while the state machine itself defines how to act in response. However, there are many existing approaches relying on state machines for realization.

The system presented in [1], the approach of team NimbRo for the RoboCup[1] Soccer competition, is one example for an autonomous behavior control system based on state machines. The system consists of two main parts, the *State Controller Library* (SC Library) and the *Behavior Control Framework* (BC Framework). Both parts are independent in principle, but are used together to address a variety of situations. The SC Library internally holds a state queue, containing states which should be executed by the controller. Each state can modify this queue in order to perform transition. This means, there is no explicit definition of transitions, but the transitions are encapsulated in the state execution instead. The BC Framework, on the other hand, is realized similar to the behavior-based approach and uses the behaviors provided by the SC Library in order to compose more complex behaviors.

1 `http://www.robocup.org/` *(03/31/2015)*

In general, there are two different approaches of using actions in the context of state machines. One approach is that actions are called when executing a transition between two states. This means that each state is determined by the runtime status of the robot, such as its current physical or logical properties. If one of these properties changes, it causes a transition and actions are selected and executed while transitioning to the subsequent state, resulting from the external property changes. Unfortunately, this approach is hardly able to react to failure or unexpected outcomes of the executed actions since the target state of a transition is already defined. The second approach defines actions to happen while a state is active and transitions only point to possible subsequent states, but don't execute anything themselves. Depending on the outcome of the called actions, the appropriate transition is selected which then points to the next state, and thus to the next actions to be executed in response to the previous result. This approach is less capable of reacting to environmental changes happening independently from executed actions, but is more robust regarding action execution itself.

Working towards a more flexible and interactive behavior control, in [2], Beer et al. define a framework for different levels of autonomy and propose criteria for determining an adequate level respective to each situation. This work can be seen as a first step towards defining behaviors that not only let the robot operate autonomously, but also try to include a human into the loop for assisting with decisions.

2.1.2 Human Robot Interaction

While approaches focusing on autonomous behavior control are centered around the robot, more complex or previously unclear scenarios critically require good operator integration. Therefore, this section briefly presents the state of research regarding human robot interaction (HRI) as to be considered for integrating cooperative behavior approaches into the natural interaction process between the robot and the operator. Deriving principles and metrics from classic HRI will help in order to apply the collected experience to interaction with behaviors and evaluate existing approaches.

The basis for a good interaction between robot and operator is provided by the following two aspects: Monitoring the current state of the robot and offering ways to operate the robot. If either of these two aspects is lacking, it will noticeably decrease the performance of the system. As already investigated by [20], human skill is an important factor for human robot interaction. It is not sufficient to just focus on the robot and improve its capabilities without considering what can be better solved by the human. Human robot teams can only perform well if both members of the team are able to utilize their unique capabilities. Communication and a good insight to what the robot is doing are basic prerequisites for this aspect.

Typically, an interactive behavior control will offer good ways of operating the robot since it is providing additional abstraction of instructions and has the goal to make operator commands less explicit yet more decisive. Opposed to this, monitoring the robot is often not integrated into behavior control, especially when focusing on as much autonomy as possible. For this reason, the main focus of a practical framework should be providing a good understanding of what the robot is currently doing in order to let the operator decide whether the robot needs help or is capable of executing the current sub-task on its own. In the context of state machine-based

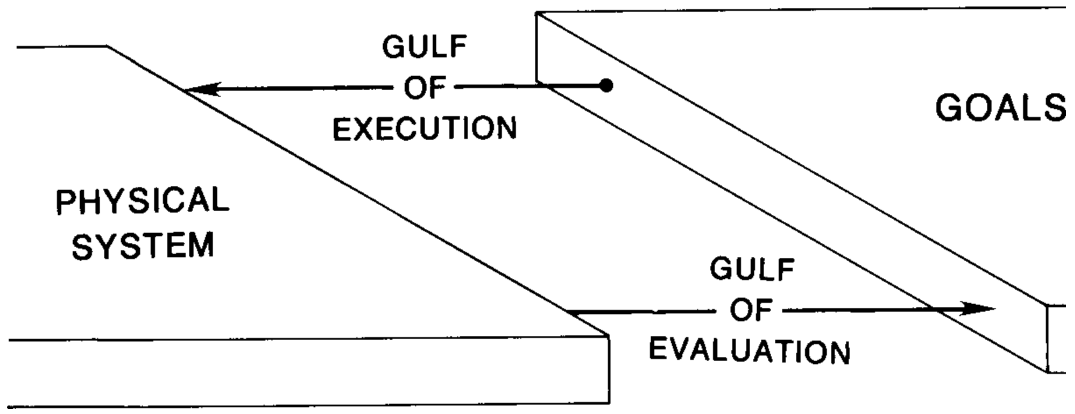


Figure 2.2: Gulfs of evaluation and execution as defined by Norman in [32].

behaviors, this mostly refers to the active state of the behavior. But the data characterizing the current state is as well important. The behavior framework, for example, should not only indicate that the robot is currently moving, but also to which goal pose. In [24], as an example, different strategies for autonomously assisted grasping have been investigated. When using autonomous planning, the planner finally visualizes the calculated gripper pose in order to let the operator know what will be executed.

In [38], Scholtz proposes a theory for human robot interaction based on Norman's seven stages of interaction [32]. Along with these seven stages, Norman identifies two possible sources of failure. The gulf of evaluation refers to the operator's limitations in being aware of the current situation of the robot and can indicate insufficient monitoring, while the gulf of execution assumes proper intents but refers to failure in giving the corresponding commands. [38] includes several roles of human team members, such as an operator for giving commands to the robot, a supervisor for being aware of the current situation and coordinating the team, a mechanic for making physical adjustments at the robot, and a bystander as external agent interacting with the robot. In the given scenario of rescue robotics, these roles are typically reduced to only having a supervisor and an operator, since physical access to the robot is often not available when executing the task. Both roles can be represented by the same person, but the characteristics of each role stay the same and the effectiveness of the system can be evaluated by metrics as presented in [40].

Although it is possible to use a single operator for controlling the robot, depending on the complexity of the system, it might be advantageous to utilize the capabilities of multiple operators. During the DRC Trials, the interaction approaches of eight different teams were evaluated and are presented in [43]. Although the exact operator roles in each team varied, an approach involving multiple operators has been chosen by almost all of the teams and the roles could roughly be divided into operators and supervisors, corresponding to the definitions above. This also corresponds to the results of [29].

In conclusion, when thinking about how to realize operator interaction with the behavior system, it is important to provide a good understanding of what the robot plans to do. This task can then be assigned to a dedicated high-level supervising operator who can then take decisions and give commands to either the robot by providing the possibility to send abstract high-level commands to the robot which can reliably be executed autonomously, or by giving instructions to an operator in order to specify low-level commands when interventions are required.

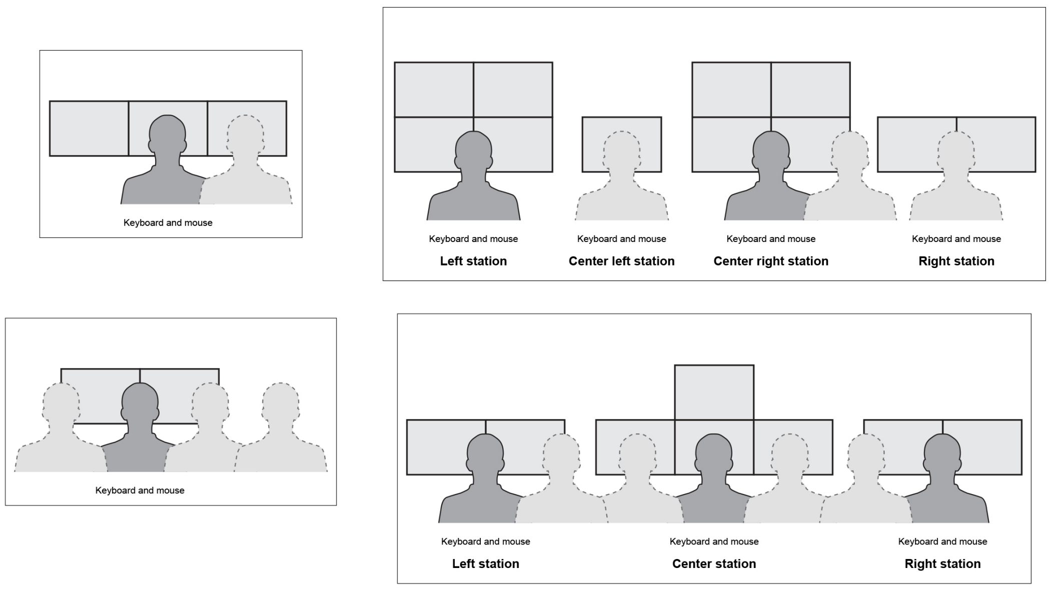


Figure 2.3: Four of the multi-operator approaches at the DRC Trials, illustrated by [43].

2.1.3 Cooperative Behaviors

It is the big advantage of runtime modifications to behaviors that the human ability of reacting flexible to unforeseen situations can be leveraged in order to optimally support the robot by choosing the right means for solving the task. Therefore, expertise on the field of human robot cooperation is the most important factor when thinking about the best way of how operators can use their cognition to alter existing behaviors based on the current situation or faced difficulties.

Combining approaches for autonomous behaviors with human robot interaction helps to realize such cooperative behaviors. However, this combination is not equally promising for all individual approaches. Almost always, it is possible for the human to replace certain sensory inputs of the robot by providing user-defined data in order to influence the actions of the robot while the robot follows a pre-defined, autonomous behavior. But this type of co-working is not very cooperative and not sufficient in advanced scenarios. A solid basis for good cooperation is, for example, provided by behaviors based on state machines. Since execution is divided into single, discrete steps, there are points for coordination. The operator cannot only adapt his behavior to optimally support the robot, but the robot as well can react more sophisticated to the operator. In addition, this approach can take temporal aspects into account, such as discussed by [4].

A more fundamental concept is presented by Johnson et al. in [17] and [16]. In their work, they point out that most of the approaches of human robot collaboration have been focused on shifting as much autonomy as possible towards the robot without respecting the interdependency between both team members. But with too much autonomy of the robot it is harder to recover from failure, which typically involves close teamwork. Autonomous behavior of the robot also implies that it has information which may not have been shared with the operator, and that it is not able to consider all information available to the operator in its own decisions. As an alternative, they present an approach called *Coactive Design*.

This design approach not only shifts the focus towards the joint nature of cooperative tasks, it also adds the requirement of being able to provide feedback regarding the own actions instead of just being able to perform them. In [19], Johnson summarizes this with the keywords observability, predictability, and directability. Each of these aspects should be seen in an active and a passive way. Observability describes the capability of being able to observe the actions of the team member as well as providing sufficient information to be observed as well. Predictability refers to not only making observations, but also predictions regarding actions in the near future. Finally, directability refers to the ability of being able to give instructions as well as receiving them.

In [39], Shah et al. present *Chaski*, the realization of a task-level executive. This is not only about letting the robot decide which action it should take when working together with a human, but also to consider which idle time of the human is caused by each action in order to minimize it, and thus improve overall task performance. This framework has been evaluated by conducting a user study where a human and a robot had to assemble three defined structures with the robot able to support the human by collecting the next relevant parts. The results show that, when being able to cooperate instead of being instructed step-by-step, not only the objective task performance is increased, but also the perceived quality of teamwork is significantly improved.

Another existing approach for addressing cooperative behaviors is the *Robot Task Commander* (RTC) as presented in [14]. This framework combines scripts called *process nodes* with state

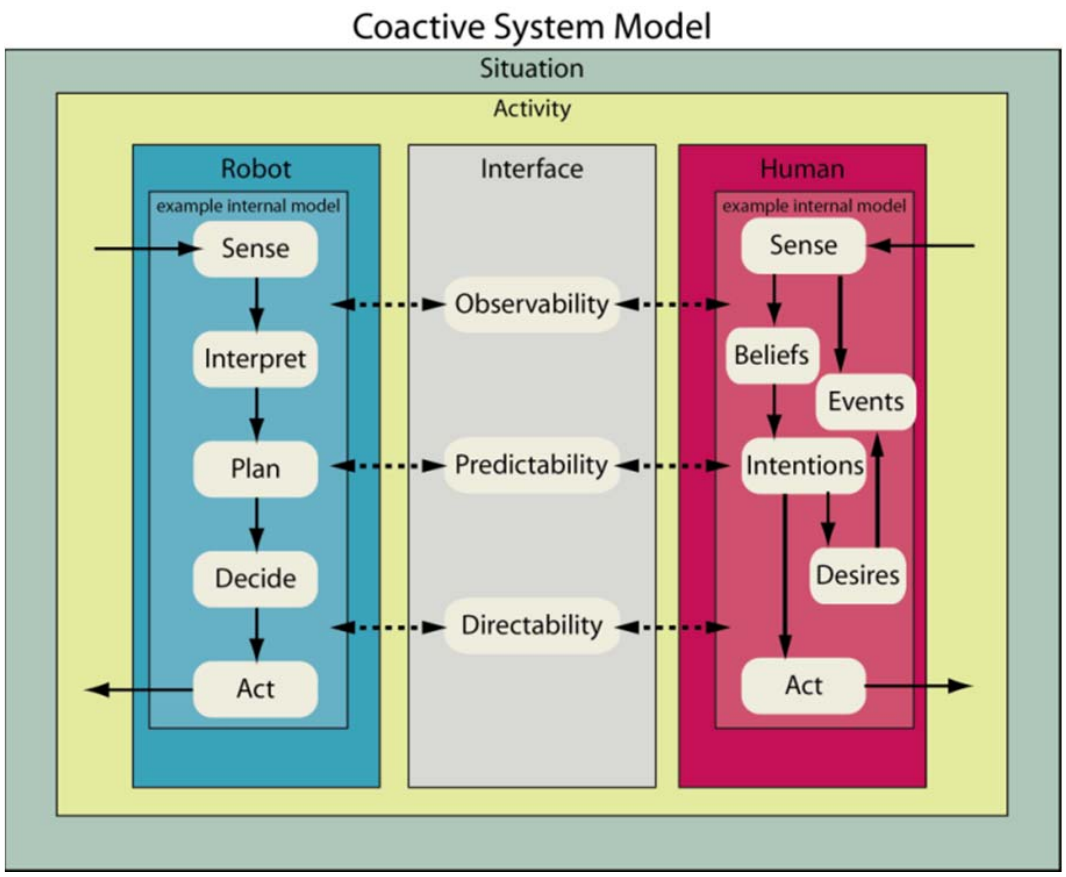


Figure 2.4: The system model of *Coactive Design* as proposed in [19].

machines, and targets not only expert users to compose behaviors, but also non-experts by providing a visual programming language for specifying the composition of state machines. For expert developers, a comprehensive integrated development environment (IDE) is offered. Behaviors are implemented in Python and communication between the user interface and the robot is done using ROS or *ZeroMQ*[2]. As stated in [34], RTC is part of the approach of team NASA/JSC to the *DARPA Robotics Challenge*.

2.2 Behavior Verification

Verifying of defined behaviors is a very important aspect of making changes to a behavior during runtime. Since practically testing of a specification before applying it to the robot is not possible in this scenario since the robot is already in the field and thus not accessible, the operator needs to be sure that applied changes to a behavior do not lead to task failure because of insufficient specification or even consistency issues.

As discussed by [12], there are several categories of verifying a robot behavior, such as logical or temporal verification. But in general, two distinct aspects have to be taken into account. On the one hand, obviously, behavior verification has to ensure that the implemented behavior meets the specification and executes exactly what is intended by the developer. On the other hand, a meaningful verification algorithm also has to check if the specifications themselves are actually reasonable in order to achieve the desired goal of the behavior.

Extensive research has already been done regarding the first aspect, implementation verification. Given a specification model, for example in UML[3] as covered by [21], software tools can reliably generate code out of it. A summary of available work on this topic is given by [11]. Therefore, automatically generating code from a specification is not expected to be a problem, even if a custom implementation is required depending on the exact way of specifying and executing a behavior. Using code generation for this step of realizing behaviors not only makes it more efficient, but also guarantees that the implementation meets the specification.

The second aspect, however, targeting specification verification is far more complicated to realize. In order to be able to verify that a specification can achieve the desired goal, not only the specification has to be modeled, but also the goal and relevant constraints. In addition, formally modeling the goal typically involves reducing the goal to a certain domain, such as a pose or configuration to be reached. Modeling the goal in an abstract way, without already implicitly involving the specification itself, is much more complicated. Furthermore, the constraints, which need to be considered in order to verify that they have been met, have multiple sources such as the physical properties of the robot, but also the environment. Since the available data regarding modeling these aspects of a task is typically very limited in the assumed scenarios relevant to this thesis, comprehensive verification regarding the correctness of the specification itself is not approached.

However, a partial verification of the behavior specification might be feasible. Reducing the demand of making sure a specification is correct to the demand of making sure that a specification meets a set of defined criteria, makes it possible to provide at least some sanity checks, and

[2] `http://zeromq.org/` *(03/31/2015)*
[3] `http://www.uml.org/` *(03/31/2015)*

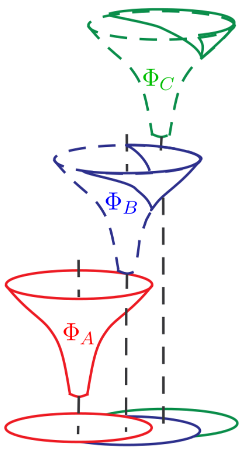
thus eliminate known, well-defined sources of common errors. A way to realize this approach is the definition of state constraints, as discussed in chapter 3.5.

2.2.1 Behavior Synthesis

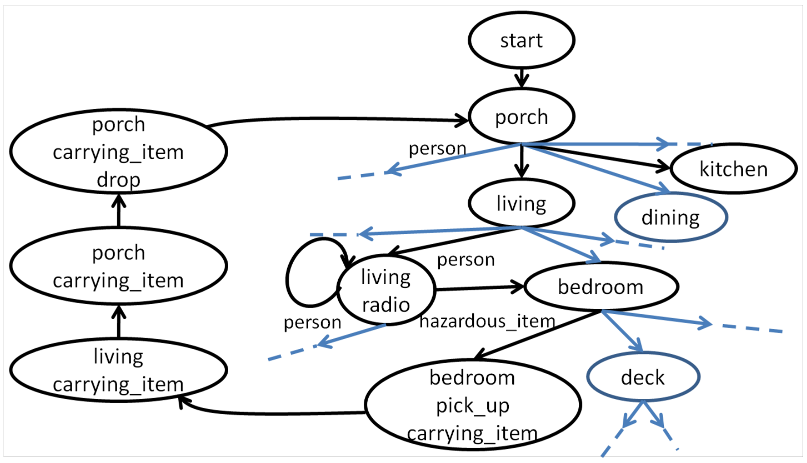
Thinking about modeling certain aspects of constraints applied on a specification finally leads to synthesizing behaviors. Instead of modeling the specification of a behavior manually, followed by a check if all constraints are respected, the specification is automatically generated from the given constraints. This process is similar to generating code for a specification, but one level higher regarding abstraction. In contrast to explicitly specifying which steps the robot should take in order to achieve a goal, you would just specify the goal itself and applicable constraints.

Although a full synthesis of behaviors is not easy to realize in the given scenario as discussed above, at least a partial synthesis, matching the capabilities of the partial verification, would be highly desirable in the context of runtime modifications to behaviors. When making changes during execution, the operator combines his own role with the role of a developer since he starts modifying the specifications of a behavior. By assisting the involved tasks required by the operator with a certain level of synthesis, changing the specification rather means giving high-level commands to the synthesizer instead of doing basic development, and thus becomes more suitable for being done by the operator.

Automatically composing certain available basic functionality in order to achieve a given goal, while respecting composition constraints, is, for example, targeted in [7]. By respecting the given constraints, local policies are selected in order to converge to the given goal. Furthermore, *Linear Temporal Logic* (LTL) [36] provides a tool for specifying certain constraints with respect to logical and temporal conditions. It can be indicated whether the specifications are synthesizable or not [35] and they can also be given in natural language using *LTLMoP*. Although going beyond the scope of this thesis, these aspects are considered. As soon as they will be available and integrated into the project, it will be possible to incorporate them into the framework developed in this thesis.



(a) Composing single policies [7].



(b) Synthesized automaton from LTL specification

Figure 2.5: Examples for possible ways to automatically compose behaviors.

2.3 Runtime Modifications

Runtime modifications can be achieved in basically two different ways. Each approach has its own benefits which will be discussed in this section. One way is to directly modify the executed source code while it is running. The capability of the used programming language to account for such on-the-fly modifications is the prerequisite for this solution. This is typically the case for scripting languages. In contrast, languages which need to be compiled before execution won't be applicable.

The other way is to specify behaviors by using an intermediate description language and parsing it for execution. Parsing can be done step-by-step whenever the next instruction or, in the case of state machines, state is required. Therefore, the description can be modified in parts which are currently not active, and these parts will be parsed later regardless of having changed since the start of execution. However, this approach is not as flexible as the first one and implies some limitations.

Since this thesis is meant to be integrated into a ROS-based project, implementation is preferably done in either C++ or Python, the two programming languages natively supported by ROS. Although there are attempts to enable a compilation-independent usage of C++ like the approach described in [3], Python is an interpreted language and does not need to be compiled. Although the currently executed code itself cannot directly be modified, modules can easily be imported during runtime and, in the case of changes, reloaded again.

Appropriate choices for an intermediate description language (IDL) are for example markup languages like XML, YAML, or JSON. In such an approach, states are implemented in any programming language without requiring this language to enable runtime modifications itself. The behavior engine then parses an IDL description which refers to the single states and defines how they are composed to form a high-level behavior. This approach of composing behaviors by using XML as IDL is for example chosen in [28]. As stated above, modifications to the IDL file could be made independently from parsing it for execution in order to achieve runtime modifications. Thus, instead of parsing the IDL as a whole, the behavior engine in this case just uses the IDL specification to look up the next state when the current one has finished. This is a very flexible way of modifying the behavior, but it also comes along with some problems.

For using an IDL this way, it is an implicitly defined prerequisite that the set of state implementations is fixed and the states itself are only functional. Each transition execution includes instantiating a new object which means that states are limited in their functionality in a way that, for example, the properties of a state object have to be constant. Nonetheless, instantiation during runtime is an additional source of error. Finally, it is the most significant problem of this approach that behaviors cannot be verified regarding consistency when parsing them on-the-fly.

In contrast, specifying behaviors as Python module is not only an approach which can benefit from the amount of available libraries and the flexibility of Python. It also enables to instantiate all states required during the execution of a behavior when it is started, since the state machine specification is part of the class description.

2.4 Existing Behavior Frameworks

After having discussed the most relevant approaches in general, this section will now focus on selected work which comes into consideration as a foundation for the approach developed in this thesis. This implies certain demands on the existing frameworks, derived from the concepts discussed in the previous sections of this chapter.

Since the main contribution of this thesis is the ability to modify currently executed behaviors, being able to extend the regarding framework by one of the two ways discussed in section 2.3 is an important criterion. This means that implementation has to be possible by either using a scripting language like Python, or by providing behavior definition in an intermediate format such as XML. For the reasons discussed in this section, it is favorable to not need to rely on an intermediate description language in order to improve robustness and provide flexibility for developing single states. However, both approaches are possible and if the framework offers functionality for addressing these issues, it is also a valid choice.

Its provided degree of operator interaction is another relevant factor for choosing the right framework as basis. Ideally, the framework already provides means to monitor the current state of execution at a remote operating station, and allows for adjustments by the operator influencing runtime parameters or at least pausing behavior execution at certain points in time. Pausing execution is a helpful prerequisite for modifying behavior during runtime because it allows to ensure consistency of changes. On the other hand, changing the behavior should not prevent the robot from taking actions in the meantime. This is discussed in detail in chapter 3.3.

In addition to the mentioned requirements, there are some advantageous aspects a chosen approach may support in order to facilitate development of advanced features. These aspects mostly result from the fact that behaviors will be modified during runtime which requires efficient and intuitive specification of behaviors. Furthermore, the behaviors have to be robust in order to prevent failure during execution. For this reason, the framework ideally incorporates consistency checks and some sort of behavior verification. An appropriate user interface would also be a plus. Last but not least, it must be possible to integrate the system into the rest of the project by relying on ROS.

2.4.1 XABSL

Providing a high-level specification language for robot agent behaviors is the goal of *XABSL*. It has been developed by Lötzsch et al. in context of the *RoboCup* as described in [42] and extended in [9]. Behaviors can be specified by composing basic behaviors, referring to simple actions, and behavior modules called *options*. Options can refer to other options and thus form a hierarchy, the so-called *option graph*. Internally, options are modeled as state machines where each state contains a decision tree to determine the next transition.

XABSL relies on XML as an intermediate description language. Although initially development has been done by writing the XML description manually, *XABSL* has been extended to use

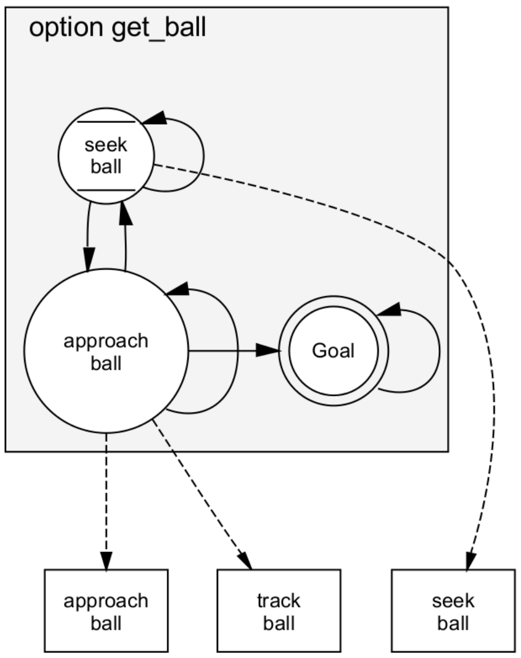


Figure 2.6: An example option definition in *XABSL*, taken from [9].

a dedicated programming language which is compiled into XML. In order to improve the versatility of developed behaviors concurrent execution of actions has been added later as well as coordination between multiple agents.

Considering the given requirements, *XABSL* may not be the ideal choice. Although the concept appears to be generally applicable, significant effort would be required to integrate operator interaction into the decision tree of a state and allow for remote monitoring during runtime. The need to compile the *XABSL* programming language is not ideal as well, and a step back to using pure XML would be required. Finally, *XABSL* is not integrated into ROS.

2.4.2 RoboCup Frameworks

Besides *XABSL*, several other behavior frameworks have been developed along with the *RoboCup* competitions. Since these competitions, such as robot soccer or service robotics, require a significant amount of autonomous capabilities and strategic reasoning, behavior frameworks developed in this context are likely able to provide a good basis for further extensions.

b-script has been developed by de Haas et al. and is presented in [8]. This approach is not based on an intermediate description language for defining behaviors. Instead, *b-script* itself is a scripting language based on the programming concept of generators, and combines aspects of Python, such as its scripting nature, with aspects of C++, e.g., its strong type system. Since the goal of *b-script* is providing behaviors which can be executed when facing limited hardware performance, it can also be compiled to C++. Behaviors developed with *b-script* have been applied in the *RoboCup* Standard Plattform League by the teams *RoboEireann* and *B-Human*.

The *Lua-based Behavior Engine* [31] developed in conjunction with the *Fawkes Robot Software Framework* by the *RoboCup* team *AllemaniaACs* is another example for such a behavior framework. Although primarily developed for the use with *Fawkes* and the *Nao* robot plattform, it has been ported to *ROS* later and is based on the lightweight scripting language *Lua*. This engine assumes three behavior levels, namely *Control Loops*, *Skills*, and *Agent*. Using this behavior engine, skills can be defined in order to incorporate basic robot capabilities. An external behavior executive is then required to compose and execute these defined skills.

Although these frameworks provide good ways for defining autonomous behaviors and especially multi-agent interaction, they are hardly applicable for scenarios such as disaster mitigation. One important problem is the limited incorporation of operator involvement during runtime, mainly supporting passive monitoring for debugging, but no cooperation in order to fulfill a task. Furthermore, their main application are robots with very limited hardware resources, operating in a well-defined and structured environment, such as a soccer field. This is very different from what this thesis is targeting, impeding to utilize some of the underlying concepts of these behavior frameworks.

2.4.3 SMACH

Developed by Boren et al. in [5], *SMACH* offers a Python API for defining executable state machines. States are realized as classes which can implement the `execute` function to perform blocking actions and return an outcome based on the result in order to determine a transition. The transitions are defined by the state machine and state machines can be used as states as well in order to define a hierarchical composition. In *SMACH*, states can not only return transitions, but also arbitrary data to be passed between states, the so-called *userdata*. *SMACH* comes along with a user interface for locally monitoring the execution of a state machine by displaying its structure and indicating which state is currently active, as shown by figure 2.7.

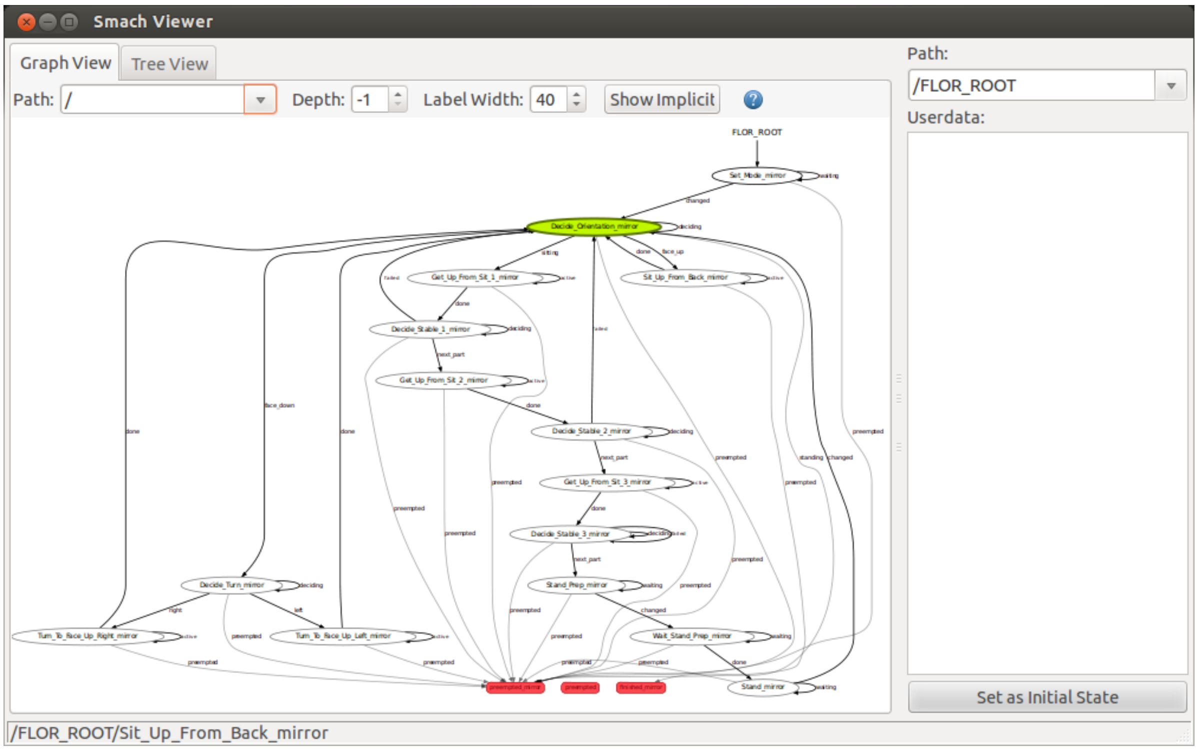

Path: /
Depth: -1
Label Width: 40
Path: /FLOR_ROOT
Userdata:
/FLOR_ROOT/Sit_Up_From_Back_mirror

Figure 2.7: The *SMACH Viewer* while monitoring execution of a behavior defined by *Team ViGIR* for the *Virtual Robotics Challenge*.

Although being a standalone library in its core, *SMACH* is integrated into ROS and works well in this combination for coordination between external ROS nodes. It has proven its applicability in several projects along with the PR2[4] and other robots and is for example combined with a decision making tool in [13].

In general, *SMACH* is applicable as basis for the behavior framework since it is easily extensible and well integrated into ROS and Python. It also offers a basic form for monitoring the execution of a state machine. However, it does not incorporate the operator into its concept and thus needs to be extended with cooperation principles and the ability to give user commands to behaviors.

2.4.4 Flor Behavior Engine

The *Flor Behavior Engine* (FlorBE) is a previous work of the author of this thesis. It has been developed in the context of *Team ViGIR*'s previous participation at the *DARPA Robotics Challenge* and is presented in [37]. Main goal of this behavior engine is to better integrate the operator into behaviors, allowing for cooperative approaches to tasks and compensation of missing autonomous capabilities of the robot, while taking as much workload as possible away from the operator considering the respective scenario.

FlorBE is based on *SMACH* to utilize its flexibility and extends several aspects, mainly regarding communication with a remote operator. For this reason, it comes along with a very simple user interface for triggering transitions and displaying transition intents of the behavior to the operator. This user interface is intended to be integrated into an existing system for monitoring the physical and internal state of the robot, but based upon the existing communication protocol between robot and operator, a more advanced user interface can be added.

The introduction of a coordination mechanism, the *Autonomy Level* is a further extension to *SMACH*. This mechanism helps the operator to flexibly reduce the autonomy of the robot and

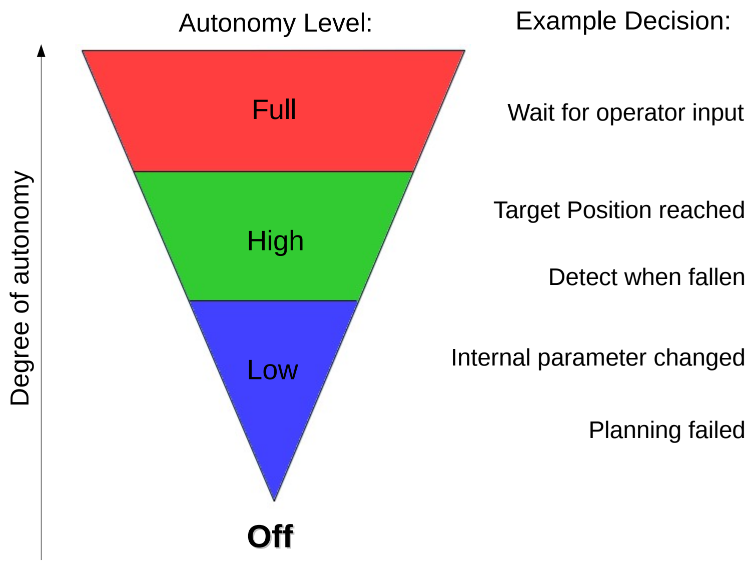


Figure 2.8: Example actions of different *Autonomy Levels* as defined in [37].

[4] https://www.willowgarage.com/pages/pr2/overview *(03/31/2015)*

thus prevents it from making decisions on its own. As a result, behaviors are able to deal with changing uncertainty in scenarios while using the same state machine for implementation of the actions to be taken.

This combination of *SMACH* and approaches for coordination between operator and robot is a good basis to build upon. For this reason, more details regarding *FlorBE* are given in chapter 4.1. However, behavior development is very tedious when implementing complex behaviors and especially when defining the transitions between states the current way is prone to consistency errors. Therefore, improving the development process to a level at which behaviors can easily and safely be defined while executing a task will be a very important extension to *FlorBE*. Furthermore, the capabilities of Python will be used for importing behaviors during runtime and thus for being more flexible regarding custom behavior changes.

3 Concepts

The onboard behavior engine part of the approach developed in this thesis builds upon previous work of the author, and therefore extends *FlorBE* which has been briefly described in the previous section and will be discussed in detail in chapter 4.1. This chapter discusses the available basis and required extensions in a formal way to provide a basis for implementation and further development. Additionally, the requirements of the given scenario, which have been presented in chapter 1.1, are taken into consideration by the concepts of this chapter.

Most of the concepts presented in sections 3.1 and 4.1.2 are based on the existing implementations of *SMACH* and *FlorBE*. However, they are summarized in a formal way and have been extended to support the newly added concepts. Also, providing arbitrary data has been a problem in *FlorBE* and section 3.2.2 provides a concept for solving this issue.

Subsequently, section 3.3 presents the overall concept of runtime modifications to behaviors and discusses aspects such as consistency. Further, Section 3.4 targets on how to assist development of behaviors and to specify changes, providing a basis for the newly developed user interface. Finally, section 3.5 discusses a formal approach on guaranteeing successful behavior execution already when defining a specification by introducing state constraints.

3.1 Definitions

In order to enable discussion of the underlying concepts of this thesis, it is important to exactly define the used terms and relate them to each other. The goal of this thesis is the development of a flexible high-level behavior engine, thus we assume that the basic capabilities of the robot are already available and provided externally.

Definition: Primitive. *A primitive $p \in P$ is any atomic capability of the robot which converts a set of input data D_i into a set of output data D_o in order to fulfill a specified goal G_p.*

These primitives, in general referenced as actions, are typically not instantaneous and side-effects may change the physical state of the robot such as moving its joints. For example, there can be a grasping primitive which provides a way to close the fingers of the robot. Input data can be a specification of how much the fingers should be closed or maybe which force should be applied to the grasped object. The output data then can indicate the sensed force to the fingers and the closing state. A second example for a primitive is a state which plans a certain motion, getting a target endeffector pose as input and generating a joint configuration which can be used for reaching this pose. This primitive would have no side-effects since it only plans the motion, but does not execute it. A third primitive could then provide a way to execute a certain joint configuration.

Execution of a primitive can fail which means that a subset of the output data doesn't meet the goal specification. Related to the example of motion planning, this could mean that the specified target pose is not reachable. Primitives are not explicitly defined for behaviors and can be provided in any format. Thus, for making them usable to behaviors, there is a way required for interfacing them.

Definition: State. *A state $s \in S$ interfaces functionality with behaviors, where the set of all state implementations S includes states based on primitives $S_p \subseteq S$. Each state defines outcomes s_{Oc} and execution of a state is terminated by returning one outcome matching the result of execution $oc \in s_{Oc}$.*

States based on primitives form the majority of all types of states. They evaluate the data returned by their respective primitive in order to decide about which outcome to trigger. For this purpose, each $s_p \in S_p$ defines an evaluation function $e : D_O \longrightarrow s_{Oc}$ which maps the output data of its primitive to one of the declared outcomes of the state. Assuming for example the grasping primitive described above, a grasping state could specify three outcomes: *grasped*, *failed_to_close*, *failed_to_grasp*. The outcome *grasped* is only returned if the fingers are actually closed and a force to the finger sensors indicates that the object is inside. It would return *failed_to_close* if it is unable to close the fingers and *failed_to_grasp* if the fingers are closed, but there is no object inside.

In special cases, states can also invoke multiple primitives and combine their single results, but in favor of simplicity and modularity, this is a rather exceptional case. Besides referring to a primitive, states can also be implemented in an arbitrary way depending on the purpose they should provide to their containing state machine.

Definition: State Machine. *A state machine SM composes a set of states S_{SM} where each state $s^{(i)} \in S_{SM}$ is the instantiation of a state implementation $s \in S$. When a state returns a specific outcome $oc^{(i)} \in s^{(i)}_{Oc}$, the transition function $t_{SM} : S_{SM}, s_{Oc} \longrightarrow S_{SM} \cup SM_{Oc}$ defines which state is executed next.*

The transition function t_{SM} assigns each pair of a state and one of its outcomes $(s^{(i)}, oc^{(i)})$ to its successor. This successor can either be the next state to be executed $s^{(i+1)}$, or one of the outcomes $oc^{(SM)} \in SM_{Oc}$ of the state machine itself which means that finishing execution of this state will also finish execution of the whole state machine.

State machines are states themselves and thus provide a way for representing hierarchical state machines. The purpose of a state machine is to define the control flow of a behavior, i.e. the sequence in which states are executed depending on their result, using the specified transition function.

In addition to the control flow, states need a way to pass data to each other. For example, there can be a state which provides perception data by using an object recognition primitive and returns a pose of interest. Another state may now plan footsteps to a target pose for letting the robot walk and thus needs a way to get the pose generated by the perception state.

Definition: Userdata. *Each state machine SM defines userdata D_{SM}, represented by a set of key-value pairs where $f : K_{D,SM} \longrightarrow V_{D,SM}$ maps each specific key $k \in K_{D,SM}$ to its respective value. States define userdata keys they need, s_I, and provide, s_O.*

Passing data from state A to state B with $s^{(A)}, s^{(B)} \in S_{SM}$ is thus defined as $f(k_{I,sB}) \mid k_{I,sB} = k_{O,sA}$ where $k_{I,sB} \in s_I^{(B)}$ is an input key of $s^{(B)}$ and $k_{O,sA} \in s_O^{(A)}$ is an output key of $s^{(A)}$.

Using the concept of *userdata* as provided by SMACH [5] gives an easy and flexible way of passing arbitrary data between states. As mentioned above, there can be one state for generating a certain joint trajectory and a second state for executing any joint trajectory. The concept of *userdata* allows the planning state to specify an output key for holding the generated trajectory after the state has finished execution and the executing state to specify an input key for providing the joint trajectory to be executed. This not only allows for easy combination of these two states, but this one execution state can also be used along with any other state which generates a joint trajectory or even accept manually defined trajectories.

Definition: Behavior. *A behavior B is implemented by its state machine B_{SM}. Furthermore, each behavior defines a set of parameters B_P where each $p \in B_P$ is chosen when starting a behavior. Behaviors can be used as states of other behaviors.*

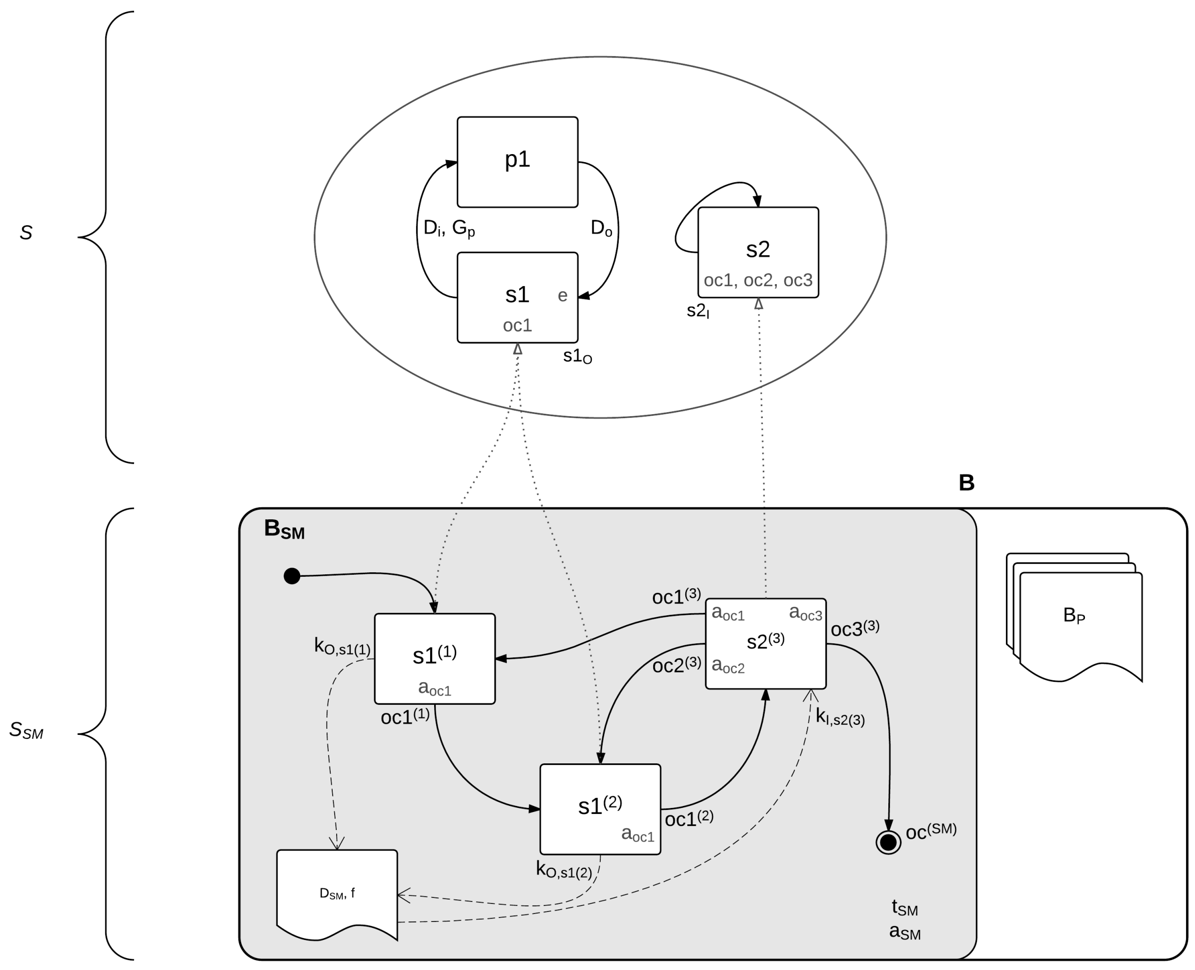


Figure 3.1: Composition diagram of the concepts presented in this section, t_{SM} is implicitly defined by the solid arrows inside B_{SM}. In this example, s1 is a state implementation based on the primitive p1 and may execute an action generating data while s2 performs branching based on its input data.

While the implementation of a behavior is given by its state machine, the interface of a behavior is defined by its behavior manifest. The manifest contains meta information such as the name of the behavior and the author, but also important interface specifications e.g., which other behavior is included in this one. Including a behavior into another means the included behavior is instantiated along with all the states of the outer behavior's state machine and embedded into this state machine as one of the states. This enables the creation of behaviors in certain layers of abstraction. Some lower level behaviors, for example, may be constructed by combining only a few states based on primitives and then being composed by a complex task-level behavior. The number of layers of abstraction depends on the complexity of available primitives as well as of the tasks to be solved.

A concept introduced by the *Flor Behavior Engine* in [37] is the *Autonomy Level*. It regulates the control flow of a state machine by stopping certain transitions when the operator declares that they are not reliable enough with respect to the current situation.

Definition: Autonomy Level. *In a state machine SM, each outcome of a state $oc^{(i)} \in s^{(i)}_{Oc}$ defines a minimum autonomy level $a^{(i)}_{oc}$ which will block the corresponding transition $t_{oc,i} := t_{SM}(s^{(i)}, oc^{(i)})$. During execution, the current autonomy level is determined by a_{SM}, thus only allowing the execution of $t_{oc,i} \mid a^{(i)}_{oc} < a_{SM}$.*

All transitions which are not allowed to be triggered autonomously are suggested to the operator and require explicit manual confirmation to be executed. This mechanism ensures that transitions are only executed when their required outcome is reliably detectable. The *Autonomy Level a_{SM}* can be changed at any time during behavior execution, enabling a way of small adjustments regarding the control flow of the behavior. However, the changes are very limited and in order to provide the flexibility required by the scenarios described in chapter 1.1, additional ways of making changes to a behavior need to be added.

3.2 Operator Interaction

In order to facilitate a close teamwork between robot and operator, the behavior framework supports a built-in interaction protocol. Following this protocol, a running behavior can be monitored and several commands can influence execution of the behavior. Basic monitoring of the current state is supported by the *Behavior Mirror*, a concept targeting at robustness and reduction of required bandwidth, which will be explained later in this section.

Based on the previously defined *Autonomy Level*, the behavior will not always be allowed to actually trigger its desired outcome. In this case, the operator should be notified that a certain transition may be ready to be taken. Afterwards, it is up to the operator to verify or falsify this suggestion. In order to reduce bandwidth usage and prevent redundant messages, all suggested outcomes $oc^{(i)}$ are added to a set R_i of sent requests. Thus, a request is only sent if $oc^{(i)} \notin R_i$.

While rejecting a suggested outcome is easily achieved by ignoring the respective suggestion, verifying it needs a command for triggering an outcome. Actually, the taken approach does not distinguish between outcomes triggered as a reaction on a suggestion or triggered on purpose of the operator. However, when the operator commands a certain outcome, the currently active state is forced to return this specified outcome regardless of the result of its own evaluation function.

In addition to its regular outcomes, each state instantiation has a special implicit outcome $oc_p^{(i)} \in s_{Oc}^{(i)}$. The purpose of this outcome is to preempt the current execution and to offer the operator the possibility to completely stop a behavior at any point in time. For this reason, the outcome $oc_p^{(i)}$ of any state inside a state machine $s^{(i)} \in S_{SM}$ fulfills the following condition: $t_{SM}(s^{(i)}, oc_p^{(i)}) = oc_p^{SM}$. This means, a preempting outcome is always propagated to the hierarchy's root and thus causes the whole behavior to be preempted, not only the current state.

Another command available to the operator is locking a behavior in its currently active state. This command is newly added and has not been available in *FlorBE*. If a behavior is locked in a certain state, it means that this state will still be executed in the usual way. But in case the state would return any outcome, this outcome is temporarily stored, but not returned. This is expressed by the following condition: $\forall oc^{(i)} \in s_{Oc,l}^{(i)} : t_{SM,l}(s_l^{(i)}, oc^{(i)}) := s_l^{(i)}$ where the subscript l indicates that the referred state is locked. The regular transition function t_{SM} is temporarily replaced by a special transition function for locking $t_{SM,l}$. After a state has been unlocked again, it will return its stored outcome, if any.

3.2.1 Behavior Mirror

The *Behavior Mirror* allows for easy and efficient monitoring of the current state of execution of any active behavior. It is supposed to mirror the remotely running state machine in order to provide a local, easily accessible representation. All tools which require access to the behavior in order to monitor its execution, can instead use the *Behavior Mirror*. Since a mirrored behavior is intended to completely remove the need for further remote behavior access, the *Behavior Mirror* has to make sure the mirrored behavior correctly reflects the actual behavior's structure at any time.

Definition: Mirrored Behavior. *A behavior B_m is called a mirrored behavior of B whenever the structure of its state machine $B_{SM,m}$ is isomorphic to B_{SM} and $s_{a,m}^{(i)} \in S_{SM,m}$ is determined by $\varphi(s_a^{(i)})$, where φ is the isomorphic transformation $\varphi : S_{SM} \longrightarrow S_{SM,m}$.*

Note that φ is bijective, meaning that it is always possible to know $s_a^{(i)}$ when having access to $s_{a,m}^{(i)}$. Furthermore, the definition does not include any requirements regarding the actual functionality of a state, meaning the state implementation of $s^{(i)}$ and its mirrored counterpart $s_m^{(i)}$ may differ. This is a rather essential requirement since it is not desirable to actually execute a behavior in order to mirror it. Instead, the *Behavior Mirror* provides a separate implementation for mirrored states which does not execute anything state-specific and just updates monitoring information.

This concept enables to have minimal updates regarding mirror synchronization while always knowing the currently active state of the remote behavior. Prerequisite is a once synchronized state, which trivially is the case when starting a behavior. Afterwards, when executing a transition, it is sufficient to notify the *Behavior Mirror* about which of the available transitions has been taken. Since the number of available outcomes is naturally rather small, there is not much data to be sent.

In addition to synchronizing the currently active state while a behavior is running, the *Behavior Mirror* also has to be aware of when a behavior is running. For example, still publishing the last active state when the behavior has already stopped execution is not desired and, of course, the *Behavior Mirror* has to be notified whenever a new behavior starts. In the past, *FlorBE* achieved this by sending separate *start* messages and relying on the preempting capability of the mirrored behavior. However, this is not the most robust solution and therefore, this approach has been changed in *FlexBE*. The new approach uses a dedicated onboard engine status message in order to notify the mirror and any other related component about behavior starts and finishes. Furthermore, it defines error codes and allows for more detailed runtime status updates in addition to starts and finishes.

3.2.2 Behavior Input

Another important part of operator interaction is the ability to exchange arbitrary data. For example, a behavior could ask for operator assistance in order to identify a certain pose. Thus, the operator needs a way to provide the pose data, which is available at the control station, to the remote robot. This type of data is different compared to the previously treated kind of data. Until now, only data regarding the control flow of a behavior has been addressed.

In contrast to data regarding the control flow, the kind of task-specific data required during execution cannot be determined beforehand. However, in order to realize a modular and reusable approach, it is no solution to specify each type of data request independently. Instead, a general way to provide data, no matter of which type, is preferred when being able to have a special communication channel especially dedicated to handling these data requests. This can be achieved by converting the data to be sent to the behavior into a common format. For this reason, there is the *Behavior Input* component which uses serialization in order to achieve this goal. In addition to just converting the data of any type into a common, serialized representation, this component can also apply compression of the sent data.

Definition: Behavior Input. *Providing data to a behavior is modeled as a special primitive* p_{ID}. *Input data* D_i *of* p_{ID} *specifies the required type of data while* D_o *provides the serialized result. This result is evaluated by* e_{sID} *and the deserialized data provided as output userdata* $k_{O,sID}$.

As depicted in figure 3.2, the whole process of input data transfer is done in an event-based fashion. Whenever the behavior needs some data, it will send a certain input request indicating the operator which type of data is needed and for which purpose. The operator is then able to process this request in an arbitrary way depending on how the operator control station is implemented by the respective system. After he is done, the *Behavior Input* component of the control station receives the data and can then send the data back to the robot using a dedicated communication channel where arbitrary data is serialized and can be compressed as described in the previous paragraph.

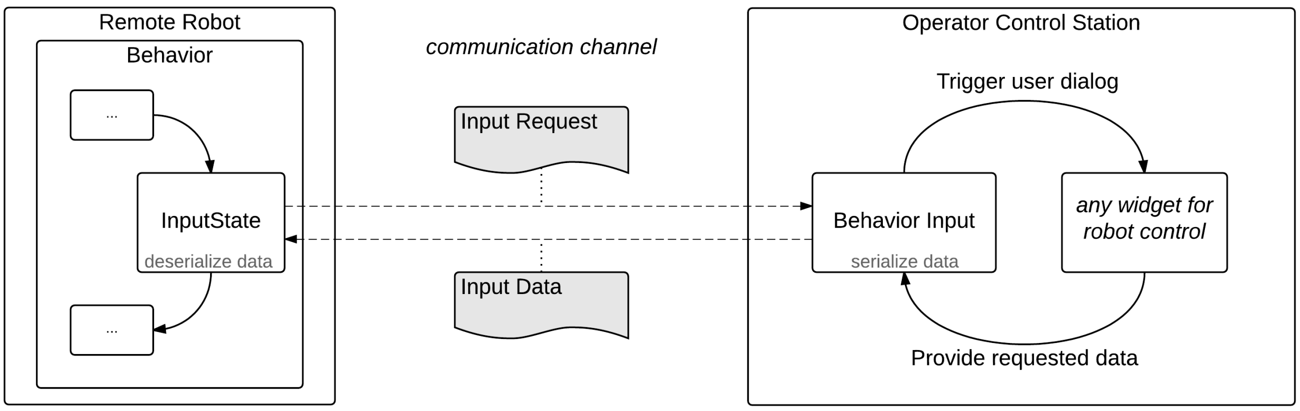


Figure 3.2: Process of sending input data to any remotely executed behavior.

3.3 Runtime Modifications

The ability to perform runtime modifications is the most complex command available in *FlexBE*. It enables the operator to make arbitrary changes to the structure of a behavior without the need for stopping, compiling and re-starting it. Although this capability is very helpful regarding adaptability to unexpected situations, it also introduces some challenges which had been discussed in 1.1. For this reason, the next section will present approaches to avoid failures related to runtime modifications and defines constraints to preserve consistency across versions of a behavior. But at first, it is necessary to define what exactly is considered as a modification.

Definition: Behavior Modification. *Modifying a behavior B and its state machine B_{SM} corresponds to adding or removing elements to or from S_{SM}, changing the properties of any $s^{(i)} \in S_{SM}$, or changing the transition function t_{SM}.*

It is important to notice that this definition refers to the state instantiations $s^{(i)}$ specific to the behavior state machine B_{SM}. The state implementations themselves, $s \in S$, are not included in this definition and are thus not part of a behavior modification. Therefore, the set of state implementations S remains the same across behavior modifications, thus building a constant base to which behaviors can refer. This is especially important since different behaviors can instantiate states of the same implementation. If it was possible to also alter the state implementation when modifying a behavior, this modification would also implicitly affect other behaviors.

Properties of a state are for example the *Autonomy Levels* of its outcomes, since $a_{oc}^{(i)}$ is a value specified for each state instantiation $s^{(i)}$. In contrast to this, the outcome itself cannot be changed, since the set of outcomes s_{Oc} is defined by the state implementation $s \in S$ and is therefore independent from a specific state machine. However, the outcomes of an inner state machine are not given by its implementation and thus can be changed.

Another property of a state is its given name inside a state machine, which also fulfills the purpose as an identifier for the transition function t_{SM}. In order to define t_{SM}, each state instantiation specifies its successors depending on the returned outcome by using its unique name. When now a name is changed, for example the name of $s^{(i)}$, t_{SM} has to be updated as well by also adapting the name change to the successor specification of each state $s^{(j)} \in P^{(i)}$ where $P^{(i)}$ is defined as $P^{(i)} := \{\forall s^{(j)} \in S_{SM} \mid \exists oc^{(j)} \in s^{(j)}_{Oc} : t_{SM}(s^{(j)}, oc^{(j)}) \overset{!}{=} s^{(i)}\}$ which precedes $s^{(i)}$.

An even more powerful modification can be done by changing the parameter values of a certain state instantiation. For example, one of the state implementations offers to perform an arbitrary calculation which is specified by a function provided as parameter of this state. When now changing the parameter value of instantiations of this state, one actually changes parts of its implementation.

Changing the *userdata* of a state machine is rather complicated. The problem is that it not only specifies available keys for accessing arbitrary data, but also has to provide this data depending on what results occurred during the past execution. For example, if a new state $s^{(A)}$ is added such that $s^{(A)} \in P^{(B)}$ precedes the currently active state $s^{(B)}_a$ and there is an $k_{O,sA} \in s^{(A)}_O$ which corresponds to one of the input keys $k_{I,sB} \in s^{(B)}_I$, $s^{(B)}_a$ would still refer to its own value passed to $k_{I,sB}$ when the state has been entered. This resulting inconsistency can probably lead to failure of behavior execution. It is not possible to update $k_{I,sB}$ to the value which would have been provided by $k_{O,sA}$ because the state $s^{(A)}$ has just been added during the modification and not been executed before, thus $k_{O,sA}$ would be undefined.

However, there is a second way of how to add *userdata* to a state machine. Each behavior has a set of pre-defined *userdata* keys which also assigns a default value to each of the specified keys. Adding keys to this set is a safe way of adding *userdata* to a running state machine and thus is allowed as part of a behavior modification.

3.3.1 Consistency

In order to define constraints with the purpose of providing consistency across behavior modifications, it has to be defined first how a modification can be realized. As already defined in the previous section, a behavior modification can consist of changes to S_{SM} and t_{SM}. Nonetheless, a modification should not always be applied immediately. For example, consider a case where a state $s^{(A)}$ is removed from S_{SM}. If also removing the reference to it in t_{SM} from all $s^{(i)} \in P^{(A)}$ is not an atomic modification, the state machine would be in an inconsistent state. Furthermore, also if it is atomic, the state machine is inconsistent as well because all outcomes which previously pointed to $s^{(A)}$ are now unspecified. So, before wanting to apply this modification, a consistent state has to be ensured. This can be achieved by using an explicit version update when having reached a correct and consistent new behavior version, which will also help to reduce the risk of failure and improve communication efficiency.

Definition: Version Update. *Modifying a behavior B is realized as an update from B to its more recent version B' where both, B and B' are consistent behavior specifications.*

Each version update requires a pivot state which defines how to map the current state of execution to the new version. To be more precise, the pivot state is the state which is currently active before performing the behavior update and is supposed to be still active when the update has finished.

Definition: Pivot State. *When performing a version update, exactly one state $s_a^{(i)} = s_{piv}^{(i)}$ is called the pivot state of this particular update. $s_{piv}^{(i)}$ is immutable during the update and has to be element of $S_{SM} \cup S_{SM'}$.*

In order to provide consistency, the pivot state has to be locked during the whole process of behavior modification: $s_{piv}^{(i)} \Rightarrow s_l^{(i)}$. This ensures that the pivot state stays active during the update process. Furthermore, locking the pivot state before actually starting to modify the behavior ensures the required immutability during the modifications. Since state machines are also states themselves, the pivot state can be a state machine as well. In this case, the immutability requirement implies that the inner state machine of the pivot state remains the same which means that no type of change is allowed to any of the inner states or the inner transition function.

If the behavior would not be locked during a modification, the active state is likely to change in the meantime because specifying the desired modifications requires some time. Even if the amount of required time is very small and the expected execution time of the action requested by the active state is sufficiently high, events might always occur which can trigger a transition by letting the state return one of its outcomes $oc^{(i)}$. This means the onboard behavior would transition to a different state $s_a^{(i+1)} := t_{SM}(s^{(i)}, oc^{(i)})$. Propagating this transition to the version currently being modified by the operator is not possible, since it is not guaranteed that still $s^{(i+1)} \in S_{SM}$ because this state could have been deleted along with the modifications made to

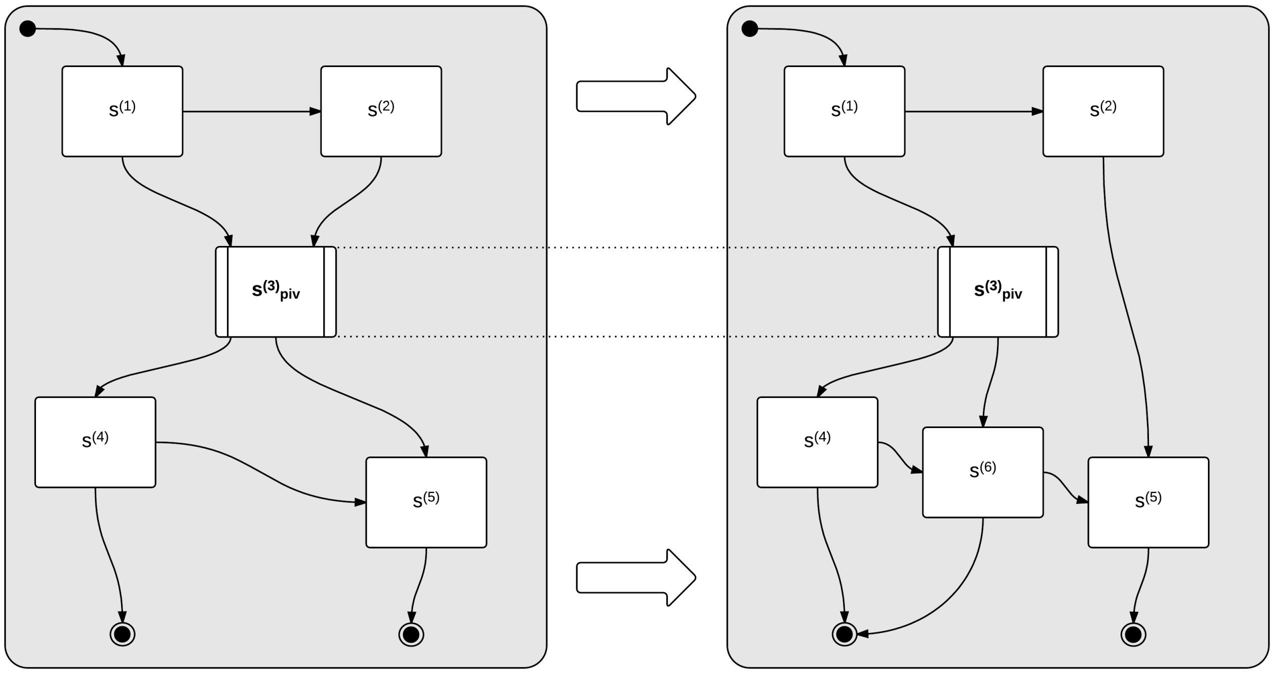


Figure 3.3: Example version update where the active state, $s_{piv}^{(3)}$, has to stay active and unchanged during the whole process. However, the targets of its transitions can be changed.

the behavior. As a result, this would violate the condition that the pivot state stays active during a version update, as required by the definition of the pivot state, because $s_{piv}^{(i)} \neq s_a^{(i+1)}$. For this reason, behavior modifications are only possible if a behavior is locked. This also ensures that the robot executes no unexpected actions while the operator might be distracted by modifying the behavior, as already discussed in chapter 1.1.

3.3.2 Behavior Update

When commanding a behavior version update, the operator needs a way to either send the new behavior version or send an indication of all modifications which can then be applied to the already known and running version to construct the new one. In order to keep the required amount of data low, it is favorable to just send a minimal summary of applied modifications instead of the whole new version, since the part of which a behavior changes is typically much less than the part which stays the same. This minimal summary of modifications is called *patch* and applying a patch to the appropriate previous version of a behavior will create a new behavior containing exactly the changes for which the patch has been created.

Definition: Behavior Patch. Δ *is a function in the behavior space and called the patch of a version update* $B \longrightarrow B'$ *if it fulfills the condition* $\Delta_{B \to B'}(B) = B'$.

Besides transmitting minimal data by relying on $\Delta_{B \to B'}$ instead of B', given the case that B is known to the onboard engine, it is also important that performing a behavior modification does not negatively affect task performance. Although the approach of how to perform a behavior update cannot guarantee that the execution of B' itself is not a source of failure, which is targeted in the next section, it has to ensure that the progress of switching from B to B' is only taking place when it will be successful. This involves a number of verification checks referring to the properties discussed in the previous section regarding consistency.

In summary, the workflow of performing a behavior update includes several steps, while it is an important aspect of these steps that they happen in parallel to the execution of the old behavior and, if the behavior update fails, the old behavior can keep executing without interruption. So, as a first step, B' has to be generated from $\Delta_{B \to B'}$ and the condition $\Delta_{B \to B'}(B) = B'$ has to be verified. Afterwards, it has to be verified that B' itself is a consistent behavior and that the modifications are according the definition in section 3.3. Finally, the pivot state $s_{piv}^{(i)}$ has to be determined and the conditions discussed in 3.3.1 have to be checked. Only if all of these steps have been successful, B is interrupted and substituted by B'.

3.4 Behavior Development

The process of development is an important factor when designing the behavior framework, but it is even more crucial when speaking of runtime modifications. When making modifications to an executed behavior, parts of the development process are mixed with execution. This has to be taken into account at a very early stage of designing the concept of the system because some of the design goals are opposed. For example, while controlling a behavior at runtime requires focus on the current state of execution, behavior development requires a complete overview over the whole structure of the state machine, regardless of the currently active state.

For this reason, two different views on a behavior are provided to the operator by the user interface. One view is focused on the currently active state and augmented with recent feedback of the executed behavior. It also provides an interface for sending the above mentioned commands to the behavior. The second view displays the whole state machine and enables modification of its structure and of the properties of each state. During execution, modification is not allowed if the above mentioned consistency requirements are not fulfilled. However, this view can always be used for getting an overview.

One implication of runtime modification is that in most of the cases, the operator also has to fulfill the role of a developer. Ideally, during execution, there is a person available who has experience with developing behaviors so that this person can take the part of implementing the desired changes. But this should not be required and thus development of behaviors or changes must be very easy. Furthermore, in order to avoid errors and to be efficient, since time is a critical factor during task execution, the development process has to be aided and automated as much as possible while still providing a high level of flexibility and allowing custom modifications.

In order to provide this required way of easy development, the user interface allows for modeling state machines and setting the properties of each state. On saving, source code is generated for whole model which can then be executed by the behavior engine based on *FlorBE* and *SMACH*. In addition, manual extensions of the generated code can be made and enable great flexibility for behavior development, especially when the development takes place before execution.

Required robustness regarding specification errors is another important aspect of modifying behaviors during execution. It is not sufficient to only have a behavior executive which can handle failure of a behavior without further negative consequences. The development of behavior modifications has to take into account possible sources of errors as well and to offer functionality for detecting and correcting them. Thus, a consistency check of a whole behavior is performed at least before it is saved, and partial checks, such as the validity of given parameter values, are performed as soon as possible.

3.5 State Constraints

Constraining state execution is a very helpful mechanism for making sure that a state is less prone to failure during its execution. Conditions, such as having certain data available or being in a stable pose for manipulation, can make sure that the prerequisites for primitives used by a state are fulfilled. In order to specify such conditions, each state implementation defines an individual set of preconditions, depending on the primitives it refers to or the functionality it offers.

Definition: Precondition. *Each state implementation defines a set of preconditions s_Φ. In order to enter a state $s^{(i)} \in S_{SM}$ during execution, each precondition $\phi \in s_\Phi^{(i)}$ has to evaluate to* `true`*. $\phi : \Sigma \longrightarrow \{\texttt{true}, \texttt{false}\}$ is a function applied to the current situation of the robot σ.*

In addition to the conditions under which a state can be executed, state constraints can be used as well in order to guarantee certain properties when a state finishes its execution. This extends the conditions required to be fulfilled for returning a specific outcome in a way that better takes into account the meaning expressed by the conditions instead of just checking specific values. Referring again to the example of grasping an object by closing the fingers of the robot, the conditions for triggering the outcome that indicates success consist of checking the value of how much the fingers are closed and which force is applied to the sensors. This is very specific to the way grasping is implemented. When using a different approach, such as a pneumatic gripper, in a second state, these conditions may be different, but their meaning is still the same. This meaning can be expressed using postconditions, stating that the object of interest is now attached to the hand, which is the same for both states.

Definition: Postcondition. *Each state implementation defines a set of postconditions $s_{\Psi,Oc}$. On returning one of its outcomes $oc^{(i)} \in s_{Oc}^{(i)}$, a state has to ensure that each postcondition $\psi_{oc} \in s_{\Psi,Oc}^{(i)}$ corresponding to the respective outcome is fulfilled. ψ_{oc} is a function with the same properties as ϕ.*

Based on these two definitions, a critical factor for being able to make use of the conditions is to find a way how to represent the current situation of the robot σ. σ needs to be represented in an abstract way, applicable for all different types of states and building a common basis to which states can refer. This includes choosing a suitable level of abstraction as well as having a way to measure the current value of each property included in σ. One way to get these values is relying on postconditions, since they define in which situation, regarding some properties, the robot is after returning a specific outcome. But these properties of the robot are only a subset of σ and may also, obviously, not only be defined by states, but by external events as well.

Although a complete definition of σ is not yet available, it can already be defined partially. One aspect taken into consideration when verifying the composition of states is availability of *userdata*. Defining the input keys $s_I^{(i)}$ of a state thus implicitly adds a set of preconditions to the state, expressing that all used keys have to be defined when possibly entering the state: $\phi_k(\sigma = D_{SM}) = \exists k_{O,sX} \in D_{SM} \mid k_{I,s} \overset{!}{=} k_{O,sX}$. In this case, σ is defined by the available userdata keys D_{SM} and $s^{(X)} \in P_r^{+(i)}$ refers to previous states where $P_r^{+(i)}$ is the transitive closure of $P_r^{(i)}$, all recursively defined predecessors of $s^{(i)}$. Available keys are defined by the implicit postconditions of the previous states, as well as a set of default *userdata* of the state machine.

4 Behavior Engine

The implementation of *FlexBE* is based on ROS and split into several ROS packages, according to the functionality of each component. An overview of the resulting structure is given in figure 4.1. Beneath the user interface, two more nodes are executed at the operator control station (OCS). `flexbe_mirror` corresponds to the concept discussed in chapter 3.2.1 and `flexbe_input` to the concepts of chapter 3.2.2. `flexbe_core` contains the core functionality of the *FlexBE* behavior engine. Communication between the OCS and the onboard engine uses ROS topics, transmitted via a communication bridge to connect the two remote systems. Implementation of the remote communication is not part of this thesis, however, minimal but sufficient data exchange is an important requirement for the engine.

As depicted in figure 4.1, `flexbe_behaviors` is also considered as an external node. This node basically connects the engine with available behaviors and thus needs to be defined along with the system for which *FlexBE* is used, depending on which behaviors are actually defined. `flexbe_atlas_states`, however, is not required and, in this system, extends the collection of common states in `flexbe_states` with robot-specific states.

The first part of this chapter provides a more detailed description of the relevant capabilities and implementation details of *FlorBE*, and discusses how they can be extended to realize the requirements of the new runtime-modifiable behavior engine. Subsequently, the overall implementation of the onboard behavior engine is presented, including the extensions made to the *SMACH* state machine interface. Finally, three aspects of the implementation are discussed in detail, such as how switching a behavior during runtime is implemented based on the concepts of chapter 3.3. The operator interface, although a very important aspect of the behavior engine, is not part of this chapter and, due to its complexity, discussed instead in the next chapter.

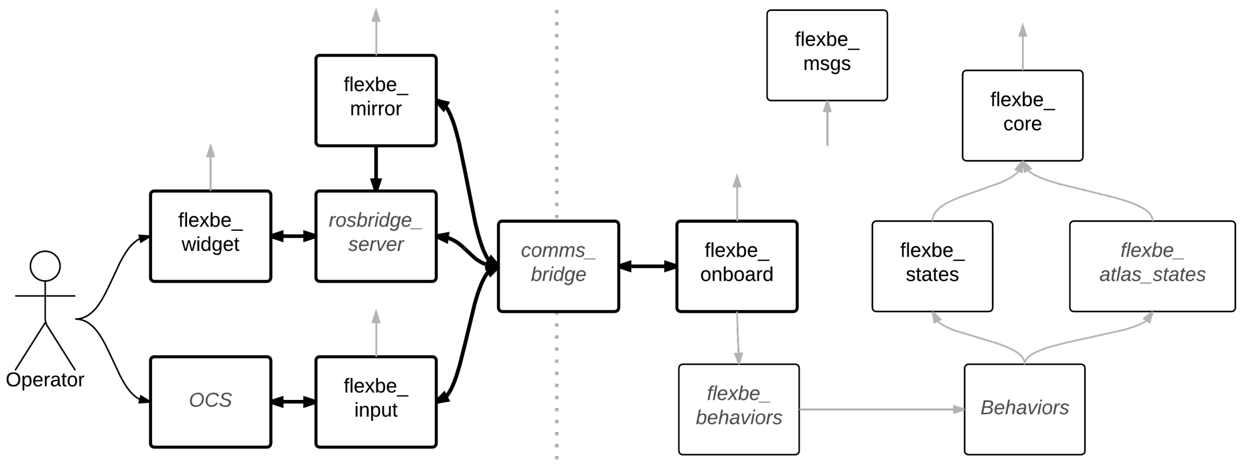


Figure 4.1: Component interaction in *FlexBE*. Italic font indicates external nodes, nodes with a bold border are executed as an own process each. Bold arrows refer to ROS topic-based communication, thin arrows are internal references.

4.1 Foundations

FlorBE [37] as a basis for the new behavior framework already provides a solid set of features and certain specific extension points. The concept of behavior specification, as defined by *FlorBE*, is also applicable for the newly developed framework and will therefore be presented in this section. Also, operator interaction with the behavior is done by relying on *FlorBE*. However, some specifics of operator interaction have to be extended or modified in order to enable more capable commands such as behavior modification. Furthermore, a new user interface will replace the behavior control widget and will extend the existing communication protocol, especially regarding robustness and feedback.

4.1.1 Behavior Specification

In *FlorBE*, behaviors are specified by two documents. The first one is the so-called *Behavior Manifest*. This manifest is represented by an XML document as shown in listing 4.1. and contains all meta information about the respective behavior, such as its unique name, its author, and the date of its creation. Furthermore, the manifest defines the interface of the respective behavior. The interface consists of two parts. The first part are the behavior parameters. Each parameter has a certain type, for example text, numeric, or boolean, and can be referenced by using its unique name. The values of parameters are chosen by the operator each time a behavior is started and then remain constant during execution. The second part is the declaration which other behaviors are included in the current behavior. For this purpose, their unique names are listed and, when starting a behavior, their respective parameters are loaded as well.

Finally, the manifest points to the corresponding behavior implementation, which is the second document. The behavior implementation is a Python class inheriting from the behavior superclass provided by *FlorBE* and can be located in an arbitrary Python module. Typically, each

```xml
<?xml version="1.0" encoding="UTF-8"?>

<behavior name="Get Up">
    <executable package_path="flor_behavior_get_up.get_up_sm" class="GetUpSM" />
    <author>Philipp Schillinger</author>
    <description>
        A behavior that lets the robot stand up when lying on the ground.
    </description>

    <params>
        <param type="numeric" name="timefactor" default="1.0" label="Timefactor (multiplier)" hint=
            "How fast a motion is executed.">
            <min value="0.1" />
            <max value="5.0" />
        </param>
    </params>

</behavior>
```

Listing 4.1: Example for a behavior manifest as used in *FlorBE*.

behavior implementation is placed in a separate ROS package. However, it is also possible to group multiple behaviors and only use a single ROS package for this group.

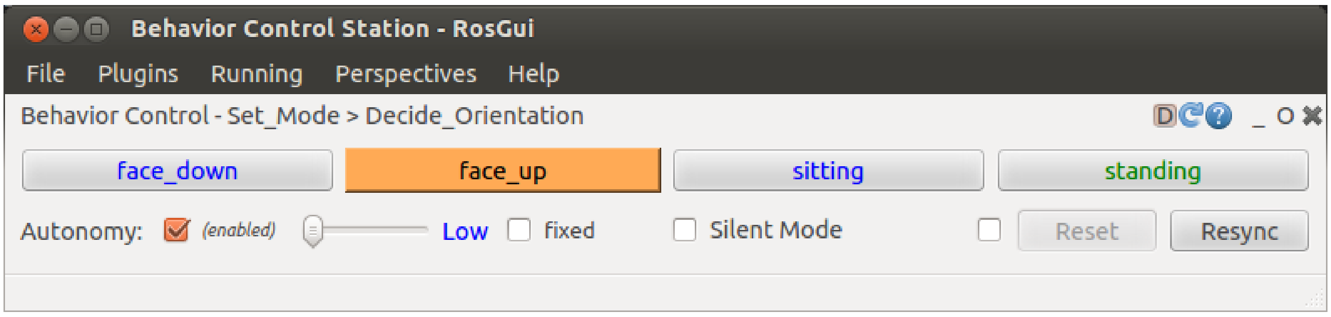
When selecting a behavior to be executed, the onboard engine will receive a message representing a start request and specifying the target behavior. This specification is done by providing the reference to the behavior implementation as given by the above mentioned manifest. The respective class is then imported, instantiated and executed. In addition, each start request defines values for the behavior parameters which then are used for execution. The core part of each behavior implementation is its state machine. Built upon *SMACH* [5], this state machine works according to the definitions made in section 3.1 and supports *userdata* for data exchange between single states as well as the *Autonomy Level* for regulating autonomous execution.

SMACH is extended to support all required features by an inheritance chain where each class of the chain adds a new basic feature to custom states and state machines, respectively. An overview of the old inheritance chain, as provided by *FlorBE* is given, in figure 4.4. This inheritance has been significantly extended, as presented in section 4.2.1. Also, although the concept of behavior specification is still applicable, creating and modifying these two documents manually does not meet the defined requirements of intuitive behavior definition and will thus be generated automatically from a model definition in *FlexBE*.

4.1.2 Operator Interaction

Operator interaction in *FlorBE* relied on a minimal widget for sending simple commands, such as forcing a transition or changing the *Autonomy Level*. It has mainly been created for expert users and provided no support to the operator for making the correct decisions. A screenshot of this behavior control widget is given in figure 4.2. What has proven to be helpful in the way this widget is realized is that it is highly context-aware when executing a behavior and only provides interaction options which are actually executable in the current situation.

Communication between the OCS and the onboard engine in order to send commands from the behavior control widget to the executed behaviors, was realized using ROS topics. Also, monitoring of the current state of the behavior has been realized this way, using a simple version of the *Behavior Mirror*, of which an improved version is presented later in this chapter. However, *FlorBE* did not enable to get direct feedback regarding the execution status of an issued command back to the operator control widget. This is an important aspect which has to be added in the improved version of the operator interface.


Behavior Control Station - RosGui

Behavior Control - Set_Mode > Decide_Orientation

Autonomy: ✓ (enabled) Low

Figure 4.2: The simple behavior control widget of *FlorBE*.

4.2 Onboard Executive

Based on the foundations provided by *FlorBE*, the onboard behavior executive is responsible for realizing all commands sent by the operator. Most important, on receiving a start command, it will prepare the specified behavior and start its execution. While most parts of the basic onboard executive can be used as available in *FlorBE*, this particular part of starting new behaviors has to be almost completely reworked. The reason for this is that now, behavior execution should not block all starting requests received in the meantime, but check instead if a switch to the newly received behavior is possible and perform this switch. In addition, robustness has been increased along with these extensions and more precise feedback during the whole process of behavior preparation is now available.

As depicted in figure 4.3, starting a new behavior is divided into several steps. Each step is contained in a *try/catch* block to detect and handle possible failure. Furthermore, each start request is processed in a separate background worker thread. In the worst case, this thread will fail to execute and thus simply drop the start request. The stability of the system, especially of other components, is not at all affected. The same subsequently applies for behavior execution itself, which is performed in the same background thread as starting. This approach provides more robustness against failure of behavior execution and has been added as part of the extensions to *FlorBE* for this thesis.

When a starting message is received, it will specify which behavior should be started and, if required, indicate changes to the available version of this behavior. Thus, the first step of processing a start request is to load the source code corresponding to the specified behavior, generate the code to be executed from the probably indicated changes, and store it in a temporary file where it incorporates a conventional class implementation. This Python class is then imported and instantiated in the second step. Since behaviors can also include other behaviors, the third step consists of importing all embedded behaviors and instantiating them as well. Fourth, the behavior parameters are set to their specified values for this execution. Finally, all states used by this behavior are instantiated and the behavior state machine is composed. After completion, the behavior is ready for execution and will be started if no other behavior is

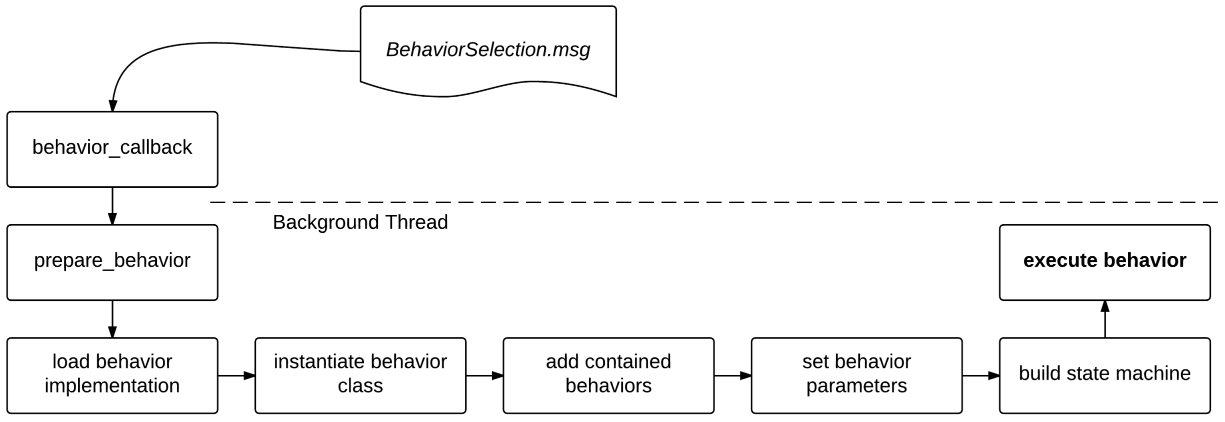


Figure 4.3: Process of starting a new behavior as specified by receiving a *BehaviorSelection* message.

```python
package = __import__("tmp_%d" % msg.behavior_id, fromlist=["tmp_%d" % msg.behavior_id])
clsmembers = inspect.getmembers(package,
    lambda member: inspect.isclass(member) and member.__module__ == package.__name__)
beclass = clsmembers[0][1]
be = beclass()

# ... execute behavior ...

del(sys.modules["tmp_%d" % be_id])
```

Listing 4.2: Runtime import of a behavior by the onboard engine as executed in step two of starting a new behavior.

currently running. What happens if there already is another behavior running is described in section 4.4.

Importing source code at runtime has already been discussed in chapter 2.3. Python enables this functionality by providing the command `__import__` to import a specific module when required. As shown in listing 7.1, a behavior is imported by providing a reference to the generated temporary source code file and can be instantiated afterwards. After having finished execution of a behavior, it is important to remove the imported module. Otherwise, Python would remember this module and would not execute the import command next time, while just using the remembered version and thus the old source code.

Regarding the previously described fourth step, setting the parameters of a behavior includes setting the parameters of all embedded behaviors as well. In order to avoid ambiguous parameter keys, the keys of embedded behaviors have their own namespace which corresponds to the path of the respective behavior inside the state machine of the top-level behavior. In the aforementioned third step of starting a behavior, a dictionary of all embedded behaviors is created. This dictionary maps the respective state path, and thus the parameter namespace to a certain behavior instantiation. Subsequently, in the fourth step, the key specification is split into the key name and its namespace. On this basis, the corresponding behavior instantiation is selected from the dictionary and its respective parameter, as referred by the key name, is set. If no namespace is specified for a key, it refers to a parameter of the top-level behavior.

A further extension provided by this thesis is a more precise and helpful operator feedback. The feedback has been improved in three ways. First, a status message always holds the current state of execution such as *running*, *finished*, or *failed*. This enables all related components such as the *Behavior Mirror* or the user interface to always reliably know if a remote behavior is running and adapt accordingly. Second, a heartbeat signal has been added to the onboard executive. This heartbeat enables to detect if a connection to the robot has been lost and will warn the operator in this case. Third, a new behavior logger will not only handle all logging which has to be done onboard, but also makes sure that the logged messages are transferred to the operator control station and will be displayed in the runtime control part of the user interface. As a result, the operator is immediately and precisely informed about any detected errors and also will have access to warning messages and more specific feedback than only state updates.

4.2.1 Inheritance Chain

Providing functionality to states is solved similar to *FlorBE* by using an inheritance chain for the state class. In order to implement the desired features according to the concept of this thesis, *FlexBE* extends the inheritance chain by adding new classes. Furthermore, the existing classes have been modified to give more precise feedback and to improve their modularity. In this regard, the most significant change is that previously, a state machine subscribed to all command topics such as the topic to preempt a state or manually trigger a transition, and when receiving a message on one of these topics, it has called the corresponding function of the state. This led to the drawback that a state could only provide its functionality if its containing state machine also knew about it and supported it. In the new approach, each state itself subscribes to its respectively required topic, using a cached proxy subscriber, and waits for commands independently. As a result, it is completely transparent to the state machine which functionality is offered by its states.

Figure 4.4 shows the previous execution flow of the old inheritance chain used by *FlorBE*. In contrast, 4.5 gives an overview of the new inheritance chain implemented for *FlexBE*. In the following, each of the classes in the inheritance chain is briefly described and its functionality is presented. Overall, the order of the states in the chain has been changed to better reflect the fact that a manually triggered transition rather substitutes the result of the user-defined condition checking, whereas a triggered preemption is a more important command and should be checked earlier. Improved modularity is another extension which applies to all states in general. The idea behind is that it should not make a difference to the state in which type of state machine they are embedded. In *FlorBE*, the states only had an interface for triggering their functionality via function calls, which needed to be coordinated by the parent state machine. Now, in *FlexBE*, states themselves listen to ROS topics matching their offered commands.

EventState

The *EventState* offers the ability to implement certain events during execution of state. Each state can specify code to be executed for example when a state is entered from another state by using the `on_enter` event. The *EventState* has already been existing in *FlorBE*. However, *FlexBE* offers the new events `on_start` and `on_stop`, which are executed respectively every time the complete behavior is started or stopped. Especially the `on_stop` event is helpful for making sure that all processes triggered by a certain state are correctly terminated, no matter how a behavior has been stopped. This helps for example in cases where one state starts a logging process while relying on the fact that a second state will stop it before the behavior finishes. But when the behavior is ended manually, the stop state may not be reached and therefore the starting state can now use the `on_stop` event to stop the logging process.

OperatableState

The *OperatableState* has been the most important state extension in *FlorBE*. It uses the concept of an *Autonomy Level* to decide if a transition is safe to be executed and, if not, prevents its execution. It also interfaces with the *OperatableStateMachine* for monitoring the state execution,

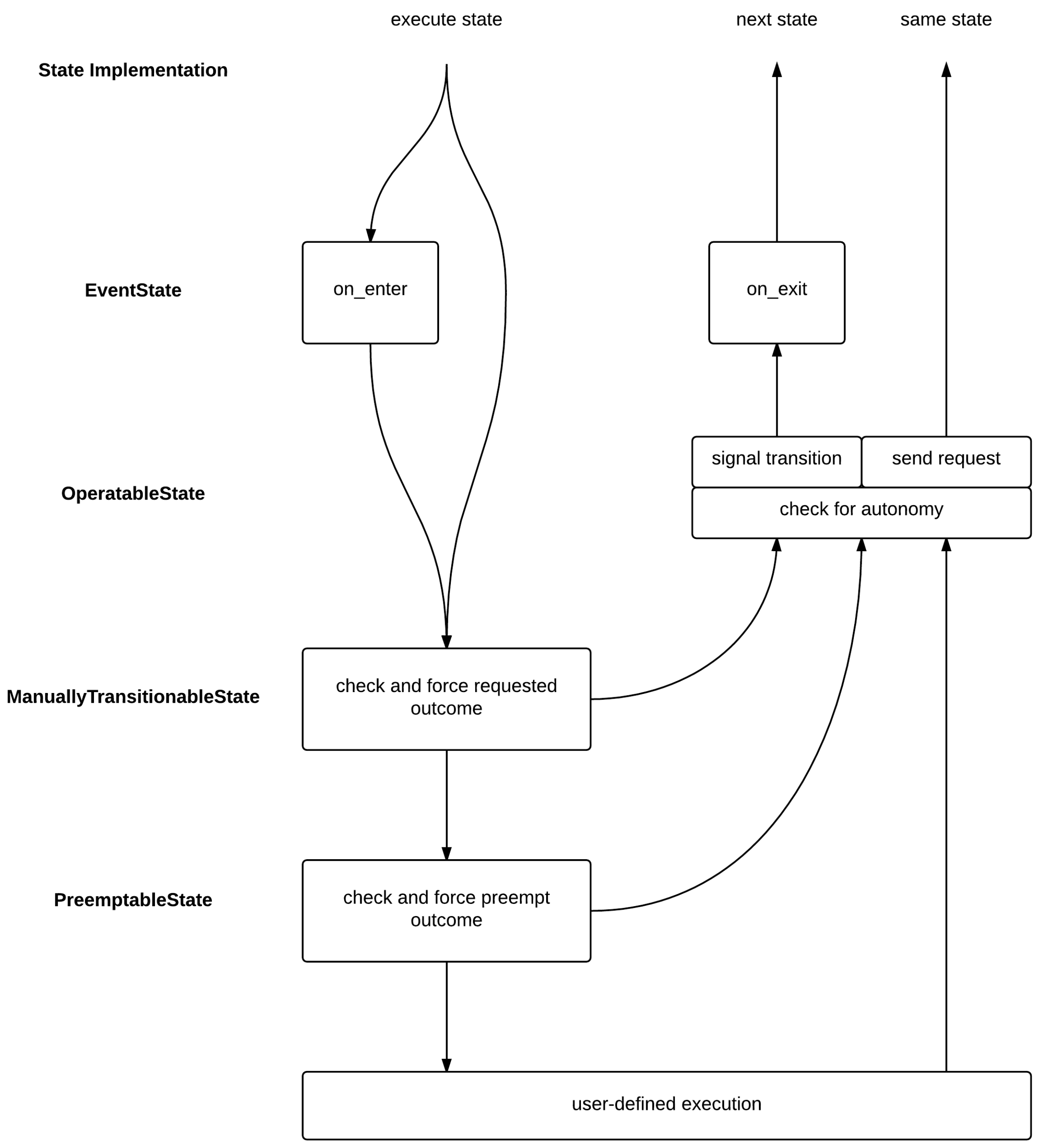


Figure 4.4: Flowchart of a state execution in *FlorBE*

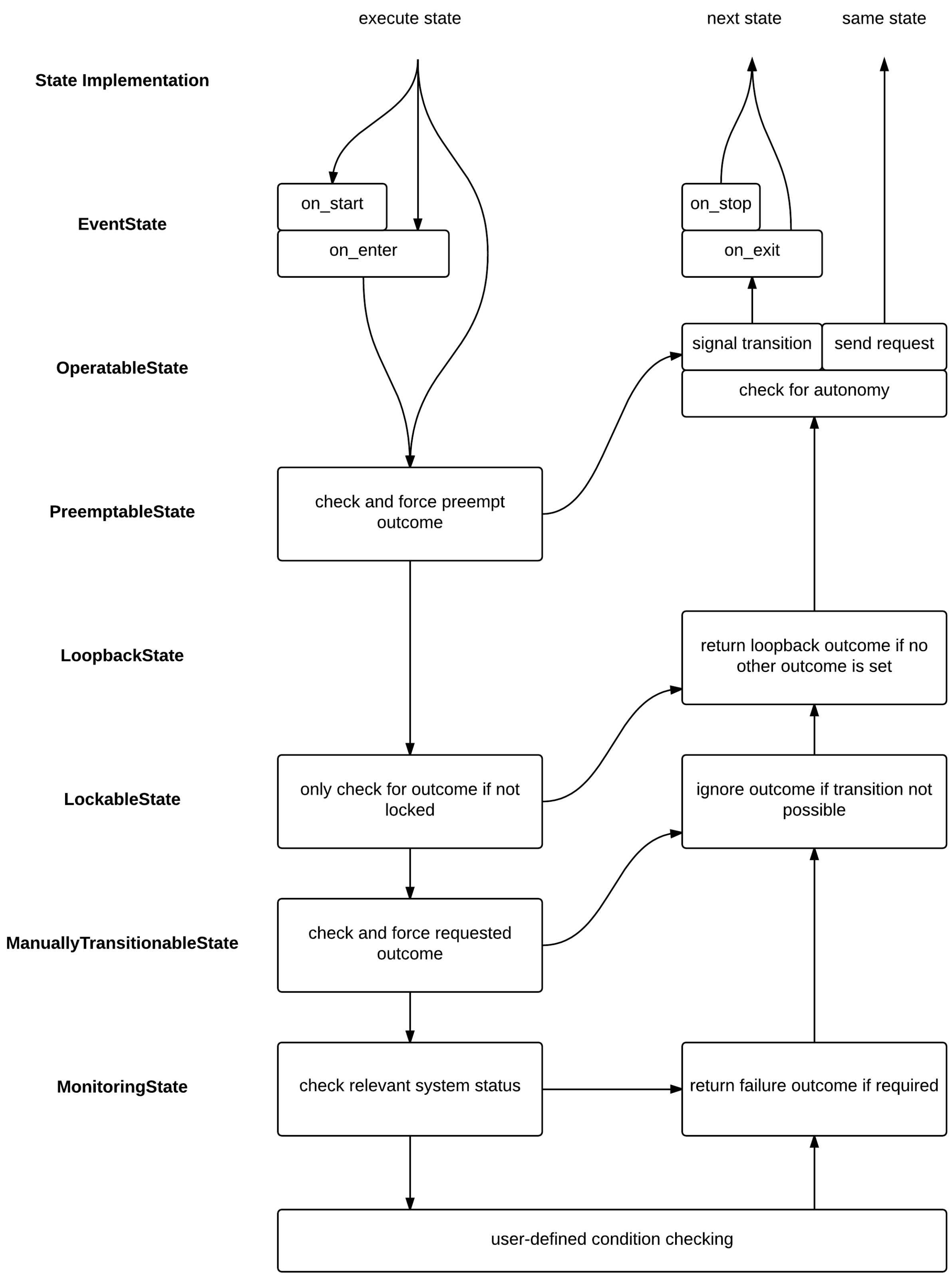

execute state
next state
same state
State Implementation
EventState
on_start
on_enter
on_stop
on_exit
OperatableState
signal transition
send request
check for autonomy
PreemptableState
check and force preempt outcome
LoopbackState
return loopback outcome if no other outcome is set
LockableState
only check for outcome if not locked
ignore outcome if transition not possible
ManuallyTransitionableState
check and force requested outcome
MonitoringState
check relevant system status
return failure outcome if required
user-defined condition checking

Figure 4.5: Flowchart of a state execution in *FlexBE*

although due to the modifications along with *FlexBE*, independent usage is possible. For this reason, the *OperatableState* offers an interface to generate an abstract definition of its properties in order to enable the *OperatableStateMachine* to generate a behavior description.

PreemptableState

The *PreemptableState* enables to preempt the currently executed state and, if combined with the *PreemptableStateMachine*, also the preemption of the whole behavior state machine. Preempting a state means that it will no longer check its conditions and if preempting a behavior, no new commands will be sent by this behavior. Execution of external components, which already has been triggered by the active state, cannot be canceled automatically when preempting a state. However, using the new `on_stop` event offered by the *EventState* in *FlexBE*, the developer of a state can ensure that also these actions are stopped, depending on the interface offered by the respective external component.

LoopbackState

The *LoopbackState* has been added by *FlexBE* to create a clean basis for using the typical pattern for defining states. In contrast to *SMACH*, state execution is non-blocking in *FlorBE* and *FlexBE*. After using *FlorBE* some time, an implementation pattern emerged where states typically command actions in their `on_enter` event and check if these have been finished in the periodical execution of the state afterwards, while always returning a special outcome when no condition was met. Using the new *LoopbackState*, states can just check for their outcome conditions and do not have to return any outcome if the state stays active. Also, the *LoopbackState* enables state condition checks to be executed with a certain defined rate.

LockableState

The *LockableState* is the most important state extension in *FlexBE*. It realizes the concept of locking a state, referring to the fact that this state still keeps executing, but is not able to transition to the next state if it has finished execution. If an outcome has been returned while being locked, the outcome is stored and will be returned as soon as the state is no longer being locked. Also, if used in combination with the *LockableStateMachine*, state machines can be locked as well. Since state machines are only containers and not executed themselves, the *LockableState* not only locks its container if required, it also makes sure that in this case no outcome is returned, which will lead to an outcome of its locked container, while outcomes which only lead to transitions inside the container are still allowed. This concept is visualized in figure 4.6.

ManuallyTransitionableState

The *ManuallyTransitionableState* enables to force particular outcomes of a state while it is being executed and waiting for its corresponding actions to complete or the required conditions to be fulfilled. This can be seen complimentary to the functionality offered by the *OperatableState*.

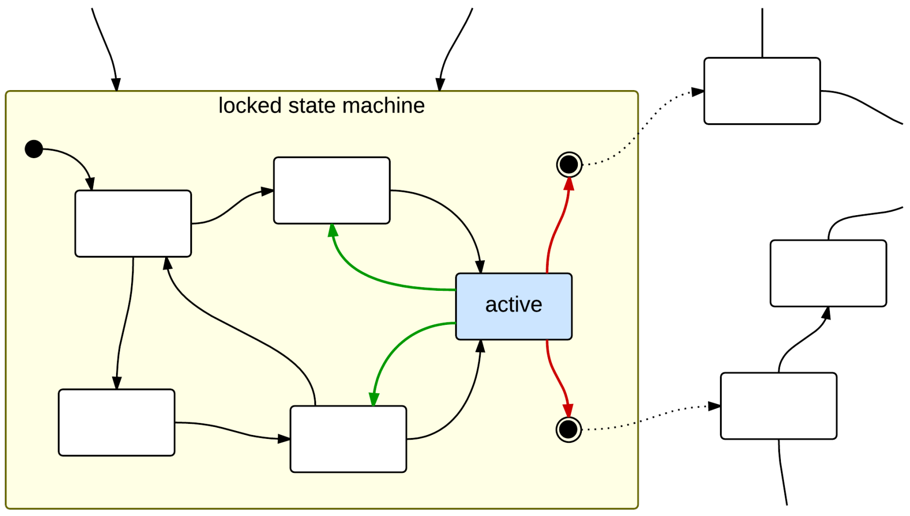


Figure 4.6: Example of allowed transitions inside a locked state machine as defined by the *LockableState*.

While the latter can block particular outcomes if their required autonomy is too high for the current level of autonomy of the robot, meaning the behavior wants to execute them but the operator does not, the *ManuallyTransitionableState* can trigger outcomes if the operator wants to execute them but the behavior does not.

MonitoringState

The *MonitoringState* has been added in *FlexBE* in order to increase the robustness of defined behaviors and to enable immediate reaction to external failure before it negatively influences execution. For this purpose, the *MonitoringState* defines a list of possible sources of failure which are monitored by an external health monitor based on the diagnostics component of ROS. Each state can then define the specific sources of failure which can influence its execution and as soon as one of these criteria reaches a critical level, the state will react to it by returning either an explicitly defined failure outcome, or one generated by the *MonitoringState* for this specific source of failure.

4.2.2 State Implementation

State implementation is realized using a certain pattern, mainly relying on the events offered by the *EventState* and the functionality added by the *LoopbackState*. In general, each state requires the implementation of two parts defining its execution and typically includes at least one more part. The first part is implemented in the constructor of a state. In this step of execution, all components and references later required for the actual state execution are created. This especially refers to required topic subscribers or *actionlib*[1] action clients. The second part is the periodic condition checking in the `execute` function of the state. In this part, typically no actions are executed or commands sent. Instead, it is determined if one of the outcomes of this state can be returned and, if applicable, output *userdata* of this state is written.

[1] `http://wiki.ros.org/actionlib` *(03/31/2015)*

Most of the implemented states trigger a certain action which is executed while the state is active. Commanding this action is done in an optional part, relying on the `on_enter` event defined by the *EventState*. This function is executed once when the state is entered and can be used for sending commands in order to start execution of the desired actions. Further available events can be used depending on the required functionality of the state. However, a developer should always keep in mind that a behavior can be preempted at any time and also outcomes, forcing the state to transition, can be commanded externally by the operator. Therefore, it is a good practice to consider using the events `on_exit` and `on_stop` for making sure to clean up after execution. An example state implementation is given in the appendix of this thesis.

Along with the development of the behavior engine itself, several states have been developed as well in order to offer solutions for common aspects of behavior creation and to extend the functionality offered by the pure engine. Such states include executing an arbitrary function callback as state, in order to convert certain input data to the required type of output data for future usage inside the behavior state machine. Function callbacks can also be used to check simple, very behavior specific conditions and trigger the corresponding outcome based on the result of this function. Additionally, simple but useful functionality such as waiting for a certain amount of time or sending a log message to the operator using the integrated *Behavior Logger* is available as state.

The *InputState* is a special state of the offered standard state implementations. This state is the counterpart to the `flexbe_input` node as presented in chapter 3.2.2. In its *on_enter* event, this state sends an input request to the operator specifying which type of input data is required by the behavior. Afterwards, it periodically checks if the operator has provided this required data, retrieves it from the serialized content of the message, and passes it as output *userdata* to the next states. The *ConcurrentState* is another special state. This state can be used to execute a set of states simultaneously and return an outcome based on their result and according to a specified evaluation rule.

4.3 Code Transfer

In *FlorBE*, specification of the behavior to start with has been realized by sending the unique name of a behavior within a start message. This allows for compact messages, but has the significant disadvantage that behaviors have to be known by the onboard engine. Therefore, only selected behaviors, which have been defined and implemented prior to starting a task and are therefore present onboard, could be selected. Since this thesis targets runtime modifications, it cannot necessarily be assumed that the implementation of a behavior is known in advance. However, it is also not desirable to transfer the full implementation of a behavior during runtime while probably being restricted to a slow communication channel.

When deploying the robot for starting to solve a task, the available behaviors can be synchronized between the onboard and control station. This provides a good basis, since changes made during runtime are typically rather small compared to the behaviors itself, which have been developed over a longer period of time with the available pre-knowledge of the task. If a behavior has not been modified since the last synchronization, it can still be referred instead of sending the full specification for starting it. Furthermore, when changes have been made, it is not necessary to transmit the whole new version. The changes only can be sent instead, and the new version can be constructed onboard based on the changes. For this reason, it is important for the control station to not only store the latest version of each behavior, but also to keep track of the version available onboard.

In order to provide the required source code for executing a behavior, the *FlexBE Editor* user interface generates it based on the created model. Details regarding this code generation process are presented in chapter 5.2.3. The generated code is then compared to the version of the respective behavior known to onboard and a patch according to chapter 3.3.2 is calculated for generating the new version by transmitting minimal data. The differences to be addressed by the patch are determined by using *difflib*[2], a standard Python library for comparing text. Based on this result, each difference is considered as a replacement, and the position, represented by the starting and the ending index, is used along with the content which is supposed to replace the previous content at this position. The patch is unidirectional, but it provides all information required in this context and contains no redundancy which would require additional bandwidth. In order to make sure that the correct version is constructed from the patch, a checksum of the initially generated source code is calculated and transmitted along with the patch.

When the behavior specification is received by the onboard engine, the onboard engine will load the referred stored behavior, apply the changes defined by the provided patch, and verify that the resulting behavior specification matches the desired one by calculating its checksum and comparing it with the one received. If these checksums match, the code transfer has been successful.

4.4 Behavior Switch

Switching a behavior is implemented very similar to starting one. As already mentioned in section 4.2, each behavior is executed in its own background worker thread. Thus, the intention to switch the currently running behavior to a new version, is indicated by just sending a start request containing the modifications to be applied. The way how a start request is realized and how the modifications are specified, has been discussed in the previous section. In case of a behavior switch the same format is sent, while the fact that the specified behavior matches the currently running behavior indicates that a switch will be performed.

When a start request for a new behavior version is received, the first steps of preparing the behavior are the same as described in section 4.2. They are completely independent from the currently running behavior. Once a behavior is ready to be started, which means all of its states have been instantiated and the state machine is composed, it checks if a behavior is already running. If there is one, the actual behavior switch is induced.

As shown by figure 4.7, a series of checks is performed first in order to verify if switching is possible. For example, it might be the case that the *preempt* command message sent to the previous behavior has been lost for some reason and the received behavior is actually not a new version of the running behavior, but a completely different one instead. In practice, the operator would not have received a confirmation for successful execution of the preempt command, and

[2] `https://docs.python.org/2/library/difflib.html` *(03/31/2015)*

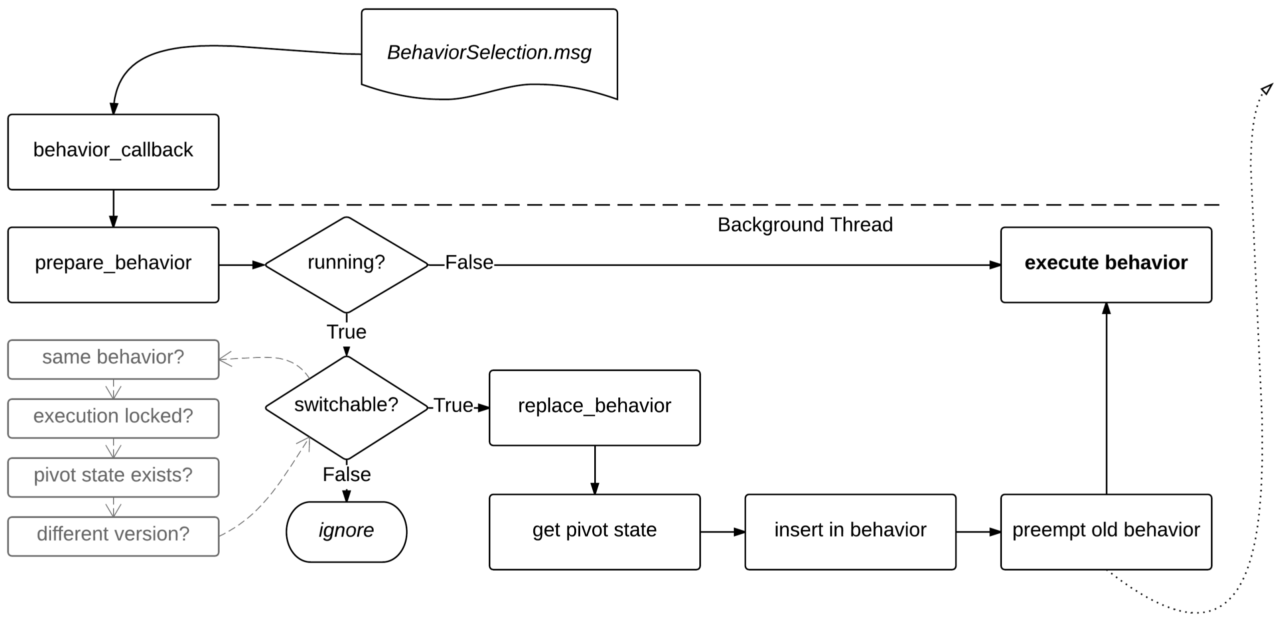


Figure 4.7: Process of switching to a new behavior. The old behavior keeps running independently until the new one can be reliably executed.

thus would have known that the message has been lost or at least not yet received. Furthermore, the user interface would not have allowed to start a new behavior while another one is still running. But however, things can always happen which require verification checks to be completely safe.

In order to decide whether the received behavior is actually a new version, the name of the specified behavior is checked first. The name of a behavior is unique, thus a specification using the same name has to be one for the same behavior. But describing the same behavior does not necessarily mean that a switch is possible. As discussed in chapter 3.3.1 regarding consistency, a behavior always has to be locked in its pivot state in order to allow for a switch. Therefore, the second check verifies that the behavior is actually locked, while the third check consists of finding the pivot state specified by state of the active behavior which is locked in the new version and checking if the requirement of immutability is fulfilled. Finally the version ID, representing a checksum of the source code, is compared to detect if the version is actually different.

After having passed all checks, the new behavior is ready to be started. In order to ensure a seamless switch, the pivot state of the switching process is integrated into the new version by replacing its corresponding state there. As a result, as soon as the pivot state is unlocked and takes one of the transitions, it will no longer transition inside the old behavior, but use the transition function of the new version instead and continue with one of the states of the new behavior. Finally, the old behavior is preempted. At this point, the behavior switch gets actually executed, because now the old version is not available anymore. Beforehand, if any step had failed, the new worker thread would just have been stopped and the old behavior would have kept executing. But as soon as the old behavior is preempted, the new version is ready to be unlocked and then able to continue execution of the task.

4.5 Monitoring

In order to provide comprehensive feedback about the current status of behavior execution, the onboard engine sends messages to the operator station whenever the active state of the behavior state machine changes. But instead of sending extensive messages containing information such as the active state, a *Behavior Mirror* is used for basically two purposes. First, it reduces the amount of data to be sent during execution to a minimum since it is able to apply knowledge regarding the structure of the behavior and to relay the update information by augmenting it with data regarding the name of the active state and available outcomes. Second, it allows for abstraction regarding the communication channel due to the fact that the mirror is executed at the operator control station, and thus enables other monitoring tools to directly connect to behavior status updates sent by the mirror. Further background regarding the concepts of the *Behavior Mirror* is provided in chapter 3.2.1.

The first information the mirror needs is the structure of the executed behavior. Therefore, when the user interface commands to start a new behavior at the robot and generates the code for execution, it also generates the structure for the *Behavior Mirror* in a special format which can directly be used to mirror the structure of the commanded onboard behavior. The structural data is provided as a list of state specifications, each containing the absolute path to this state inside the behavior, its outcomes, transitions, *Autonomy Level* and, in case the state is a state machine, its child states. Neither the class of a state nor its *userdata* keys have to be included, as they only refer to data processing and do not contribute to the structure. When receiving this structural message, the *Behavior Mirror* builds a state machine based on the provided structure.

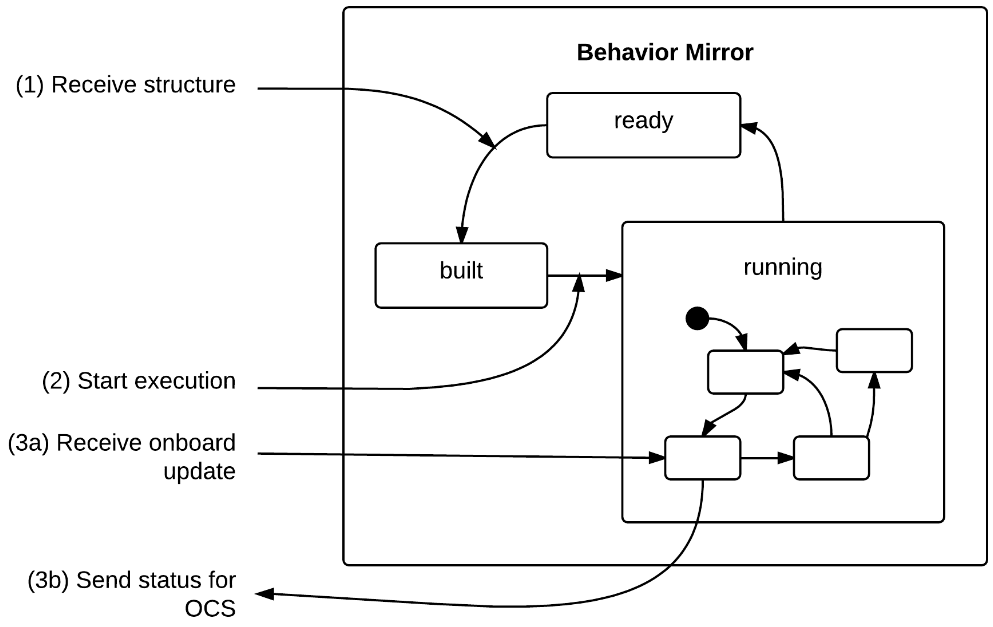


Figure 4.8: Simplified implementation overview of the *Behavior Mirror*: The mirror will be built after receiving a structural message and is executed when being notified that behavior is running. During execution, the single state instantiations trigger transitions based on updates and send OCS updates themselves.

All states are instantiations of a special mirror state implementation. As a result, they will send updates for monitoring instead of executing specific functionality.

In order to make sure that a structure actually corresponds to the executed behavior, the structural message as well contains the ID of the behavior as checksum. If the *Behavior Mirror* detects a difference between the ID used to create the structure and the ID corresponding the execution, it will drop its current structure and request a matching structure directly from onboard. This makes sure that a match is achieved. However, since requesting the structure from onboard has to use the possibly degraded communication channel to the robot, this is only a backup solution in case of a mismatch.

As depicted in figure 4.8, after receiving the structure of the target behavior from the control station (1) and being notified by the onboard engine that the matching behavior has been started (2), the *Behavior Mirror* starts execution of its internal state machine, representing the behavior structure, as well. Afterwards, during the whole behavior execution, updates are directly received by the single states of the mirrored state machine. Updates are in fact notifications about the index of the chosen outcome and are sent whenever an outcome is triggered at the topic `/flexbe/mirror/outcome`. In response, the states generate a specific update message and send it via the topic `/flexbe/mirror_update` to all monitoring tools. This solution is very efficient because update messages not only require knowledge about the current state, but also regarding its available outcomes and further specific information such as the *Autonomy Levels*. Handling these updates locally in corresponding states thus reduces complexity significantly.

5 Behavior Operation

The user interface of *FlexBE* is a very important part of the behavior framework. Not only does it allow to start and stop behavior execution, it also significantly facilitates development of new behaviors and provides access to all features offered by the onboard engine. In contrast to the previously described behavior engine, the software components presented in this section run at the operator control station (OCS). This includes the user interface, an input manager for providing specific data, and the already mentioned *Behavior Mirror* for monitoring.

The main focus of this chapter is the user interface and its capabilities to not only operate behaviors, but also to define them. Having these two capabilities realized in one single user interface is rather complex compared to having two independent interfaces. However, since the framework enables runtime modifications, having independent interfaces is not sufficient anymore. During execution, there are several restrictions regarding when and how a behavior can be modified. These restrictions ensure consistency and have already been discussed in chapter 3.3.1. Therefore, combining both aspects in one single user interface guarantees that all conditions for editing a behavior are met.

In this chapter, an overview of the approach of the user interface and its basis is given at first. Subsequently, the second section focuses on defining new and modifying existing behaviors and presents how these behaviors are provided to the onboard engine. This section is followed by a section focused on operating behaviors during runtime. Finally, new concepts that support communication with the onboard engine in addition to the *Behavior Mirror* are introduced.

5.1 Approach

The main purpose of the user interface is to provide the functionality offered by the onboard part of *FlexBE* to the operator in a user-friendly way. Since the concept of *FlexBE* relies on having the robot and the operator work together as a team, not only the robot has to be perfectly integrated into the system, but also the operator. Especially mechanisms to reduce the cognitive load the operator has to take when executing a behavior, and even more a developer has to take when specifying behaviors, are an important focus of the chosen approach for the user interface. For this reason, not only are state machines drawn in an intuitive way to quickly provide an overview of their structure, but also has a number of usability features been added such as visualizing the *userdata* dataflow inside a state machine, offering completion suggestions when entering parameter values or *userdata* keys, providing detailed documentation whenever possibly of advantage, or indicating errors as soon as they are made.

Although the whole system is based on ROS, the user interface itself is realized as a *Google Chrome App*. Main reason for this decision is the great flexibility to design an intuitive and useful interface for operating behaviors. The combination of *HTML*, *CSS*, and *JavaScript* allows to easily specify each single aspect of the user interface. In order to connect the user interface to ROS, *Rosbridge*[1] is used to communicate to the other ROS nodes and *RoslibJS*[2] provides a

[1] `http://wiki.ros.org/rosbridge_suite` *(03/31/2015)*
[2] `http://wiki.ros.org/roslibjs` *(03/31/2015)*

ROS interface to *JavaScript*. Furthermore, the framework *RaphaëlJS*[3] is used for drawing state machines.

5.1.1 Infrastructure

As being mainly a scripting language for augmenting websites with client-side functionality, *JavaScript* is not an intuitive choice for a large project like this. In order to enable development of software consisting of more than 10,000 lines of code, it is required to have a good infrastructure regarding abstraction and a clean interface to allow each component to interact with each other without being too specific.

One important concept missing in *JavaScript* is the ability to define classes. An object-oriented approach significantly reduces the complexity of large projects and therefore is desirable. But although *JavaScript* is based on functions instead of classes, it is possible to resemble classes. As shown by the example in listing 5.1, a state is defined as being a function itself. Private variables are simply variables inside the scope of this function while functions are one specific type of variables and can be locally defined in the same way as assigning a simple value to a variable. *JavaScript* allows to instantiate functions as objects by using the `new` keyword. In this context, the given function can be seen as a constructor for the new object. Having this in mind, public functions can be provided by extending the instantiated object. This is done by adding new attributes to the object using the `this` keyword. Again, there is no difference between having a simple variable or a function as attribute.

One thing to be noted in listing 5.1 is the second line. In *JavaScript*, the `this` keyword does not necessarily refer to the object of which a function is called. Instead, in most constellations, it rather refers to the calling context. For this reason, the second line is added as a convention to provide the functionality known from `this` by using the newly introduced keyword `that` for storing a reference to the object itself.

Inheritance in *JavaScript* is realized by a concept called *Prototyping*. Each object has a property called `protoype` which refers to another object. This other object basically represents the properties of the first object given by its superclass while the superclass is implicitly defined by the prototype object. As already mentioned, there are in fact no classes in *JavaScript*. For this reason, the prototype of an object is an object itself, not a class. However, when instantiating

```javascript
State = function(name) {
    var that = this;
    var state_name = name;

    this.getName = function() {
        return state_name;
    }
}
```

Listing 5.1: Example for a simple class definition in *JavaScript*.

[3] `http://raphaeljs.com/` *(03/31/2015)*

```
 1 Statemachine = function(name) {
 2     State.apply(this, [name]);
 3     var that = this;
 4     var states = [];
 5 
 6     this.addState = function(new_state) {
 7         return states.push(new_state);
 8     }
 9 }
10 Statemachine.prototype = Object.create(State.prototype);
```

Listing 5.2: Simple example for class inheritance in *JavaScript*.

a new object, the prototype of the constructor function is automatically copied as prototype of the new object.

An example for this process is given in listing 5.2. In line two, the properties of the prototype of the new object are set. This basically corresponds to calling the constructor of the superclass in conventional class-based inheritance. In addition, the prototype of the function `Statemachine`, which represents the class of all state machine objects in this example, has to be set to a new instance of the `State` prototype. This makes sure that each new state machine object will receive a state prototype.

In summary, applying the concept of object orientation to *JavaScript* is not always intuitive and sometimes hard to understand. But it is possible and when being used to it, the same patterns as in object-oriented programming can be realized in a similar way. Further explanations and details can be found at [41].

In addition to having a basis for implementing the desired concepts in *JavaScript*, it is required to split the whole user interface into separated blocks of functionality, but also to combine *HTML* and *CSS* with *JavaScript* as tools for specifying layout structure and appearance. All static parts of the user interface are defined in *HTML*, while there is a container with a unique ID for each spot where dynamic content has to be added later using *JavaScript*. In addition, *CSS* definitions specify the visual properties of all displayed objects and also provide pre-defined style classes for objects later created dynamically. Furthermore, a dedicated event coordinator realized in *JavaScript* directs events occurring at the static components defined in *HTML* to the modules implementing the respective functionality in *JavaScript*. The *JavaScript* part itself is split into several modules in order to reduce complexity and provide abstraction. In the following, each module is briefly described.

User Interface (UI)

This module is responsible for realizing the dynamic part of the user interface which cannot be defined in *HTML*. Its structure is closely oriented on the visual components. Each of the main views, which are described in detail in the next section, is represented by an own class. Additional panels, such as a panel for displaying the properties of a state, are realized in an additional sub-module.

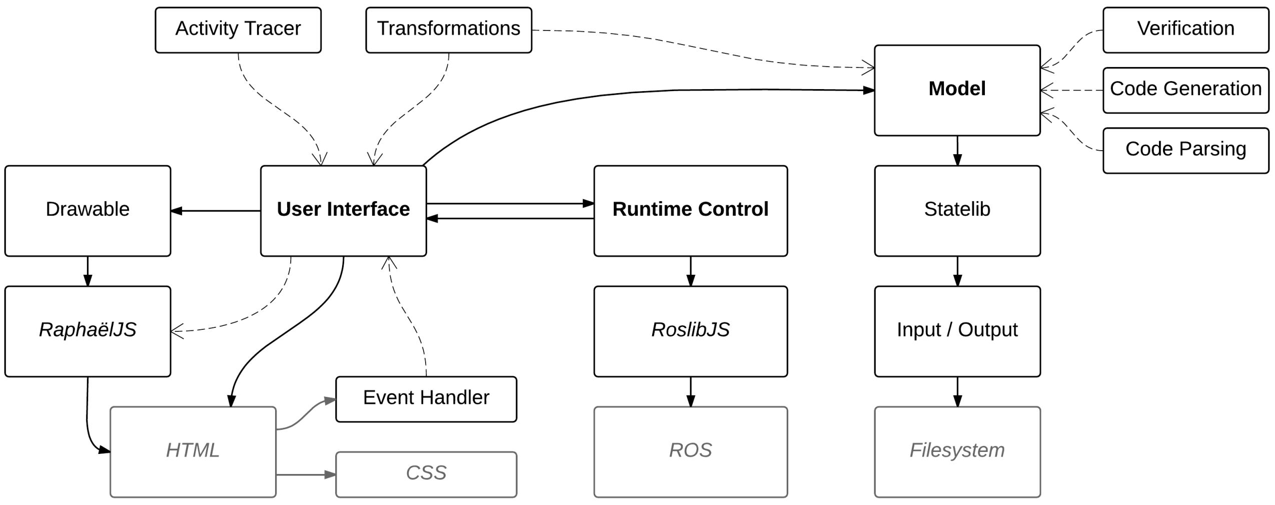


Figure 5.1: Component overview of the user interface of *FlexBE*.

Drawable

This module abstracts from the used external library for creating *SVG* drawings, *RaphaëlJS*, in order to provide drawable representations of objects. Using this module enables drawing a state or a transition instead of needing to compose rectangles, lines, and text boxes manually. In addition, it contains a helper class for providing ways of interaction to all drawn objects and offering features such as placing an object as close as possible to its target position without overlapping any other objects.

Input/Output (IO)

This module handles and abstracts access to the file system and offers functionality to load or save a whole behavior. In addition, only parts of a behavior, such as its manifest for accessing its interface, can be loaded using this module without the need for loading the whole behavior.

Statelib

This module handles parsing of single state implementations and generates abstract definitions for them. It manages all state definitions and provides functionality for accessing these, as well as accessing the definitions of behaviors when wanting to embed a behavior as state. The features of this module are presented in-depth in section 5.2.2.

Model

This module contains all classes used for modeling a behavior, such as `State`, `StateMachine`, and `Transition`. Having a model of the behavior represented by objects of these classes provides a lot of functions for accessing specific parts or getting certain information about it. One feature offered by a `StateMachine` object, for example, is generating a data flow graph for its content. Information on what exactly a data flow graph is, will be given in section 5.2.1.

Runtime Control (RC)

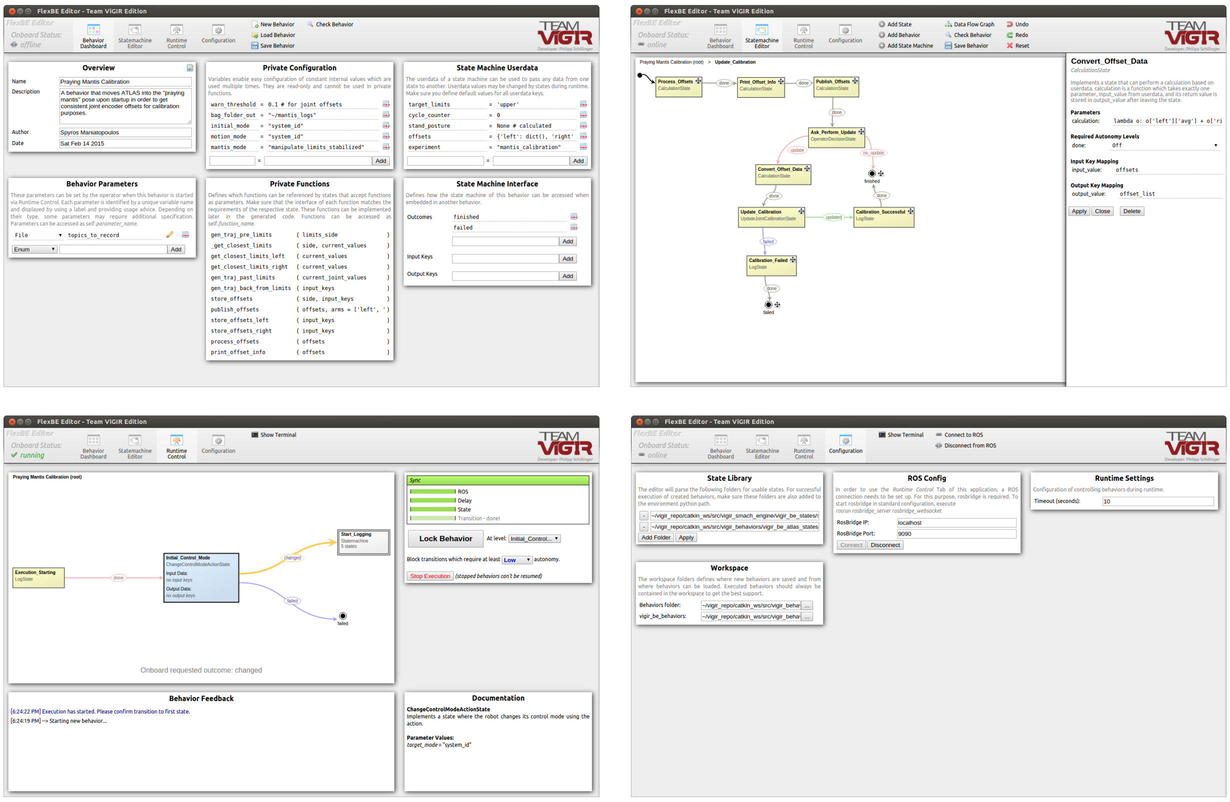
This module interfaces *ROS* by using *RoslibJS* and offers functionality to send and receive *ROS* messages. Furthermore, it contains the runtime controller class for managing the runtime state and deciding which actions are allowed with respect to a possibly running onboard behavior. Details to this module are given in all parts of section 5.3.

Finally, some top-level classes such as the previously mentioned event coordinator are contained in the root module, and further auxiliary modules containing *CSS* definitions, images and *JavaScript* code for testing, are available as well.

5.1.2 Operator Views

The user interface itself is separated into four different sections, called *views*. Each view has a clearly defined purpose and allows for concentrating on one specific aspect. The operator can switch between these views at any time using one of the four view buttons in the menu bar. In addition to the view buttons, the menu bar always indicates the connection status to the robot and contains further buttons defined by each of the views which are only displayed when the respective view is active.

The first view is the *Behavior Dashboard*. Its purpose is to give a high-level overview of the behavior and foremost define its interface. Thus, the view provided by the *Behavior Dashboard* can be seen as an outer view or *black box* view onto the behavior. Although not strictly divided



Overview

Name: Praying Mantis Calibration

Description: A behavior that moves ATLAS into the "praying mantis" pose upon startup in order to get consistent joint encoder offsets for calibration purposes.

Author: Spyros Maniatopoulos

Date: Sat Feb 14 2015

Private Configuration

Variables enable easy configuration of constant internal values which are used multiple times. They are read-only and cannot be used in private functions.

```
warn_threshold = 0.1 # for joint offsets
bag_folder_out = "~/mantis_logs"
initial_mode   = "system_id"
motion_mode    = "system_id"
mantis_mode    = "manipulate_limits_stabilized"
```

State Machine Userdata

The userdata of a state machine can be used to pass any data from one state to another. Userdata values may be changed by states during runtime. Make sure you define default values for all userdata keys.

```
target_limits  = 'upper'
cycle_counter  = 0
stand_posture  = None # calculated
offsets        = {'left': dict(), 'right'
experiment     = "mantis_calibration"
```

Behavior Parameters

These parameters can be set by the operator when this behavior is started via Runtime Control. Each parameter is identified by a unique variable name and displayed by using a label and providing usage advice. Depending on their type, some parameters may require additional specification. Parameters can be accessed as self.*parameter_name*.

```
File    topics_to_record
Enum
```

Private Functions

Defines which functions can be referenced by states that accept functions as parameters. Make sure that the interface of each function matches the requirements of the respective state. These functions can be implemented later in the generated code. Functions can be accessed as self.*function_name*.

```
gen_traj_pre_limits        ( limits_side                    )
_get_closest_limits        ( side, current_values           )
get_closest_limits_left    ( current_values                 )
get_closest_limits_right   ( current_values                 )
gen_traj_past_limits       ( current_joint_values           )
gen_traj_back_from_limits  ( input_keys                     )
store_offsets              ( side, input_keys               )
publish_offsets            ( offsets, arms = ['left', ')
store_offsets_left         ( input_keys                     )
store_offsets_right        ( input_keys                     )
process_offsets            ( offsets                        )
print_offset_info          ( offsets                        )
```

State Machine Interface

Defines how the state machine of this behavior can be accessed when embedded in another behavior.

Outcomes: finished, failed

Input Keys

Output Keys

Praying Mantis Calibration (root) > Update_Calibration

```
Process_Offsets (CalculationState) --done--> Print_Offset_Info (CalculationState) --done--> Publish_Offsets (CalculationState)
Publish_Offsets --done--> Ask_Perform_Update (OperatorDecisionState)
Ask_Perform_Update --update--> Convert_Offset_Data (CalculationState)
Ask_Perform_Update --no_update--> finished
Convert_Offset_Data --done--> Update_Calibration (UpdateJointCalibrationState)
Update_Calibration --updated--> Calibration_Successful (LogState) --done--> finished
Update_Calibration --failed--> Calibration_Failed (LogState) --done--> failed
```

Convert_Offset_Data

CalculationState

Implements a state that can perform a calculation based on userdata. calculation is a function which takes exactly one parameter, input_value from userdata, and its return value is stored in output_value after leaving the state.

Parameters
calculation: `lambda o: o['left']['avg'] + o['ri`

Required Autonomy Levels
done: `Off`

Input Key Mapping
input_value: `offsets`

Output Key Mapping
output_value: `offset_list`

Praying Mantis Calibration (root)

```
Execution_Starting (LogState) --done--> Initial_Control_Mode (ChangeControlModeActionState)
    Input Data: no input keys
    Output Data: no output keys
Initial_Control_Mode --changed--> Start_Logging (Statemachine, 5 states)
Initial_Control_Mode --failed--> failed
```

Onboard requested outcome: changed

Sync: ROS, Delay, State, Transition - done!

Lock Behavior At level: Initial_Control...

Block transitions which require at least Low autonomy.

Stop Execution *(stopped behaviors can't be resumed)*

Behavior Feedback

```
[6:24:22 PM] Execution has started. Please confirm transition to first state.
[6:24:19 PM] --> Starting new behavior...
```

Documentation

ChangeControlModeActionState
Implements a state where the robot changes its control mode using the action.

Parameter Values:
target_mode = "system_id"

State Library

The editor will parse the following folders for usable states. For successful execution of created behaviors, make sure these folders are also added to the environment python path.

```
~/vigir_repo/catkin_ws/src/vigir_smach_engine/vigir_be_states/s
~/vigir_repo/catkin_ws/src/vigir_behaviors/vigir_be_atlas_states
```

Add Folder Apply

Workspace

The workspace folders defines where new behaviors are saved and from where behaviors can be loaded. Executed behaviors should always be contained in the workspace to get the best support.

Behaviors folder: ~/vigir_repo/catkin_ws/src/vigir_beha

vigir_be_behaviors: ~/vigir_repo/catkin_ws/src/vigir_beha

ROS Config

In order to use the *Runtime Control* Tab of this application, a ROS connection needs to be set up. For this purpose, rosbridge is required. To start rosbridge in standard configuration, execute *rosrun rosbridge_server rosbridge_websocket*

RosBridge IP: localhost

RosBridge Port: 9090

Connect Disconnect

Runtime Settings

Configuration of controlling behaviors during runtime.

Timeout (seconds): 10

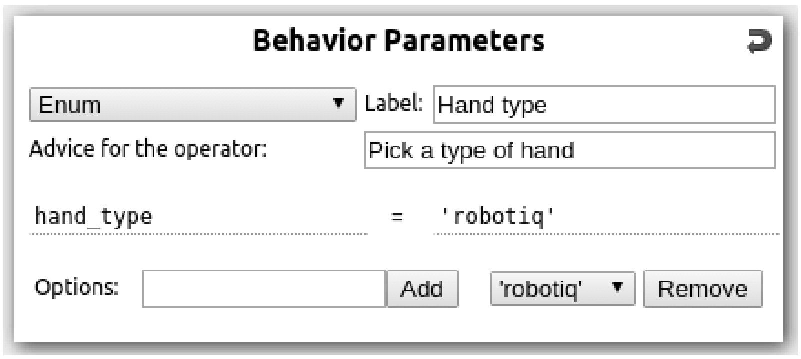
Figure 5.2: The four views of the *FlexBE* user interface.

regarding this aspect, the dashboard loosely corresponds to the *Behavior Manifest* document. Meta-data such as the name of the behavior and its author are specified here, as well as the parameters of the behavior. As depicted in figure 5.3, defining parameters for the behavior is assisted by a specific *Behavior Parameters* panel. Further panels at the dashboard help to add private variables, which can be seen as a persistent configuration of the behavior, default *userdata* and private functions. Also, the interface of this behavior, when embedded in another behavior, can be defined here.

The next two views, the *Statemachine Editor* and the *Runtime Control*, are presented in-depth in the next two sections. Roughly speaking, the *Statemachine Editor* complements the *Behavior Dashboard* when defining a behavior. But while the dashboard provides an outer view focused on the interface, the editor provides an inner view and defines the implementation of a behavior as given by its state machine. The *Runtime Control* on the other hand concentrates on providing a good monitoring of behavior execution and enables the operator to send commands to the executed behavior.

Finally, the fourth view offers the possibility to configure the *FlexBE* user interface itself. Settings and references to the file system can be specified here and are stored when closing the user interface. For example, the communication with *Rosbridge* can be configured here and folders for importing available state definitions can be chosen.

In addition to the views, there is a terminal panel, providing detailed feedback to the operator whenever commanded actions cannot be executed or relevant information is available. This terminal is hidden most of the time and only automatically displayed if required. Another addition is the *Tool Overlay*. As shown in figure 5.4, a set of commands is directly accessible using this overlay. This overlay can be displayed by pressing the *Control* and the *Space* keys. Moving the cursor across one of the commands before releasing the keys again will immediately trigger this command. Otherwise, the overlay stays displayed until a command is selected or it is hidden again. Besides accessing the most frequently used commands, this overlay lists all past modifications to the behavior and allows to jump back and forth in time to any of the listed versions. A command line input at the bottom also enables execution of more advanced commands.


Behavior Parameters

Enum | Label: Hand type

Advice for the operator: Pick a type of hand

```
hand_type = 'robotiq'
```

Options: | Add | 'robotiq' | Remove

Figure 5.3: Configuring the properties of a behavior parameter.

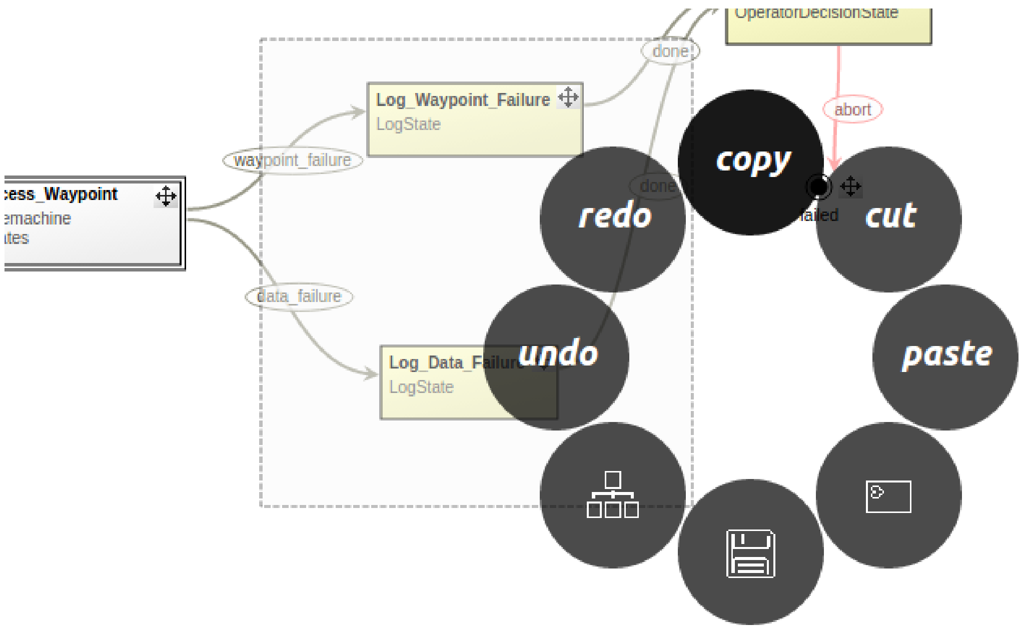


Figure 5.4: The *Tool Overlay* appears around the cursor position and offers quick commands.

5.2 Statemachine Editor

The *Statemachine Editor* view is a powerful tool for developing new behaviors or getting an overview of existing ones. It consists of a large drawing area where all states of the state machine are visualized as boxes and are connected by labeled arrows indicating existing transitions between them. The label of a transition corresponds to one of the outcomes of the outgoing state. When this state returns the specified outcome, this transition is taken and execution is continued with the state to which the transition arrow points. In addition, the arrows are colored corresponding to their required *Autonomy Level*. The outcomes of the containing state machine are drawn as terminating dots, labeled with their respective outcome. States as well can connect their outcomes to those dots which will make the state machine terminate when one of these outcomes is returned.

Figure 5.5 shows an example for a state machine visualized by the *Statemachine Editor*. As depicted, for each one of the yellow states, its state instantiation name and the name of the class of its state implementation is given. The white state is a state machine itself and contains three inner states. Although not shown in this simple example, whole behaviors as well can be embedded into a state machine as states, which is indicated by a purple state box giving the name of the respective behavior instead of the state implementation class.

Being able to develop behaviors in such an intuitive way without the need to specify each single state by writing Python code, and especially having an overview of the whole behavior state machine at any time during development, are important prerequisites for modifying behaviors during runtime. As stated in chapter 1.1, this feature implies strict requirements for the development of behaviors. Not only must the operator be able to make small modifications on its own without being a developer himself or being able to write Python code, but also the development of behavior modifications must be guided and supervised by auxiliary tools to verify the correctness of made changes.

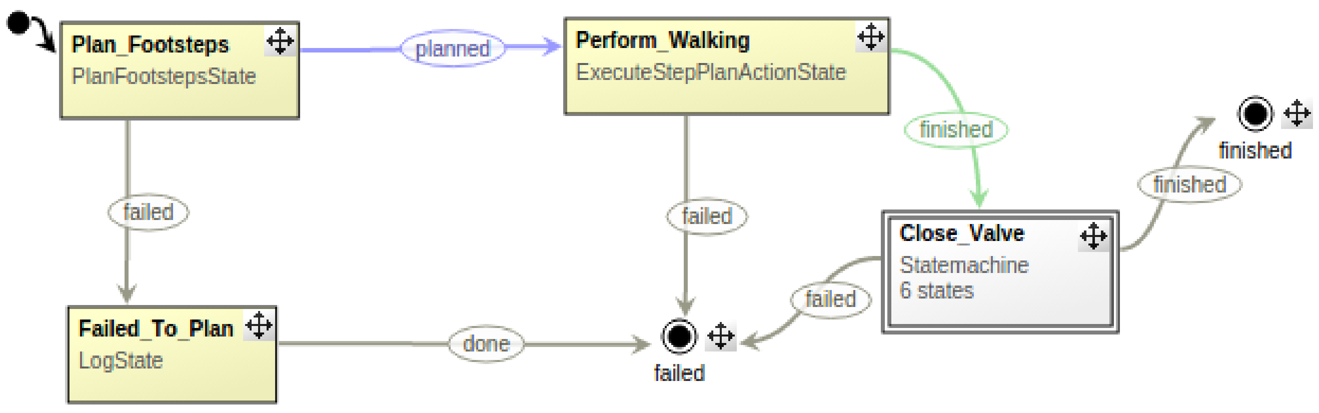


Figure 5.5: Example for a state machine visualized in the editor.

The *Statemachine Editor* enables to model the purpose of a behavior while the actual behavior implementation is generated automatically in the background according to the given specification. This guarantees the syntactical correctness of behaviors. However, the semantical correctness of a behavior as well is important, as, for example, requesting a wrong action at a time when it cannot be executed successfully might lead to task failure and should be prevented. As already discussed in chapter 2.2, verifying abstract high-level behaviors semantically is a very complex task and a large field of research itself. Nevertheless, some semantical verification checks are included as well in the editor in order to reduce the possible points of failure, but these are not yet sufficient to account for all possible sources of failure.

One semantical aspect which is checked, for example, is the availability of *userdata*. In the past, behaviors often failed in early testing runs because states referred to *userdata* keys which have not been initialized. This is likely to happen if there are different paths in a state machine and not all paths set the same *userdata*. This problem is now solved by verifying that no such path exists for any key.

A further aspect provided by the *Statemachine Editor* in addition to the robustness of developed behaviors is the efficiency of the development process. Modeling behaviors by using this editor is much faster and easier than writing code manually. Thus, development of new behaviors is not only significantly facilitated, but it is also ensured that required modifications during runtime can quickly be applied. This eliminates the need to interrupt the task execution too long, if at all, and reduces the risk of incidents while being busy with specifying behavior modifications during runtime, as discussed in chapter 1.1.

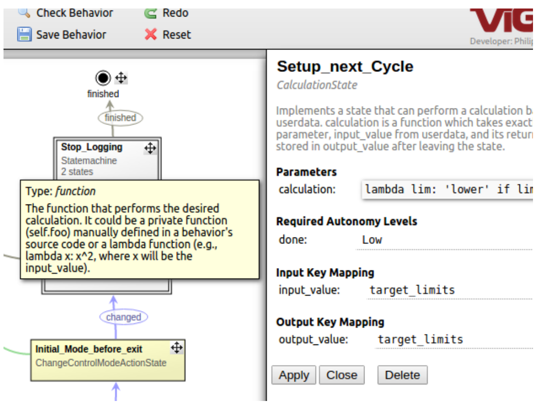
5.2.1 User Interaction

Intuitive user interaction is an important aspect of the *Statemachine Editor*. As already emphazised at the beginning of this section, state machines are visualized as a composition of state boxes and transition arrows. States can easily be moved and arranged in custom patterns by dragging them on the icon at their top right corner. Transitions can be set by clicking on them once to make them follow the cursor and then clicking on any possible target to connect them. When a state is newly added, it has no transitions. However, the possible outcomes of the states which are not yet connected by a transition, are displayed inside the state box. Clicking on one of these outcomes also triggers the process of connecting a transition.

Since state machines itself and other behaviors, which consist of at least one top-level state machine, can be used as states as well, they enable the specification of hierarchical state machines. In order to navigate this hierarchy, these states can be double-clicked to enter them. Entering means that now, their content is displayed in the editor drawing area. At the top of the drawing area is a navigation bar which allows to go up again in the hierarchy. While the content of an embedded state machine can be arbitrarily modified, embedded behaviors are read-only because they are not specified as part of the top-level behavior, but by their own behavior definition instead.

Furthermore, each state has specific properties, such as parameters. Single-clicking on a state opens the property panel of this state, displayed at the right side of the window. This panel enables the developer to change the name of a state and provides specific information by displaying the state documentation. Also, all properties of the selected state are listed and can be changed. This includes setting the parameters of this state as well as selecting the appropriate *Autonomy Levels* of all outcomes. Furthermore, the *userdata* keys used by this state can be remapped in order to match available keys of the state machine. This remapping allows to connect *userdata* of different states regardless of their internal key name. For state machines, also the available *userdata* keys and outcomes can be defined since they are not given by a fixed state implementation.

An example of how the properties panel looks like is given by figure 5.6. While, for example, entering the value of a parameter, as shown in this figure, the documentation of this specific parameter is displayed and also suggestions for possible values to be entered are provided, depending on available global variables, functions, and state specific data such as class variables of this state implementation. Providing this information is not only a convenience feature, it is an important part of the concept of facilitating the development as much as possible and of preventing the developer from entering wrong values. If, for example, a string was given as parameter value and the developer forgot to close the quotes, the input field would immediately turn red, indicating an error at this position.



Type: *function*

The function that performs the desired calculation. It could be a private function (self.foo) manually defined in a behavior's source code or a lambda function (e.g., lambda x: x^2, where x will be the input_value).

Setup_next_Cycle

CalculationState

Implements a state that can perform a calculation b
userdata. calculation is a function which takes exact
parameter, input_value from userdata, and its retur
stored in output_value after leaving the state.

Parameters

calculation: `lambda lim: 'lower' if lir`

Required Autonomy Levels

done: Low

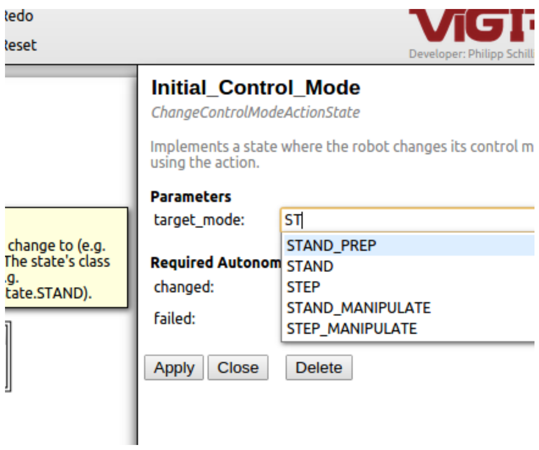
Input Key Mapping

input_value: `target_limits`

Output Key Mapping

output_value: `target_limits`

(a) Documentation tooltip



change to (e.g.
The state's class
.g.
tate.STAND).

Initial_Control_Mode

ChangeControlModeActionState

Implements a state where the robot changes its control m
using the action.

Parameters

target_mode: ST

STAND_PREP
STAND
STEP
STAND_MANIPULATE
STEP_MANIPULATE

Required Autonom

changed:

failed:

(b) Autocompletion

Figure 5.6: Documentation assists the developer by providing the correct values.

Adding a new state opens a panel similar to the state property panel. In this panel, the desired name of the new state can be given as well as the class name of the state implementation. In order to select the correct class and, at the same time, to get an overview of available choices, a list of all known state implementations is displayed in this panel. Each entry in this list not only contains the class name of this implementation, but also its respective summarized documentation. Selecting the right implementation from this list can take some time, therefore an input field for filtering the list is provided. When starting to type a desired class name into this field, only classes containing the entered text as part of their name are listed, while classes of which the name starts with the entered text are listed at the top. This enables a quick selection of the desired class, which is completed by clicking on the corresponding list entry in order to use this class. In case only a single list entry is left due to the entered filter, this class is selected automatically.

The data flow graph is an additional tool for supporting the development of new behaviors and especially helping to quickly understand what is happening in a state machine. When activating it, the data flow of the state machine's *userdata* is displayed instead of the labeled and colored transitions between the states. The data flow graph consists of directed edges between all pairs of states where the first state has an output key and the second one the corresponding input key. Or, practically speaking, it visualizes the flow of data from being written by one state to being read by another state. For the data flow graph, all possible paths are taken into consideration. If a *userdata* key is not defined by any state before being used as input key, an edge is added from the starting point of the state machine indicating that this data has to be provided by the state machine as an input key. Similarly, output keys of the state machine create data edges to the terminating dots representing the outcomes of this outer state machine.

Finally, also tools such as copy and paste of states and groups of states are provided. Selecting multiple states can be achieved by dragging a selection rectangle across all of them. When copying a group of states, internal transitions, which means that both end points of the respective transitions are contained in the selected group, are copied as well. All actions performed by the user are stored in an activity history, allowing to undo certain actions or redo them again. Furthermore, as presented in section 5.1.2, a *Tool Overlay* not only makes these actions more accessible, but also provides an overview of the whole activity history.

5.2.2 Integration of States

As already mentioned, a list of available state implementations is offered to the developer when adding a new state. In order to generate this list, the user can provide folders in the *Configuration* view which are then parsed for available state implementations. This seamless integration of the user interface into the development process is an important factor for enabling use in practice. It would be very cumbersome if a developer was required to provide information regarding an implemented state definition manually each time a new one is added. On the other hand, being able to parse available state implementations allows the developer to just open the editor and use all defined states without any intermediate step.

Parsing of state implementations is done by the *Statelib* component of the user interface. The `LibParser` recursively traverses all folders specified in the *Configuration* view and detects files

```python
class CalculationState(EventState):
    '''
    Implements a state that can perform a calculation based on userdata.
    calculation is a function which takes exactly one parameter, input_value from userdata,
    and its return value is stored in output_value after leaving the state.

    -- calculation  function    The function that performs the desired calculation.
                                You can provide either the reference to a function
                                or an anonymous function denoted by a lambda expression.

    ># input_value  object      Input to the calculation function.

    #> output_value object      The result of the calculation.

    <= done                     Indicates completion of the calculation.

    '''
```

Listing 5.3: Documentation pattern of a parsed state.

containing class definition which specify states. Theses classes are detected by checking if the class inherits from `EventState`, the endpoint of *FlexBE's* inheritance chain. Subsequently, the source code of the class is parsed in order to retrieve the relevant specification details of the state such as its outcomes, *userdata* keys, and parameters. Parsing is mostly done using regular expressions. However, in some cases where values need to be interpreted similarly as done by Python, the used regular expressions are supported by helper methods for interpretation. One example is a parameter with a given default value as string while the string contains a closing parenthesis. This parenthesis does not end the specification of parameters at this point and has to be ignored since it is only contained in a string and not considered by Python.

As already mentioned, detailed documentation of states as well as of their outcomes, *userdata* keys, and parameters plays an important role for supporting the developer with all required information. For this reason, a documentation scheme is introduced in order to provide this information. Listing 5.3 contains an example for such a documentation. At the beginning, an overview of the functionality and the purpose of the state itself is given. The first sentence is supposed to summarize it, the rest of this section should give details. Subsequently, outcomes, *userdata* keys, and parameters are documented in any order. Each line starts with one symbol, where -- indicates a parameter, ># an input key, #> an output key, and <= an outcome. Separated by at least one whitespace character follows the name to which the documentation refers, the expected type of data, and finally a detailed description which can consist of multiple lines. In the case of an outcome, the data type entry is skipped because outcomes do not contain any data. Empty lines, no matter where, are ignored.

When parsing a state, the `LibParser` not only stores information regarding the interface of the state, but also how to access the class which contains the state implementation. This information can be used later for generating required imports in the source code. Finally, all information regarding a state implementation is combined by creating one `StateDefinition` object for each state which manages access to all required information regarding this state. The classes `StatemachineDefition` and `BehaviorStateDefinition` extend the simple `StateDefinition` to provide additional information regarding embedded state machines and behaviors. However, state machine definitions are specified independently for each state machine and are not retrieved by parsing any source code. Nevertheless, in order to create the behavior state definitions, all available behaviors have to be parsed as well, including their manifest files. The single definition objects are then organized by the `Statelib` and the `Behaviorlib` class, respectively, where they can be accessed in order to instantiate states in the model.

5.2.3 Code Generation

When saving a behavior, source code is generated to represent an executable Python class implementing the modeled behavior. In order to only generate correct behaviors, a set of verification checks is executed before the code generation itself is started. These checks include simple facts such as if the name of the behavior is set or if all specified parameter names match the required Python syntax, and more advanced correlations, such as the behavior may only declare *userdata* input keys if they are also given a default value and thus are represented in the set of default *userdata* keys of this behavior. Also the state machine is verified which includes, for example, verifying that all outcomes of each state are connected to a valid target and that the local *userdata* keys of each state have a valid mapping. If the modeled behavior passes all verification checks, its source code can be generated.

The structure of a behavior class is very simple and structured since all of the common functionality is already offered by the `Behavior` super class. When a behavior is executed, the onboard engine calls the required methods of the `Behavior` class which then uses its `create` method to create the state machine. This `create` method is empty per default and can be overwritten by inheriting behaviors to specify their state machine. Furthermore, additional methods of the `Behavior` class can be called in the constructor method of the behavior for further configuration such as using behavior parameters and setting its name.

Inside the `create` method, states are declared by adding them to their containing state machine. Listing 5.4 gives an example for such a state declaration. The first argument is the state name, the second is the actual state object, being an instantiation of one of the available state implementation classes. Typically, states are not instantiated before passing them to this method call. This helps to have everything regarding a certain state in one place, as the arguments passed to the constructor call of the state class are the parameters of this state. The next three arguments embed the state into the state machine. The transition dictionary maps the outcome of the state to their corresponding targets. Each target has to be either the name of another state or one of the outcomes of the containing state machine. This dictionary is a typical source of failure when implementing behaviors manually, but since they are generated in *FlexBE*, this dictionary is verifiable correct. Next, the autonomy dictionary is similar to the

```python
OperatableStateMachine.add('Check_Control_Mode_Stand',
        CheckCurrentControlModeState(target_mode=CheckCurrentControlModeState.STAND, wait=False),
        transitions={'correct': 'Start_Logging', 'incorrect': 'Set_Control_Mode_To_Stand'},
        autonomy={'correct': Autonomy.High, 'incorrect': Autonomy.Low},
        remapping={'control_mode': 'control_mode'})
```

Listing 5.4: Declaration of a state inside a behavior definition.

transition dictionary but with the difference that an *Autonomy Level* for each outcome is given. Finally, the remapping dictionary maps internal *userdata* keys to external ones.

In order to generate the code, which is a sequential representation of the behavior state machine, the state machine has to be traversed in a certain order. Adding embedded state machines in the source code works similar to adding a normal state, but instead of providing a state instantiation, a reference to the embedded state machine is given. Thus, when traversing the hierarchy of state machines, the resulting order has to guarantee that a state machine is always at a position behind all of its embedded state machines, since Python would not know the reference to the respective state machine otherwise. For the states itself, no special order is required, since the transition function is given as a mapping of strings. Only exception is the fact that *SMACH* requires the initial state to be the first state added to a state machine per default. This constraint is also respected by the code generation process. In addition to generating the code to represent the behavior state machines, the imports for all used states are generated as well as declaring which external behaviors are contained.

The reduced flexibility constrained by the fixed pattern of generation typically is a significant drawback of code generation. However, this flexibility is often required for solving more complex tasks. Purpose of the code generation process is not enforcing a fixed structure of behaviors while restricting the developer to a certain pattern of implementation, but instead, supporting the developer by automatically providing certain parts of the implementation. Therefore, in addition to the generated code, the developer has the option to edit certain parts of the generated source code manually. These sections are explicitly declared in the generated code and when overwriting an implementation by a newly generated version, the manual sections will not get lost. This also includes the case where code in the manual sections is modified after a behavior has been loaded by the editor, thus allowing even parallel modification of source code in both, the user interface and manually.

There are four manual sections in each behavior. The first one, `Manual_Import`, is provided prior to the class definition of the behavior and is meant to be used for additional imports required by the other manual sections. The `Manual_Init` section is contained in the constructor of the behavior class and can be used for initialization code and declaring private class variables. The `Manual_Create` section is located inside the `create` method between the declaration of all private variables, as entered at the *Behavior Dashboard*, and starting to build the state machine. Thus, this section can be used for advanced calculation of private variables to be used by the state machine or to be passed as parameters to certain states. Finally, the `Manual_Functions` section can be used for manually defining additional helper functions which can be used in the other sections or passed to states.

Generating the source code of a behavior class is definitely the most complex part of the generation process, but not the only one. Each behavior implementation is represented by a *ROS* package and, in addition, needs to have a manifest located inside the *behaviors* folder of the *ROS* package *flexbe_behaviors*, while this package needs to have a dependency on the behavior package. All of these requirements are checked by *FlexBE* when generating a behavior and, if missing, are generated as well. This is not only convenient and makes creating new behaviors faster by being able to just click one single save button instead of creating all required files and entries additionally, it also ensures consistency and verifies that declarations regarding parameters and contained behaviors match each other in the source code and the behavior manifest.

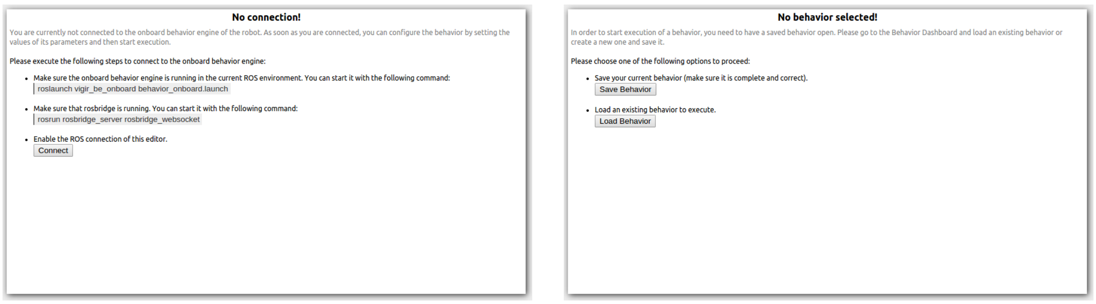
Unfortunately, arbitrary file system access is not provided for *Google Chrome Apps* in order to provide security against malicious apps. Instead, the app has to make an explicit request for each single file it wants to read or write, where the user has to specify the path, and will then provide an `Entry` object representing this file, but not offering much functionality. This limitation would completely inhibit the features described in the previous paragraph. However, the problem can be solved by using two additional aspects. First, as also used by the *Configuration* view, a *Chrome App* can store persistent data, which means that also the previously mentioned `Entry` objects can be stored in this way by converting them to a unique ID. Getting back an `Entry` object from such an ID does not require user interaction anymore. Thus, having once received permission for a specific file, omits the need for asking again for the same one. But the user would still need to specify each single file of the *ROS* package by himself and therefore would have to know the correct structure and provide it accurately. This problem is solved by the second aspect. Independently from *Chrome*, *HTML5* offers some functionality for accessing files and is able to handle `Entry` objects as provided by *Chrome*. This functionality includes getting a list of all contained files if the provided entry represents a folder, which allows to access subfolders and contained files. In combination, these aspects enable the user to only need a specification of a top-level folder once, referred to as the *workspace*, and being able to then read and write files inside this workspace afterwards automatically.

Although generating the files required for execution of the behavior by the onboard engine, no additional files for saving are generated. In order to load behaviors again, their manifest and source code are parsed to retrieve the structure and generate the model. This approach not only prevents redundancy, and thus has no problems with possible inconsistency of saved files and generated source code, but also allows to load manually written behavior code if it respects the structure pattern. However, strictly respecting this structure is not required. For example, when starting development of *FlexBE* and not yet having generated behaviors available, the behaviors which have been previously developed manually by using *FlorBE* have been loaded successfully.

5.3 Runtime Control

The *Runtime Control* view is the interface to the onboard executive of *FlexBE* while a behavior is running. It informs the operator about the current state of execution and enables to give commands to the remote behavior. In addition, the quality of synchronization is monitored at all times using the component `RC.Sync`, which is described in detail in section 5.3.2, and documentation of the active state is provided. When no behavior is running but the executive is ready to start one, this view lets the operator specify the parameters for execution and start the selected behavior. However, when no connection to the robot is available or the conditions for being able to start a behavior are not met, this view displays details regarding possible sources and suggests ways how to solve the problem, as shown in figure 5.7.

In order to provide this context-based user interaction, *Runtime Control* runs the component `RC.Controller` in the background which contains an event-based reactive state machine for keeping track of the connection status. As already discussed in chapter 2.1.1, there are two



No connection!

You are currently not connected to the onboard behavior engine of the robot. As soon as you are connected, you can configure the behavior by setting the values of its parameters and then start execution.

Please execute the following steps to connect to the onboard behavior engine:

- Make sure the onboard behavior engine is running in the current ROS environment. You can start it with the following command:
  ```
  roslaunch vigir_be_onboard behavior_onboard.launch
  ```
- Make sure that rosbridge is running. You can start it with the following command:
  ```
  rosrun rosbridge_server rosbridge_websocket
  ```
- Enable the ROS connection of this editor.

No behavior selected!

In order to start execution of a behavior, you need to have a saved behavior open. Please go to the Behavior Dashboard and load an existing behavior or create a new one and save it.

Please choose one of the following options to proceed:

- Save your current behavior (make sure it is complete and correct).
- Load an existing behavior to execute.

(a) No ROS connection　　**(b)** No behavior selected

Figure 5.7: No behavior can be started right now.

different approaches of how actions can be integrated into state machines. In contrast to the robot behaviors, which use a state-based action execution paradigm, the `RC.Controller` state machine executes its actions when doing the corresponding transitions. This is due to the fact that the actions triggered by this internal state machines are internal itself and not as likely to failure as physical actions performed by the robot. Furthermore, being able to transition to another state at any time a state is active, instead of waiting for a certain action to be completed as it would be required when a state corresponds to the action execution itself, enables to create reactive state machines of which the transitions can be triggered based on the occurrence of certain external events.

The structure of the internal `RC.Controller` state machine is depicted by figure 5.8. The upper four states represent that no behavior is running, or at least not known to be running. Requirements for being able to configure a behavior, and thus for being able to start its execution, are a connection to the robot and a saved, non-modified version of a behavior loaded. Only if both requirements are met, the operator is able to issue a start command. The lower four states correspond to the state of execution while a behavior is running. The states *LOCKED* and *NEW_VERSION* are very similar regarding the interaction they allow. But the difference between these two states is that taking the transition *unlocked* from state *LOCKED* simply results in unlocking the currently locked state, while taking this transition from *NEW_VERSION* includes performing a behavior switch.

The realization of this state machine is done in three parts. The first part consists of defining the states including the actions which have to be executed when transitioning from or to the respective state. Although the actions are specified at each state, they actually correspond to transitions since no actions are performed while a state is active. Typical actions executed in these functions would, for example, change the displayed content of the user interface. In the second part, transitions are specified as reactions to events. The signal *connected*, for example, will cause the state machine to transition to *NO_BEHAVIOR* when currently in state *NOTHING* or to transition to *CONFIGURATION* when currently in state *OFFLINE*. Signals are given as public functions and can be called by any other component being able to notice the corresponding event. Third, boolean expressions are used to evaluate the output of the state machine, which are functions of the status of the states. Short examples for each part are given in listing 5.5.

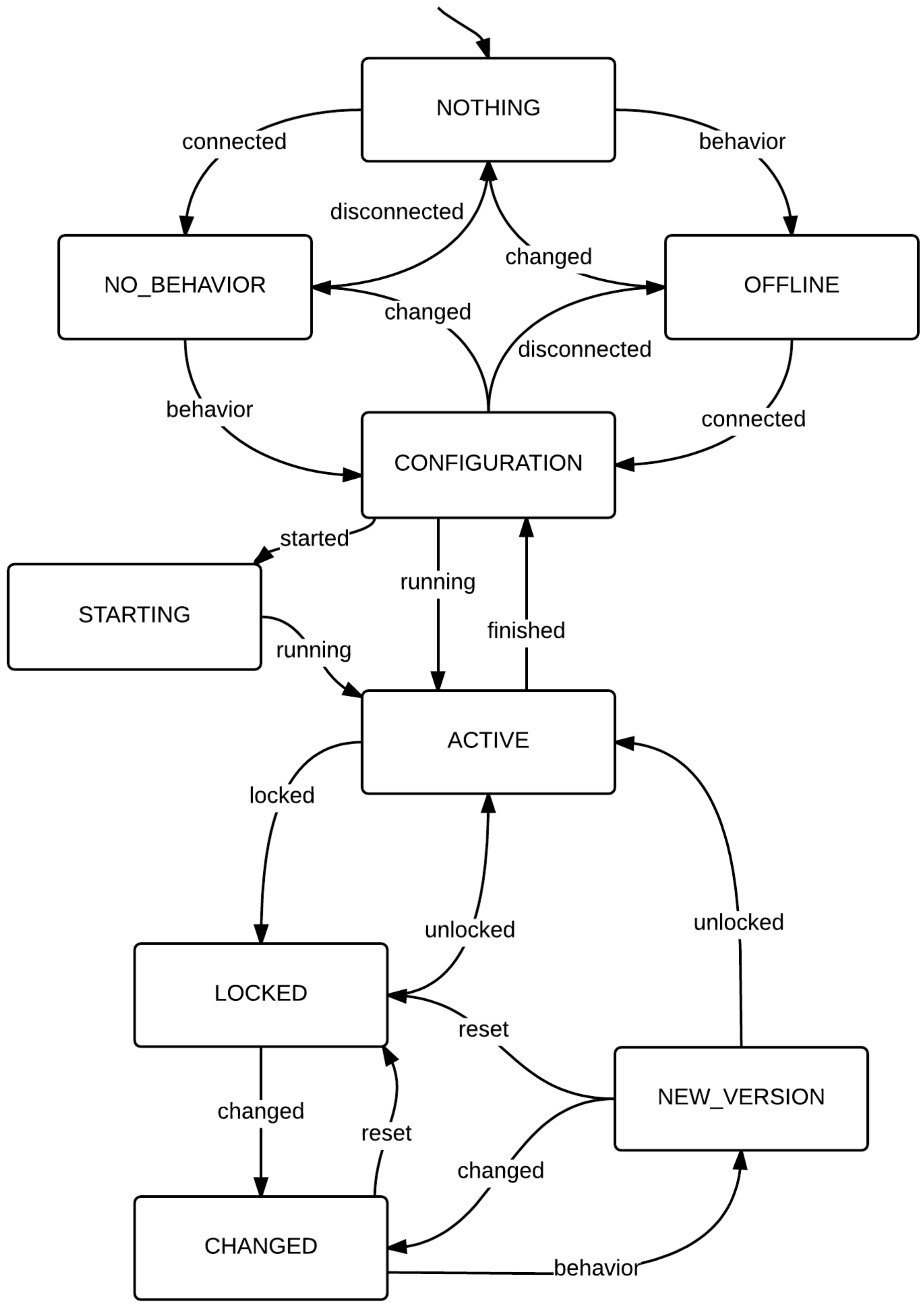


Figure 5.8: Overall structure of the `RC.Controller` state machine.

```
var STATE_NO_BEHAVIOR = { ... };
var STATE_CONFIGURATION = {
    onEnter: function() { ... },
    onExit: function() { ... },
    isActive: false,
    label: "STATE_CONFIGURATION"
};
var STATE_STARTING = { ... };

[...]

this.signalConnected = function() {
    hasTransition(STATE_NOTHING, STATE_NO_BEHAVIOR);
    hasTransition(STATE_OFFLINE, STATE_CONFIGURATION);
}
this.signalStarted = function() {
    hasTransition(STATE_CONFIGURATION, STATE_STARTING);
}

[...]

this.isConnected = function() {
    return !STATE_OFFLINE.isActive && !STATE_NOTHING.isActive;
}
this.isReadonly = function() {
    return STATE_ACTIVE.isActive || STATE_STARTING.isActive;
}
this.isRunning = function() { ... }
```

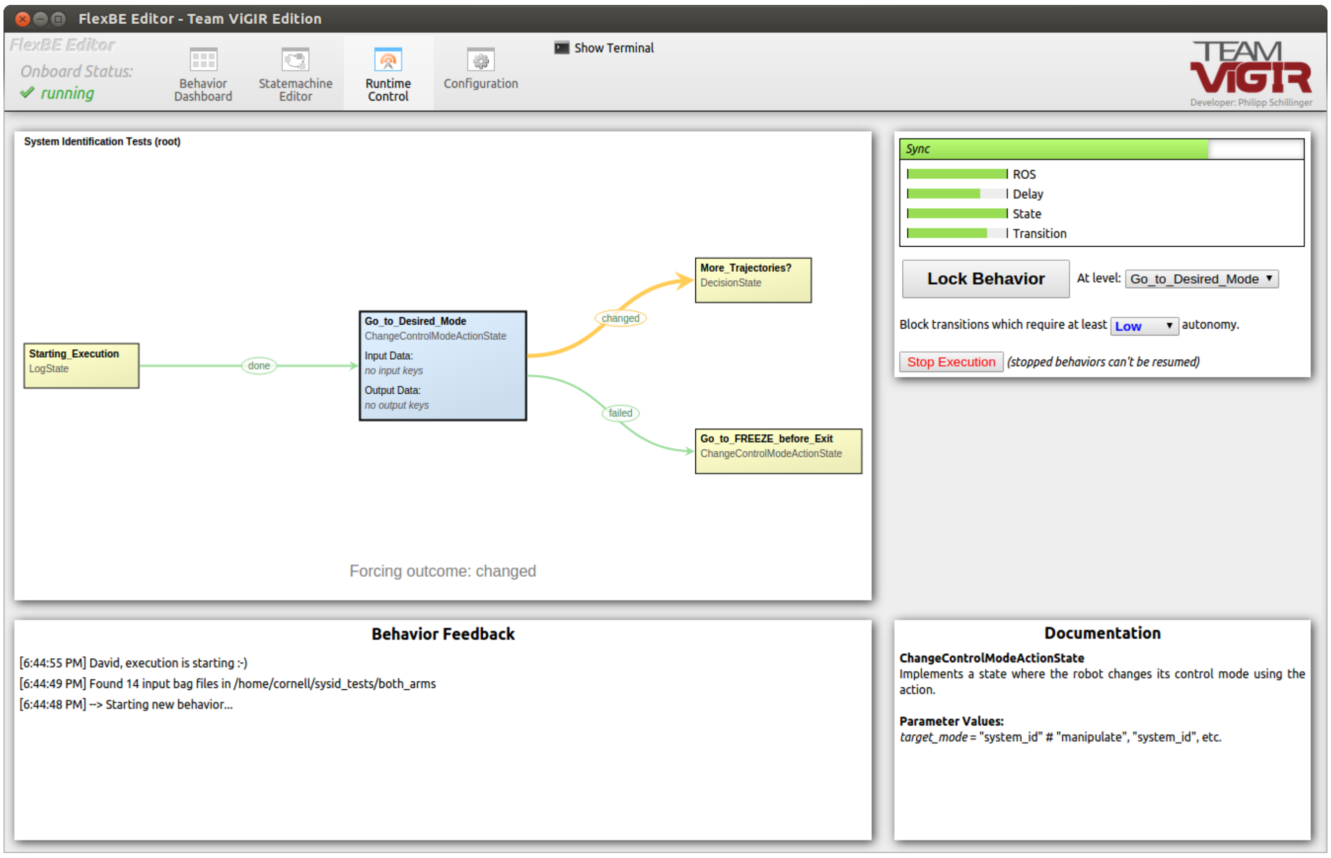
Listing 5.5: Excerpts of the `RC.Controller` state machine definition.

5.3.1 Control Interface

The interface for controlling a behavior during runtime consists of four different panels, an example is given with figure 5.9. The main panel always displays the currently active state in its center. If the active state is contained in an embedded state machine, the operator can switch to any state in the hierarchy path, enabling to choose between detailed and rather high-level feedback. Also, the mapping of *userdata* keys is given in order to let the operator know depending on which data this state executes. The way the states are displayed is the same as in the *Statemachine Editor*. Only difference is that the currently active state has a blue background. However, when switching to the editor during behavior execution, the currently active state is highlighted in blue as well.

To the left, the previous state is displayed including the transition that has been taken to reach the current state. To the right, the possible next states are given, again including the transitions that lead to them, labeled with the respective outcomes of the current state and colored regarding their required *Autonomy Level*. The operator can click one of the transitions in order to force it, causing the active state to being preempted and returning the regarding outcome. When the behavior tries to take a transition while being on an *Autonomy Level*, which is too low and therefore cannot execute it, the control interface will indicate this by highlighting the regarding transition and displaying a corresponding text information.

Below the main panel is the *Behavior Feedback* panel. This panel provides textual feedback given by the behavior engine. Single states can also utilize it for giving detailed feedback re-



At level: Go_to_Desired_Mode

Block transitions which require at least Low autonomy.

Stop Execution (stopped behaviors can't be resumed)

Behavior Feedback

```
[6:44:55 PM] David, execution is starting :-)
[6:44:49 PM] Found 14 input bag files in /home/cornell/sysid_tests/both_arms
[6:44:48 PM] --> Starting new behavior...
```

Documentation

ChangeControlModeActionState
Implements a state where the robot changes its control mode using the action.

Parameter Values:
target_mode = "system_id" # "manipulate", "system_id", etc.

Figure 5.9: Monitoring behavior execution in *Runtime Control*.

garding their execution by using the *Behavior Logger*. This logger not only prints the logged messages to the local terminal, but also sends it to the operator control station where they can be displayed at this panel. There are four different severity levels offered by the logger. The *information* level is meant to give general, rather positive, feedback. The *warning* level informs about possible problems which, however, appear to not directly influence the desired path of execution. In contrast, the *error* level gives feedback in case of failure. Typically, the *error* level should be mostly used by the behavior engine itself and not by states. Failing inside a state will return one of its outcomes such as *failed* or anything similar matching the failure. This does not necessarily imply failure of the behavior, thus a *warning*-level message is the most suitable way of informing the operator. The fourth level is the *hint* level, a special level of the *Behavior Logger*. This level is meant to give explicit textual requests to the operator, telling him what to do if the behavior expects a certain action done by him.

Next to the feedback panel is the documentation panel. It provides implicit information about the currently active state, such as its documentation for helping the operator to understand what exactly is supposed to happen when this state is executed. This supports the operator to determine himself if a state is executing correctly or likely to fail by supervising the robot. This panel also lists the specified parameter values of this state giving even more specific information regarding execution.

Finally, the control panel offers the operator to give commands affecting the behavior execution as a whole. For example, the operator can set the desired *Autonomy Level* for the behavior here. Also, execution of the behavior can be completely stopped in case something went wrong or the operator wants to switch to manual control from now on. However, stopping a behavior will completely end it, offering no way to resume execution at this point. When just wanting to pause behavior execution, the operator can lock the behavior instead. This will prevent the currently active state from transitioning, regardless of the *Autonomy Level*. Locking can not only be done at the state level, but also at any other level in the hierarchy. When locking a state machine, this means that the state machine itself keeps executing internally, but it will stop as soon as it is finished and will not return an outcome. Details regarding locking a state have been discussed in chapter 3.2.

While behavior execution is locked, the operator may modify the current behavior state machine. This can be done by switching to the *Statemachine Editor* view. Normally, when executing a behavior, the editor is in read-only mode, giving the operator an overview of the currently executed behavior, but not allowing him to perform any modifications. However, while a behavior is locked, its state machine and the properties of all of its states can be modified as if it was not running. As the only limitation, the locked state or state machine cannot be modified. After changing and saving the behavior, the operator can unlock execution again. In this case, unlocking will trigger the process of switching to the new version of this behavior. Details regarding this process have been explained in chapter 4.4.

In addition to offering the operator to give commands, the control panel also provides feedback regarding the quality of synchronization between the user interface and the onboard behavior executive. This synchronization is determined by the `RC.Sync` component of the user interface and affected by all commands given by the operator. The next section explains how

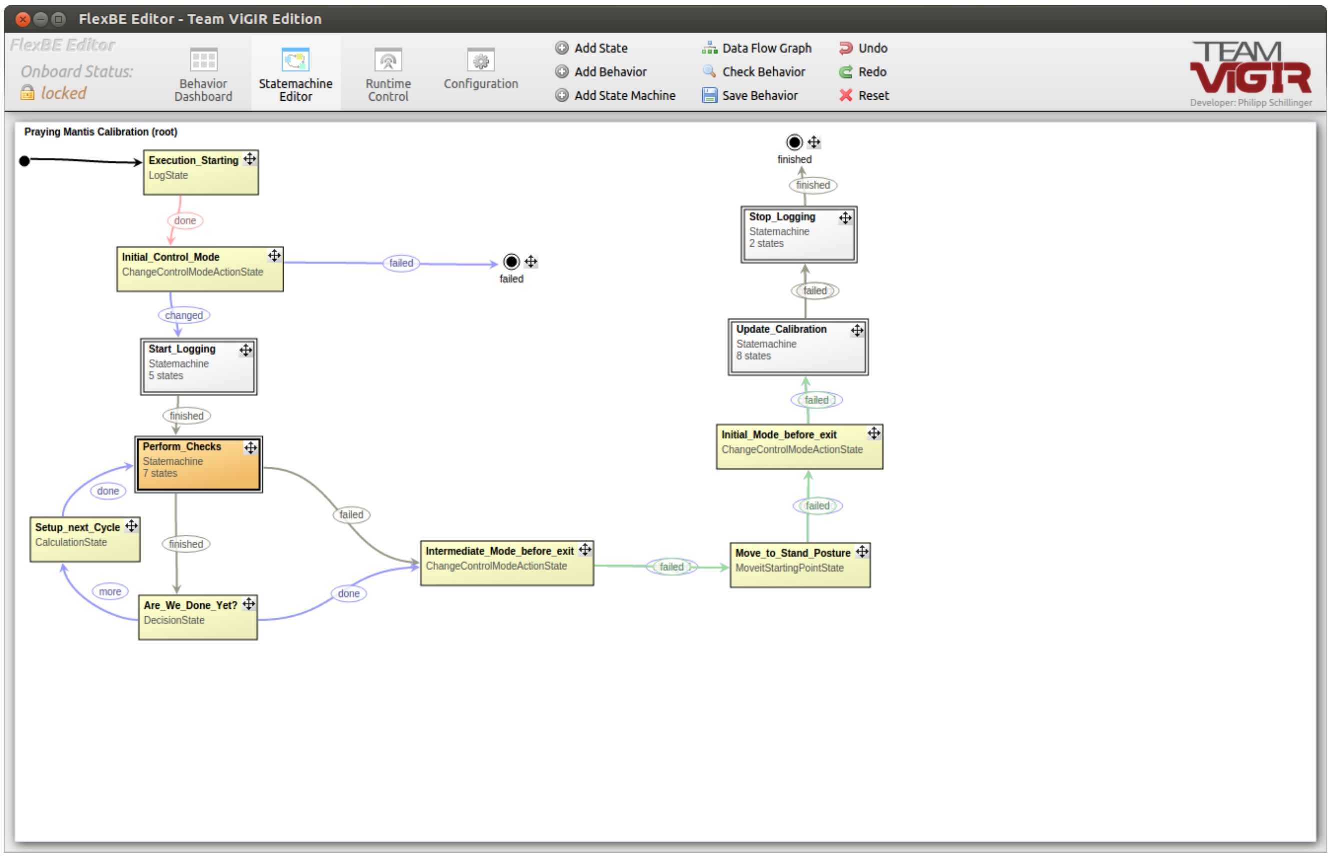


Figure 5.10: Behavior is running, but currently locked in the inner state machine *Perform Checks*.

`RC.Sync` exactly works. However, it finally provides a synchronization bar for the control panel which visually presents the quality of synchronization by being more or less filled.

5.3.2 Communication

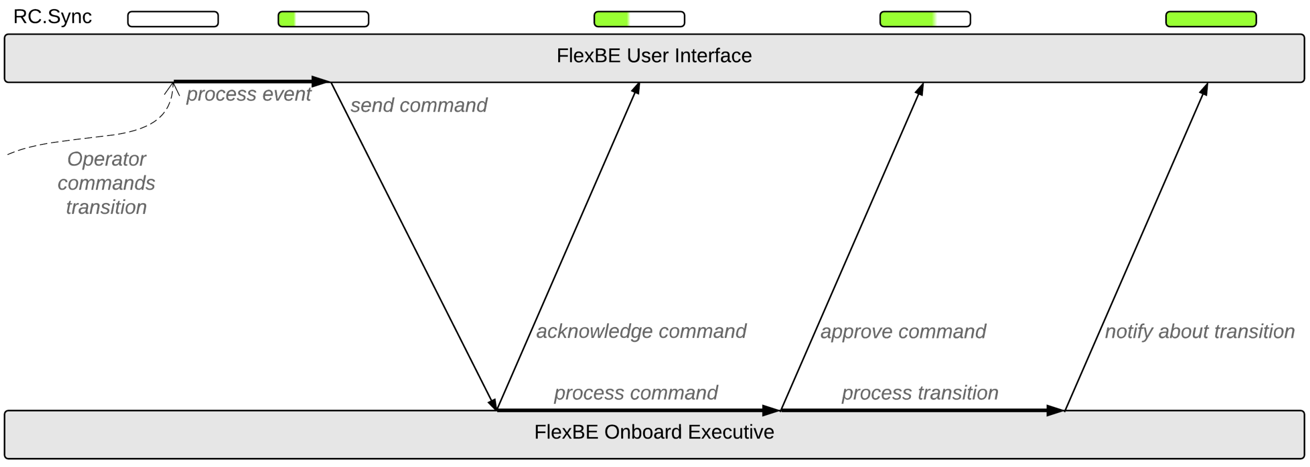
As already stated at the beginning of this chapter, communication with the onboard engine is done by using ROS. The class `RC.PubSub` interfaces with *RoslibJS* in order to send and receive messages on certain ROS topics. To the rest of the *JavaScript* code, it provides the functionality to send various commands. Also, when receiving certain messages, it calls the regarding functions, for example of the `RC.Controller` in order to signal events.

Reliability of the connection is an important factor for communication. In order to monitor the connection status, the onboard engine periodically sends a heartbeat message. `RC.PubSub` not only subscribes to this heartbeat topic in order to check if it can receive the messages, it also monitors the delay with which the single messages arrive. Based on these delays, the jitter of the connection can be calculated as an indicator for the connection reliability. In order to calculate latency, the heartbeat messages would need to have a timestamp of when they have been sent, and the clocks of both, the onboard executive and the operator control station, would need to be synchronized. If the connection reliability drops below a certain threshold and a disconnect is likely to happen, the operator will be warned in advance.

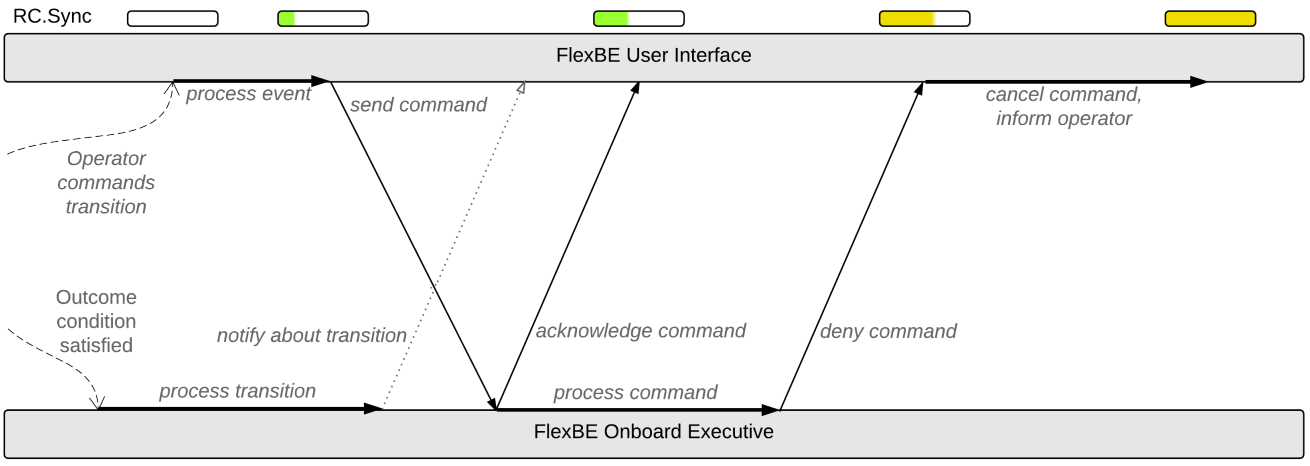
In addition to sending commands to the onboard engine, a communication protocol as well as evaluating feedback received from the onboard engine is used to determine if the command has been executed successfully, and to calculate the quality of synchronization between what is displayed in the user interface and what is actually happening onboard. For this purpose, `RC.Sync` offers functionality to register processes and keep track of their individual execution. Furthermore, it can be used to monitor the overall synchronization and status of the system by combining the corresponding values of the individual processes.

When a new process is registered to `RC.Sync`, it is given a unique key and a value specifying the impact of this process on the overall system. System synchronization is given in percent, so a value of 100 corresponds to full synchronization while 0 means a complete loss of synchronization. The impact of a process is thus given as any number between 0 and 100 corresponding to how much of the system's synchronization is lost while this process is not yet executed. The progress of this process is given as value between 0 (not started) to 1 (completed) and can be updated by any of the other components of the user interface such as `RC.PubSub` or `RC.Controller`. In addition, each process has a status which is able to indicate failure or certain problems.

The status of the system is given as the worst status of its set of processes. Calculating the overall system's synchronization is done by reducing the synchronization for the percentage of which each of its processes is not finished, weighted by their impact on the system. The summed impact of the set of processes does not have to equal 100. If the calculated synchronization value falls below zero, it is capped at zero. However, in order to achieve a synchronization above zero, the progress of all processes has to be high enough to be above zero. For example, two processes can be registered there. One having an impact value of 80 and a completion of 0 and another



(a) Successful execution of the transition.



(b) Failure due to external state change.

Figure 5.11: Communication sequence when executing a transition command. The progress bar at the top indicates the completion status of the corresponding `RC.Sync` process based on the received messages.

having an impact value of 60 and a completion of 0.5. Even though the second process is already completed half, the system's synchronization is calculated as -10 and thus set to 0. When the second process is completed, the system reaches a synchronization of 20, according to the calculation rule given above.

The output of `RC.Sync` is displayed in the control panel of the *Runtime Control* view of the user interface. This includes the synchronization value and status of the overall system as well as a list of the values of all registered individual processes, if desired. If, for example, the operator forces a transition manually, a new process corresponding this command is registered at *RC.Sync* and gives the operator feedback regarding its execution. After issuing the command, the synchronization bar will drop significantly because the commanded transition is not yet executed. But it fills again piecewise as soon as the command message is created and sent to the onboard engine, when the confirmation from the onboard engine that the command will be executed is received by `RC.PubSub`, and the process will finally be completed as soon as the resulting state is actually entered. In contrast, if the command fails to execute, for example due to the onboard engine already having executed a different transition and thus not being able to force the specified one, the status of the transition process of `RC.Sync` will be set accordingly and the process is completed unsuccessfully.

In order to keep `RC.Sync` updated accurately, a communication protocol is used for sending commands to the onboard engine, which requires the engine to acknowledge at least the receipt of the command and to indicate if there are any problems preventing the command from being executed. Depending on the command, intermediate feedback can be given regarding the progress of completion. This is especially helpful for more complex commands such as applying behavior modifications. All the feedback is sent on a dedicated command feedback topic and messages of this topic contain a specification regarding which command they provide feedback.

6 Evaluation

After having discussed the concept behind *FlexBE* and presented its implementation regarding both its onboard behavior engine and its user interface, this chapter now evaluates the results of this thesis. Since this is a valuable source of improving the initial concepts and potentially identify weaknesses of the current implementation, considerable effort has been taken to get helpful feedback and results.

Corresponding to the implementation being split into behavior execution and operator interaction, the evaluation as well has to consider both aspects. A good onboard engine is the prerequisite for efficiently executing behaviors, but without good operator integration and behavior modeling, the overall system performance will not pass a certain point. Similarly, good feedback to the operator and intuitive interaction with running behaviors is not sufficient if the engine is not robust enough to handle behavior failure or requires too much bandwidth in communication between robot and operator.

For this reason, the first section of this chapter will focus on evaluation regarding operator interaction based on practical experience using an early development version of *FlexBE* to develop and execute behaviors. The next section will then investigate behavior execution by using a dedicated test behavior suitable for measuring performance indicators. In the third section, the practical use of *FlexBE* in preparation for the *DARPA Robotics Challenge* is presented in order to give application examples. Finally, possible further work is discussed.

6.1 Operator Experience

Experience and feedback from the operator is an important source of evaluation. Since *FlexBE* has been designed with the intention to provide the operator an intuitive and easy-to-use high-level interface to the robot, a meaningful evaluation needs to involve the operator. A lot of research has already been done regarding how to evaluate human robot interaction. Steinfeld et al. give a broad overview in [40] and define several metrics while splitting the field of human robot interaction into five task categories. For the evaluation of this thesis, mainly *navigation*, *perception*, and *manipulation* are of interest.

In general, results of evaluating the user interface already in early stages of development showed that mainly two aspects of the concept positively shaped the experience of the users, both operators and developers. First, abstracting the robot's behavior by modeling it as state machines, where each state corresponds to a certain action of the robot and transitions refer to its results, turned out to be a very intuitive and broadly applicable concept. On the one hand, during execution, a operator always had a good insight of what a state means and what needed to be fulfilled in order to proceed. On the other hand, a developer could effectively compose a variety of actions and abstract from certain sequences of action by embedding sub-statemachines or other behaviors. Second, users said that the interface was easy to use and abstracted well from the complexity involved in each of the single actions. The next two sections provide a more detailed discussion of the early operator feedback.

6.1.1 Behavior Development

One main focus of the development of the *Statemachine Editor* was its usability. As a tool for creating high-level control, it should allow to work on top of an abstract basis and efficiently re-arrange the composition of low level components. Especially the *Tool Overlay* feature with its commands, such as copy and paste of whole groups of states including their intermediate connections, or access to the command history listing all past modifications to the behavior, has proven to be helpful for facilitating quick development and extension of state machines.

Furthermore, assistance to the developer while specifying the properties of a state instantiation, is important. The developer of a state implementation and the developer of a behavior using this state is typically not the same person. Therefore, in order to let the behavior developer specify the correct parameters and provide a suitable data format for the required *userdata*, good documentation of the interface is fundamental. The approach of *FlexBE* to closely integrate this documentation in the user interface and to display for the developer exactly the information he needs at each point in time, has turned out to be a crucial aspect for improving the specification of new behaviors. Also, suggestion of possible and available values for parameters and *userdata* keys helps a lot for passing the desired data to a state.

Checking if provided data, such as parameter values, is syntactically correct, is another helpful aspect of specifying the properties of a state. Instead of just checking the syntax of given values when finally generating code, checking the syntax as soon as a value is entered helps a lot because it allows the developer to immediately correct the mistake. In addition, the explicit verification of the whole behavior regarding consistency and correctness of the state composition has proven to be a helpful tool as well. This verification including some level of semantical verification in order to determine whether the developed specifications make sense. It not only improves the robustness of specified behaviors, but even more increases operator's confidence in new behaviors.

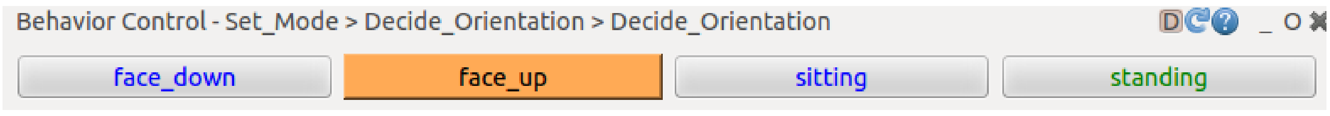
One aspect which may need improvement, is the specification of complex custom data structures as private variables and *userdata*. Passing a predefined data structure, such as a whole trajectory definition, is no problem since these structures can be described using ROS messages and are a common interface. But if the data structure is very specific to a single kind of application, custom Python dictionaries were often used in practical development in order to specify the structure, while afterwards certain states have been added to decompose the dictionary and extract the respective data to be passed to the states which execute actions. This approach is not very scalable because it is hard for the developer, and even harder for the operator, to understand which data is passed. However, the manual sections for combining the power of modeling behaviors with still being able to manually add parts of Python code, have proven themselves as a very useful concept. While the structural and compositional part of behavior specification was done by using the modeling tools provided by *FlexBE*'s *Statemachine Editor*, the manual sections were solely used for defining very specific data and there was no overhead related to the structure of behavior execution.

6.1.2 Behavior Execution

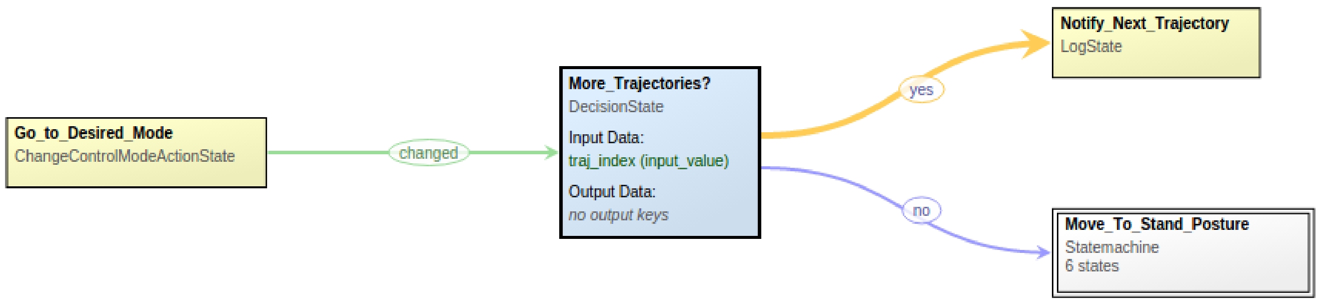
As shown by previous practical experience, a focus on the currently active state is very important when monitoring the execution of a behavior. For quickly understanding what the robot is currently doing, it is not sufficient to have a complete overview of the structure of the behavior state machine only, even when knowing which of the states is the active one. For this reason, the *FlexBE* has a special view, the *Runtime Control*, for monitoring behaviors during their execution. Similar to the previous user interface, the behavior control widget of *FlorBE*, the active state and its outcomes are displayed. One problematic aspect of the previous user interface, however, was the limited amount of provided context, such as the previous state or consequences of each displayed outcome. The new *Runtime Control* addresses this issue, as depicted in figure 6.1. This new approach has proven useful, along with the possibility to navigate in the hierarchy of the active state in order to view the current state of execution in different levels of abstraction.

The purpose of monitoring behavior execution is to give the operator the best possible estimation of what currently happens to the robot and what influences its next actions. Thus, it is not enough to just monitor the currently active state in a behavior. However, since a behavior relies on existing low-level components for realizing the commanded actions, it turned out to be a scalable approach to rely on these external components to define their own way of monitoring, instead of trying to integrate everything, regardless of what is actually used by a particular behavior. As a result, it is the goal of the behavior monitor to specifically monitor the control flow of a behavior, providing insights on which components are currently relevant and how they interact with each other.

The commands which are sent by the operator are also closely tied to the control flow of a behavior. Considering the circumstances of execution, especially the limited bandwidth and delays causing a highly constrained communication, commands cannot be executed by the robot immediately when commanded. Therefore, it is important that the operator is exactly aware of which commands are currently pending and whether they can be executed as desired. This


Behavior Control - Set_Mode > Decide_Orientation > Decide_Orientation
face_down
face_up
sitting
standing

(a) FlorBE



(b) FlexBE

Figure 6.1: Comparison of displaying the active state.

issue is explicitly addressed by the `RC.Snyc` component of the *Runtime Control* in *FlexBE*. This approach has proven as very effective for allowing comprehensive execution of commands.

Parametrization of behaviors and choosing appropriate values when starting one, is also a helpful concept of behavior execution. But since behavior development is now much more integrated in *FlexBE*, a second way for configuring a behavior for execution has emerged. Instead of using the behavior parameters for setting certain values, they were just provided as private configuration variables and submitted as changes to the behavior source code. This is completely possible, however, it is not the desired and modular way for providing these values. It turned out that the reason for using the private configuration was to allow the entered values to be saved, and thus not need to set them again for each execution. Instead, behavior parameters currently have their default value for each execution unless changed. This shows that it will be advantageous to save the last provided value to a parameter and to suggest it for the next execution instead of the default value. In addition, the operator can choose to go back to the default value at any time.

6.2 Behavior Performance

In order to evaluate the performance of behaviors as such, a *State Logger* has been implemented. This logger is integrated into the behavior engine and automatically logs each run of a behavior. During execution, one log entry is created each time the state machine takes a transition. The point in time of this transition and information on which outcome of which state has been returned are logged, as well as the implementation class of a state and if the transition has been triggered autonomously or if it has been blocked. These log files can be evaluated afterwards regarding different indicators, in order to measure the performance of various aspects of behavior execution. The user can specify a set of properties to be calculated, and regarding which criteria these properties should be accumulated.

An overview of available keys for properties and criteria is given in the appendix. This concept of decoupled data recording and evaluation enables detailed information regarding multiple aspects of behavior performance even if these have not been specified before execution. For example, the accuracy with which the robot had correctly chosen a certain transition can be calculated, as well as the number of required operator interventions at a certain point in the behavior, or the amount of time spent in a certain class of states.

Quantitative performance of behavior execution can be measured regarding efficiency and effectiveness. Both aspects have been investigated by using a relatively simple and thus comprehensible demo behavior inspired by the following scenario: A robot, in this case a simulated *ATLAS v5* robot using *Gazebo*[1], has the task to collect data regarding the environment in a disaster scenario. Its behavior is designed to iteratively walk a short distance and then take a camera image and send this data to the remote operator. However, during its mission, the robot enters a damaged building where the available light is insufficient for taking useful camera images. At this point, the operator decides to modify the behavior during a walking phase of the robot in order to collect LIDAR scan data instead of camera images. The implementation of this exploration behavior is given in the appendix.

[1] `http://gazebosim.org/` *(03/31/2015)*

State	Visits	Intv	Waiting	Active
Init Radius	22	0	0.000	0.510
Calculate Next Pose	22	0	0.000	1.053
Determine Waypoint Valid	22	0	0.000	0.447
Plan To Waypoint	22	0	0.000	0.441
Perform Walking	22	1	0.000	561.672
Take Camera Image	16	0	0.000	1.443
Take Laser Scan	6	0	0.000	0.846
Send Image To Operator	22	0	0.000	11.530
Check Finished	22	1	2.316	3.901

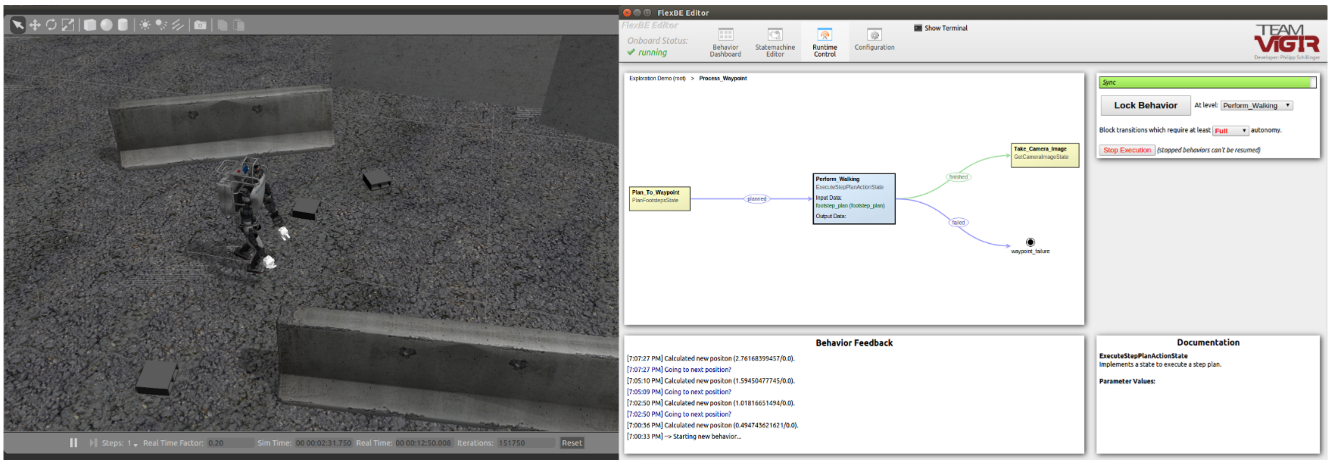
Figure 6.2: Evaluation of the *State Logger* log file of the exploration behavior.

An evaluation of this demo mission is provided in figure 6.2. For each state, the amount of state executions (*Visits*), the amount of interventions (*Intv*), the time in seconds spent with waiting for the operator (*Waiting*), and the total time in seconds this state has been the active state during execution (*Active*), is listed. The intervention and waiting time in the *Check Finished* state results from ending the behavior. In order to keep the behavior from taking another iteration, the operator lowered the *Autonomy Level* and rejected the suggested transition while instead forcing the transition to return the *finished* outcome of the behavior.

As indicated by the amount of state visits, 22 iterations have been executed. After taking 16 camera images, the behavior has been modified by the operator and the *Take Camera Image* state has been replaced by the *Take Laser Scan* for the remaining six iterations. Although this is the log file for a single behavior execution, both, the camera image and the laser scan state, are listed although they only existed in different versions of this behavior. This way of modifying the behavior during its execution without the need for restarting it has proven to be very efficient. Restarting the behavior would have required significantly more time, especially when considering that specifying this modification could have taken much more time without use of a behavior framework built for making quick and safe changes, and that the new behavior would have needed to be somehow transfered to the remote robot.

Regarding efficiency it is even more remarkable that the behavior modification did not at all cause the robot to wait for the operator and pause its own execution. As indicated by the log file, the modification has taken place while the robot was walking to its next position. Since this state has been active for most of the time anyway, it is a good point in execution to be locked in order to allow runtime modifications. While the state was locked, the robot kept walking. But in case that specifying the modifications would have taken longer than the robot walks, the robot would have waited before continuing its behavior execution.

Furthermore, the available partial semantic verification of behavior specifications has proven useful as well to prevent task failure in this context. When the operator removed the *Take Camera Image* state and added the *Take Laser Scan* state, he forgot that the *Send Image To Operator* state still referred to the *userdata* key provided by the camera state. After the verification check displayed an error message in order to tell the operator that the expected camera *userdata* is



Behavior Feedback

```
[7:07:27 PM] Calculated new positon (2.76168399457/0.0).
[7:07:27 PM] Going to next position?
[7:05:10 PM] Calculated new positon (1.59450477745/0.0).
[7:05:09 PM] Going to next position?
[7:02:50 PM] Calculated new positon (1.01816651494/0.0).
[7:02:50 PM] Going to next position?
[7:00:36 PM] Calculated new positon (0.494743621621/0.0).
[7:00:33 PM] --> Starting new behavior...
```

Documentation

ExecuteStepPlanActionState
Implements a state to execute a step plan.

Parameter Values:

Figure 6.3: Simulating the evaluation scenario while supervising the behavior using *FlexBE*.

not available, the operator was able to quickly change the reference to point to the laser scan. This process happened before even transferring the new behavior version to the robot.

During behavior execution, network usage has been measured by using *nethogs*[2] in order to determine the bandwidth efficiency of the behavior engine. Since an absolute measurement of the required bandwidth was technically not possible, several five minute runs of the behavior have been conducted. One part was executing the behavior engine as normal, while message transmission of the behavior engine was disabled for the other part. Runs with enabled message transmission accumulated a total amount 9.464 kB sent data during the five minutes, runs without sending messages accumulated 7.323 kB on average. This yields a bandwidth increase of 2.141 kB caused by the behavior engine, which equals an average data rate of only 7.14 B/sec including logged messages sent to the operator for detailed feedback regarding execution.

Furthermore, the effectiveness of this mission has to be evaluated. The goal was to collect data regarding the environment and during a mission time of slightly less than ten minutes. The presented approach of using *FlexBE* in this scenario resulted in a total of 16 camera images and 6 LIDAR scans. Effectiveness of this approach can be compared to three other possible approaches.

First, the operator could have just let the robot keep executing its initial behavior. In this case, 22 camera images would have been available after execution, but only 16 of them containing usable information. Second, the operator could have stopped the behavior as soon as it was not able to provide useful data anymore, and he could have continued the mission by teleoperation. Without considering that the available bandwidth might not have been sufficient for this approach and that the operator might not have had enough data for finding a viable next position for the robot to walk to, this approach would have been much slower. As given by the behavior log data, commanding to record one LIDAR scan took around 140 ms and execution of all states required to start walking only 111 ms on average. Third, the operator could have stopped the behavior as in the previous approach, but instead of continuing execution by relying on teleoperation, specify a new version of the behavior manually, transfer it to the robot, and execute it, considering a system flexible enough to handle initially unknown behaviors. However, this

[2] `http://nethogs.sourceforge.net/` *(03/31/2015)*

approach would have required much more time until the new behavior would have been able to start execution and thus, it is not really applicable in such a short mission.

In summary, the novel approach taken in the given scenario, now available due to the work presented in this thesis, showed a good task performance and allowed for interruption-free task execution although a significant and unforeseen disturbance regarding the expected circumstances occurred. Considering a more complex scenario than the one chosen for a meaningful evaluation, as well as highly limited communication and further restrictions in direct control of the robot, modifying a behavior in the way developed in this thesis and provided by *FlexBE* might be, so far, the only option to prevent task failure.

6.3 DARPA Robotics Challenge

Main purpose of this thesis and the development of the *FlexBE* behavior framework is its application in the *DARPA Robotics Challenge*, in order to provide the robot the ability to semi-autonomously make decisions and to autonomously execute high-level commands. In preparation for the *DRC Finals*, a lot of states and a set of behaviors have been developed using *FlexBE*. The developed states interface with the rest of the team's software and include states for footstep planning and execution, endeffector motions, grasping, and changing internal parameters. A full list of states is provided in the appendix.

Until the submission date of this thesis, the team's *ATLAS* robot had some technical issues preventing it from executing complex actions. For this reason, testing of developed behaviors mostly took place in simulation. Nevertheless, behaviors have been developed and prepared for execution. An excerpt of a developed behavior is given in figures 6.4 and 6.5. In this part, the robot is supposed to walk near an object of interest which should be grasped afterwards. The object is specified by the operator, based on the perception of the robot, as a template. Thereafter, the robot is able to extract several data, such as a viable position to stand in order to grasp the specified object, from this template by leveraging the template server. Based on

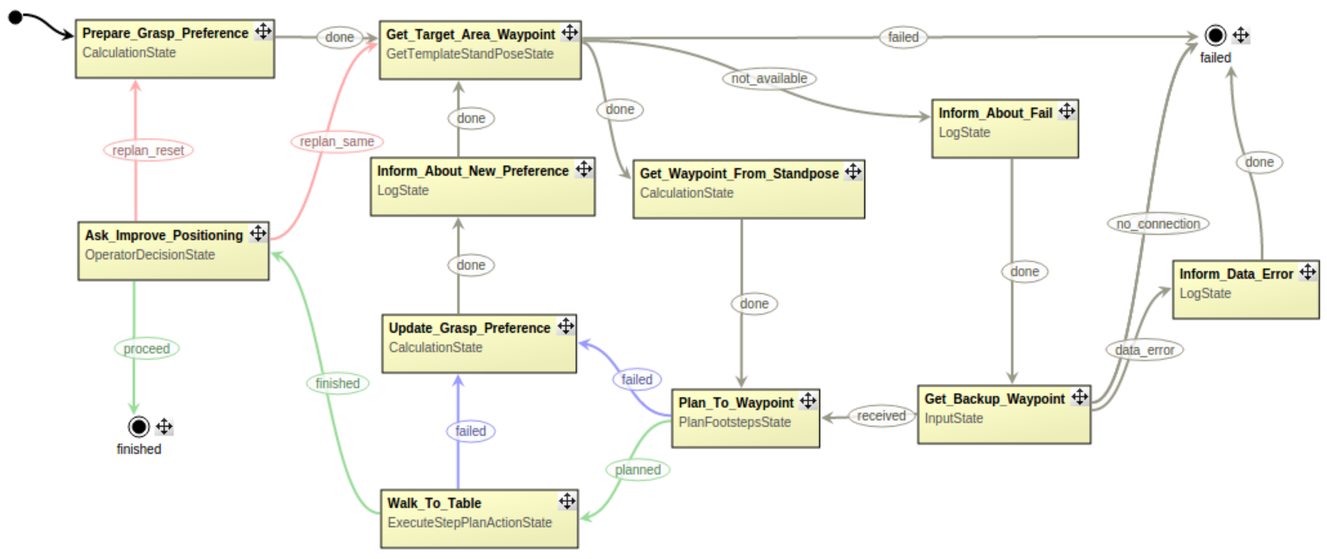


Figure 6.4: Behavior to walk near an object of interest.

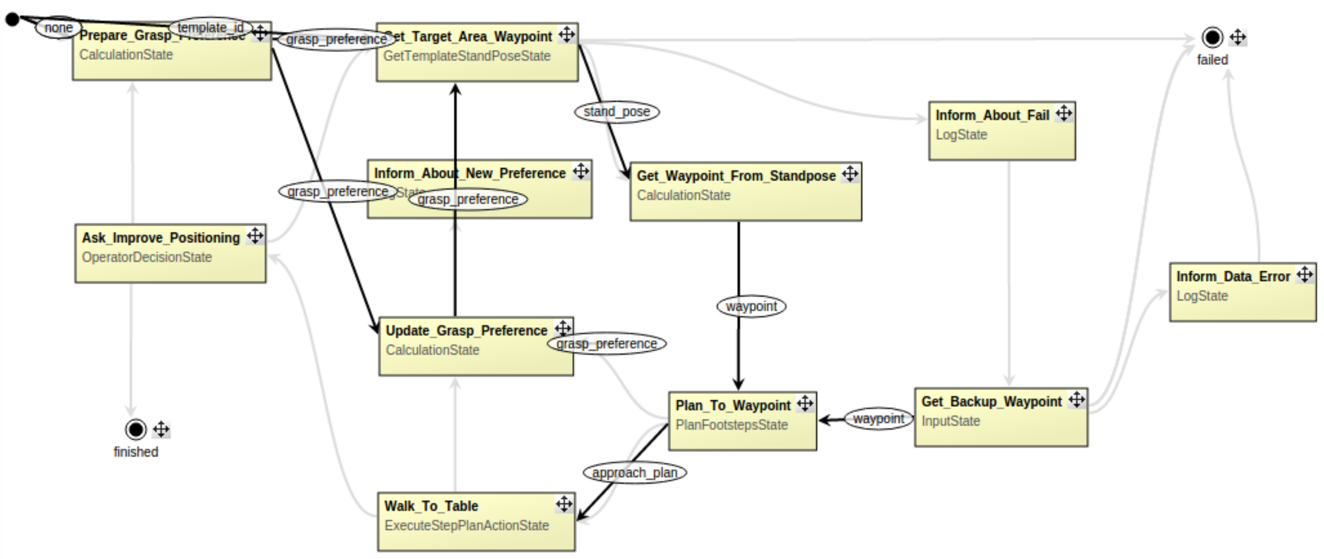


Figure 6.5: Dataflow graph overlay of the same behavior indicating which data keys are passed between the states.

this retrieved stand pose, a footstep plan is generated and executed. Considering that the robot might be able to collect more accurate perception data when having approached the object, the operator is able to command the robot afterwards to improve its position by adjusting the template and commanding another iteration.

Despite the technical issues, some behaviors have also been executed directly on the *ATLAS* robot, mainly focused on supporting development of specific basic system components and low-level robot control. Hence, the usage of this robot in the context of behaviors is presented in the next section. Furthermore, also *Team Hector* uses *FlexBE* for controlling their *THOR-MANG* robot in the *DRC Finals*. This robot is presented in the section afterwards.

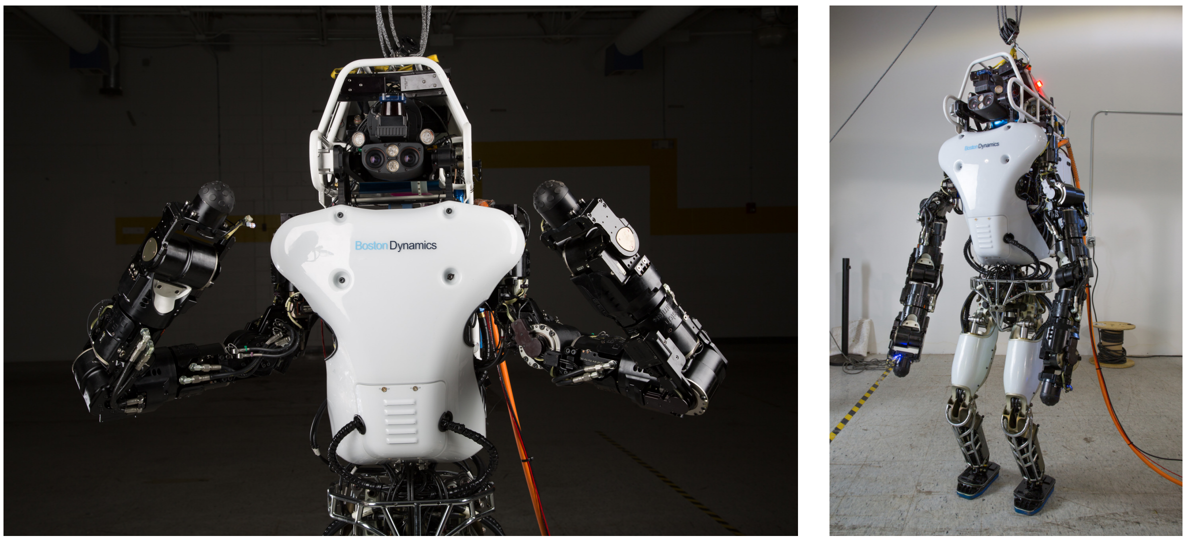


Figure 6.6: New version of the *ATLAS* robot, developed by Boston Dynamics.[1]

[1] `http://www.darpa.mil/NewsEvents/Releases/2015/01/20.aspx` *(03/27/2015)*

6.3.1 ATLAS

The *ATLAS* robot has been developed by Boston Dynamics and funded by *DARPA*. Figure 6.6 shows the latest currently available version of the robot. Compared to the previous version of this robot, around 75% of the parts are exchanged or improved in the new *ATLAS v5*. Deviant from figure 6.6, the final *ATLAS v5* includes electric forearms, containing an additional joint. Thus, the arms of *ATLAS v5* have seven degrees of freedom each, providing much better manipulation capabilities. All in all, the robot now has 30 joints, most of these hydraulically actuated, a size of 1.88 meters, and a weight of 157 kilograms.

Main goal of the upgrades to *ATLAS* was to remove the need for a wired connection to the robot. Instead of relying on an external pump for actuating the hydraulic joints, the robot now has an energy-efficient and quiet variable pressure pump built-in. Furthermore, a wireless router integrated in the head of the robot supersedes the need for wired communication and a battery with a capacity of 3.7 kilowatt hours provides enough energy for roughly one hour of mixed operation, including walking and manipulation.

During the development phase prior to the competition, main focus of behaviors was to support the development of different parts of the system, foremost by automating tests in order to determine specific properties of the robot. For this purpose, a system identification behavior has been implemented. This behavior loads defined trajectories, coordinates and automates their execution, and records all required data such as commanded and actual joint values including video recordings of all experiments. During an experiment, the operator always has the possibility to let the behavior run in full autonomy, or to reduce the *Autonomy Level* at certain points when executing rather risky trajectories or parameter settings.

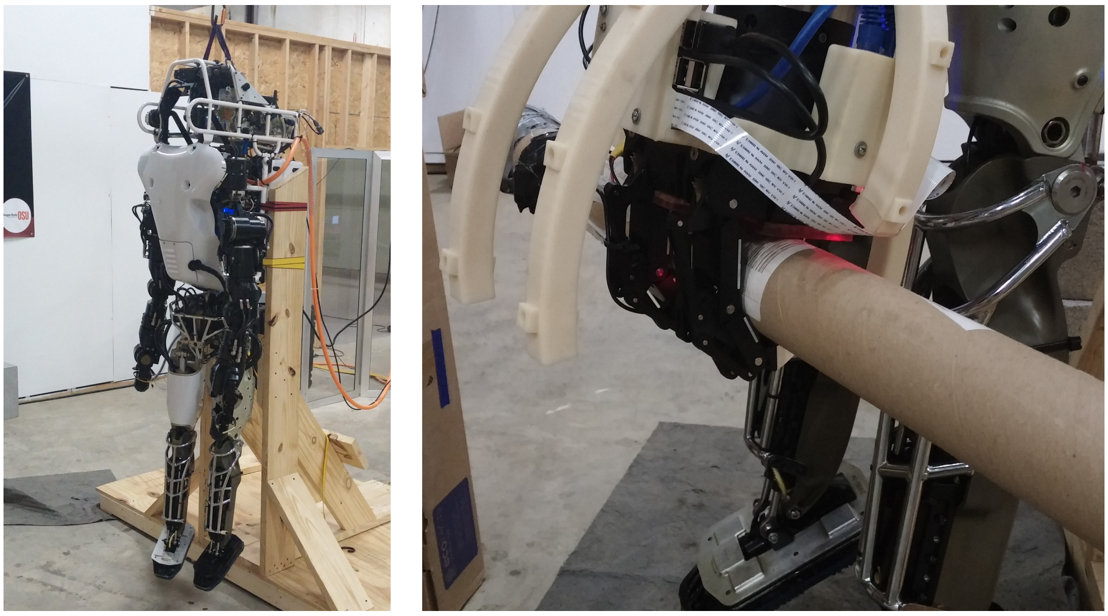

(a) (b)

Figure 6.7: Testing with the robot: (a) ready for system identification, (b) testing simple grasping.

In total, several hours of pure execution time of this system identification behavior accumulated, testing more than one hundred single trajectories with different settings and proving the robustness and applicability of *FlexBE* along with such a complex system as the *ATLAS* robot. Furthermore, development and operation of the resulting behavior have been made by different persons, providing a great opportunity for evaluation of *FlexBE* and for improving certain aspects.

Another behavior executed on the robot is the so-called *Praying Mantis* behavior. This behavior commands the joints to go near their respective limits and then commands the low-level controller of each joint to carefully push beyond these limits while monitoring the actual joint positions. This allows to determine joint offsets of the robot and to calibrate the joints for more reliable execution of motions afterwards. Executing this behavior in full autonomy each time the robot is started provides an easy and effective way for always having a calibrated robot without the need for manual calibration and for defining static offsets in a configuration file.

6.3.2 THOR-MANG

The *THOR-MANG* is a humanoid robot platform developed in a collaboration between Robotis[3], Virginia Tech[4], the University of Pennsylvania[5], and Harris Corporation[6]. The robot consists of 30 electrically actuated joints based on Robotis' Dynamixel PRO servomotors, is 1.47 meters tall and weighs 49 kilograms. This robot is used by a cooperation between *Team Hector* and *Team ViGIR* to demonstrate the applicability of the software developed by *Team ViGIR* not only for an *ATLAS* robot, but also for further robots such as *THOR-MANG*.

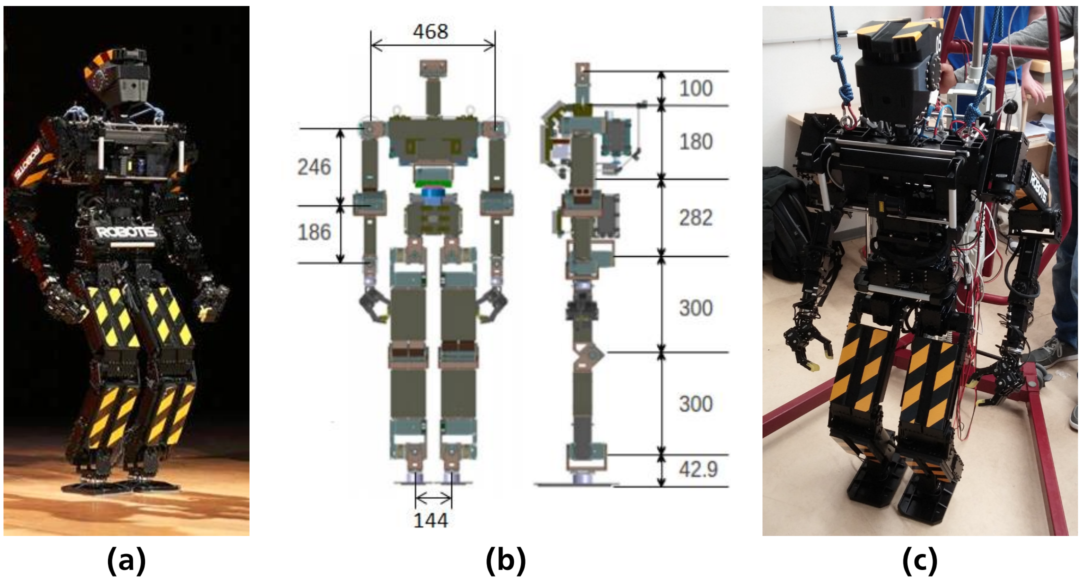


Figure 6.8: *THOR-MANG* robot platform, (a) and (b) taken from [27].

3 http://www.robotis.com/ *(03/31/2015)*
4 http://www.romela.org/ *(03/31/2015)*
5 https://www.grasp.upenn.edu/ *(03/31/2015)*
6 http://harris.com/ *(03/31/2015)*

6.4 Future Work

Based on the results of evaluation and the overall capabilities of the chosen approach, further work can be conducted in order to improve the behavior framework *FlexBE* and its user interface. This section lists some considerations regarding main improvements based on observations and discusses how currently open problems can be solved in the future. This is intended to provide a basis for further development of *FlexBE*. Until the DRC Finals, development will definitely be continued by *Team ViGIR*. Afterwards, it is planned to publicly release *FlexBE* and thus to make it available for a broad field of applications in order to effectively support the development of high-level behaviors and more intelligent and flexible robot applications.

6.4.1 Synthesis Integration

As already discussed in chapter 2.2.1, an integration of synthesis into the process of runtime modifications to behaviors has the potential to create powerful synergy effects. Obviously, it will make it much easier for the operator to specify runtime changes since he only has to give high level commands to the synthesizer instead of completely modeling the changes by himself. But in addition, it enables incorporation of even more powerful autonomous adaptation. It may not be applicable in the field of rescue robotics, but in scenarios where the environment can be much better perceived by the robot and the consequences of failure are considerably low, using a combination of behavior synthesis and runtime modifications will allow the robot to change its own behavior during execution depending on how the environment changes.

Considering having the synthesizer available as a primitive invocable by a state, one could define a sub-behavior which at first locks execution in its parent container, invokes the synthesizer to generate a new version of the top-level behavior, applies it using *FlexBE*'s capability

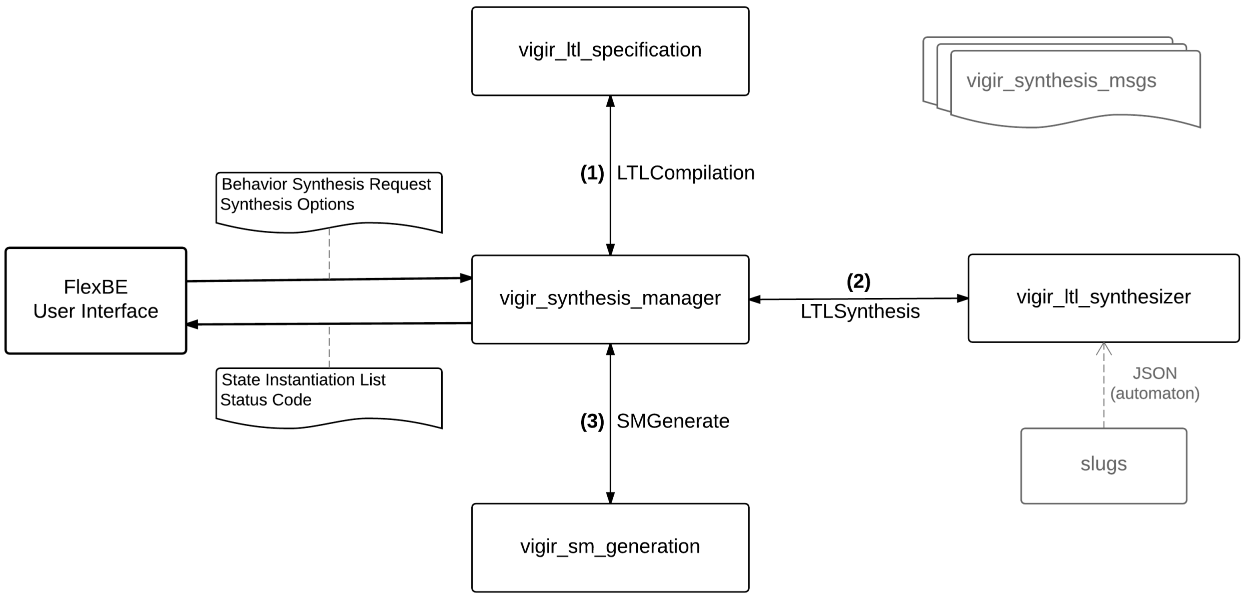


Figure 6.9: System concept for requesting behavior synthesis from *FlexBE*'s user interface.

for runtime modifications, and finally continues execution by now using the new version. This sub-behavior can be embedded into any other behavior, and be invoked as soon as a difference in the environment is detected. Synthesis of the new behavior will not be restricted in any way since the new version will still contain the synthesis sub-behavior which is the pivot state of this switch. This whole process can finally be combined with approaches on machine learning in order to optimize situational adaption of robot behaviors.

While this way of combining synthesis and runtime modifications is rather a long-term objective, supporting operator and developer by using synthesis for more efficient behavior definition is really a practical extension to the framework as it is now. It is currently developed together with the *Verifiable Robotics Research Group* at Cornell University. It is planned that the developer will not have to start with an empty state machine when starting to create a new behavior or new parts of an existing behavior. Instead, he can provide a set of high-level initial conditions and goals to be achieved by this part of the behavior. Behavior synthesis will then draft a first version of a possible behavior to achieve these goals, which can be modified and extended by the developer afterwards.

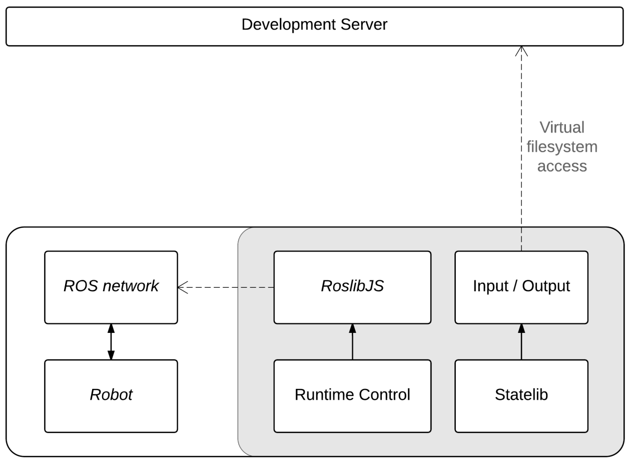
This development can be seen as a first step towards combining *FlexBE* with behavior synthesis. More advanced integration can then actively assist the developer when specifying behaviors manually by relying on the state constraints defined in chapter 3.5. For example, when the developer adds a new state to the behavior and connects it in a way that its preconditions are not fulfilled, required intermediate states can be synthesized and automatically added, instead of just displaying an error message.

6.4.2 Web-based Solution

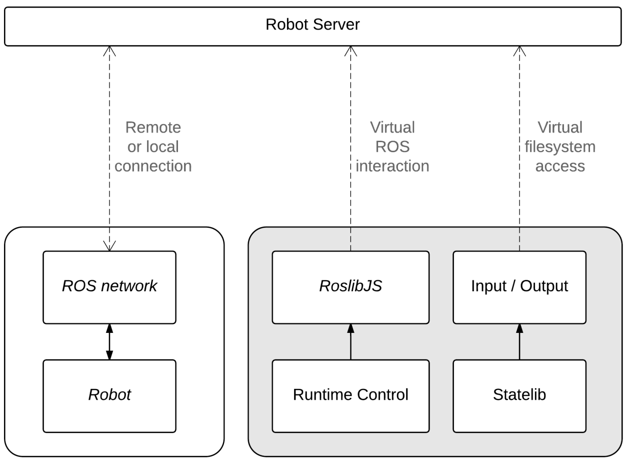
FlexBE not only incorporates a flexible behavior engine, but also a very flexible user interface by relying on *HTML* and *JavaScript*. In general, it would be possible to run the user interface as a web app, removing the need for installing any software locally and being platform independent. This means, that the operator would actually be able to control the robot even with his smartphone or a tablet. However, there are currently two aspects limiting this form of usage.

First, the user interface requires native file system access for parsing available states and behaviors, as well as generating source code and creating ROS packages for new behaviors. This is currently realized by implementing the user interface as a *Google Chrome App*, depending on *Google Chrome* to be available and requiring the user to install the app. Second, for actually executing behaviors, a connection to ROS is required. While this is possible without using *Google Chrome*, it is inevitable to have *Rosbridge* running on a *Linux* machine. Eliminating both of these two needs is the prerequisite for designing a web-based solution.

One feasible approach would be to rely on a central server. A simple development server similar to using common repositories for software development would already eliminate the need for depending on *Google Chrome* and would enable to develop behaviors in any web browser without installing any software. The development server would store all developed states and behaviors, and instead of accessing the local file system, the user interface client would just communicate with the server. However, this solution would still require a *Linux*-based operating system running ROS and *Rosbridge* to create a connection with the ROS network.



(a) Web client connected to the robot



(b) Remote web client

Figure 6.10: Possible concepts for server integration.

Instead of just having a remote server as development repository for defined states and behaviors, it is also possible to have a server with access to the ROS network of the robot. This server could even be executed as a local ROS node when working on a *Linux* machine. Instead of communicating directly with ROS from the user interface based on *Rosbridge*, the user interface would communicate instead with the server, which is then able to send and receive local messages in the ROS network of the robot. This would eliminate all dependencies on *Linux* and even *Rosbridge* and thus enable a completely remote operation from any device, possibly even via the internet.

6.4.3 Parallel Execution

Right now, *FlexBE* is designed to support only one active state at a given point in time. This state is centered in the *Runtime Control* view and the operator can send commands directly influencing its execution. However, in some cases, being able to execute multiple states at the same time would be helpful. In order to provide a first solution for these scenarios, the `ConcurrentState` has been added, allowing parallel execution of states. While this adds everything required from an execution-point of view, it does not provide the same operator integration as known from *FlexBE*.

The `ConcurrentState` is just a common state and thus, its internal functionality is transparent to *FlexBE*. When this state is executed, it is displayed as the sole currently active state, no matter which states or even state machines are executed internally. For this reason, further work can be done to better integrate concurrency and parallel execution into the user interface and the communication protocol of *FlexBE*. However, this will require some fundamental changes of the concept since there will be for example no single pivot state. Also verification will be much more complex when having to account for interdependencies between possibly simultaneously running states.

6.4.4 Multiple Robots

In future, more advanced robotic rescue missions will most likely involve multiple robots. In this context, it is important to note that *FlexBE* itself is not limited to controlling a single robot. It rather can be seen in a way that *FlexBE* is controlling a system. In current applications, as presented in the previous part of this chapter, the system relies on a single robot to execute the specified actions. But there is no restriction that the system cannot utilize the capabilities of two or more robots as well in order to execute the commanded actions.

Nevertheless, practical applications in multi-robot systems will likely need some additional features. The most important feature in this context, for sure, is the ability to execute states in parallel when sticking to the approach of central coordination of the agents of this system. If choosing an approach where each agent forms a sub-system and is controlled decentralized by running its own instance of *FlexBE*, it should be possible to use different ROS namespaces to independently control each robot. However, each robot would have its own user interface this way. This may be intended in some cases, but may not in other cases. A solution to control multiple instances of onboard behavior engines with a single, central instance of *FlexBE* would be a helpful feature in this context.

7 Conclusion

This thesis presents an approach for realizing modifications to high-level behavior control even while the robot is running, controlled by this behavior itself. Furthermore, the operator is considered as an important factor for both, monitoring behavior execution and specifying required modifications, and thus, is tightly integrated in the system. This approach enables application of the developed system in a variety of scenarios, even without detailed knowledge in advance about the environment and how the desired goal can be achieved.

Main focus of this thesis is the application in the field of rescue robotics, using advanced humanoid robots for combining capabilities such as dexterous manipulation and navigation across rough terrain. Efficient performance of robots is considerably impeded by the unstructured environment in typical disaster mitigation scenarios. Designing a complete, specific high-level behavior control approach in advance is not possible, considering the limited a priori knowledge and the rather complex system. Furthermore, the ability to communicate with the remote robot is severely limited and failure of task execution has to be considered very carefully.

The presented approach models behaviors as hierarchical state machines. Single states correspond to actions that are performed by the robot in order to get one step closer to the desired task goal. Transitions to next states are taken depending on the result of execution, allowing to create robust behaviors accounting for various kinds of failure. This approach abstracts very well from task-specific details and provides a widely applicable basis for high-level decisions and coordination between multiple, otherwise independent components of complex systems. Behavior modifications can be realized on the level of state composition, enabling to clearly identify parts which are safe to be modified and to apply verification checks to greatly reduce the risk of runtime failure.

Furthermore, this thesis includes implementation of the developed concepts. The resulting behavior engine, *FlexBE*, not only targets execution of behaviors based on the discussed concepts, but rather incorporates a fully practically applicable software environment for developing, executing, monitoring, and modifying behaviors. Worth mentioning is the included user interface, carefully designed for both expert and non-expert users and allowing to effectively abstract from low-level details while giving accurate feedback on behavior execution. By utilizing powerful synchronization concepts such as mirroring behaviors, bandwidth usage is still kept very low.

In summary, this thesis contributes to the development of more intelligent and complex robots, especially in an unstructured and hardly accessible environment. This is done by providing the capability to flexibly adapt the high-level behavior of the remote robot and effectively integrating cooperation with the operator. Specification of new behaviors and behavior modifications are assisted by an editor which also includes partial semantic behavior verification and guarantees generation of a syntactically correct implementation of the specification.

Finally, the results of this thesis are applied by *Team ViGIR* and *Team Hector* in the international *DARPA Robotics Challenge*, competing along with some of the most recent and advanced rescue robotics research efforts. *FlexBE* has already been used by *Team ViGIR* for running system identification tests, calibration behaviors, and automated component tests on the *ATLAS* robot during development of the basic system capabilities. More complex behaviors have also been developed by using the state machine editor as well as testing it with a simulated version of the system both in order to evaluate the results of this thesis and in order to solve the tasks of the *DRC Finals*, involving multiple members of the team using *FlexBE*.

Although developed in the context of the *DARPA Robotics Challenge*, the behavior framework *FlexBE* itself is not limited to a specific system and widely applicable for a broad variety of different systems and tasks. The concepts of abstraction used by *FlexBE* allow for use in any context, as long as the basic capabilities of the system can be accessed by a high-level controller as presented in this thesis. It is planned to publicly release the implemented behavior engine after the *DRC Finals* and thus support future research efforts in robotics by providing a flexible basis for system integration.

Bibliography

[1] Philipp Allgeuer and Sven Behnke. Hierarchical and state-based architectures for robot behavior planning and control. In *Proceedings of 8th Workshop on Humanoid Soccer Robots, IEEE Int. Conf. on Humanoid Robots, Atlanta, USA*, 2013.

[2] Jenay M Beer, Arthur D Fisk, and Wendy A Rogers. Toward a framework for levels of robot autonomy in human-robot interaction. *Journal of Human-Robot Interaction*, 3(2):74–99, 2014.

[3] Doug Binks, Matthew Jack, and Will Wilson. Runtime compiled c++ for rapid ai development. *Game AI Pro: Collected Wisdom of Game AI Professionals*, page 201, 2013.

[4] Kyle L Blatter and Michael A Goodrich. Modeling temporal latency in the supervisory control of human-robot teams. 2014.

[5] J Bohren and Steve Cousins. The smach high-level executive [ros news]. *Robotics & Automation Magazine, IEEE*, 17(4):18–20, 2010.

[6] Jeffrey M Bradshaw. From knowledge science to symbiosis science. *International Journal of Human-Computer Studies*, 71(2):171–176, 2013.

[7] David C Conner. *Integrating planning and control for constrained dynamical systems*. ProQuest, 2007.

[8] Thijs Jeffry de Haas, Tim Laue, and T Rofer. A scripting-based approach to robot behavior engineering using hierarchical generators. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 4736–4741. IEEE, 2012.

[9] L De Silva and H Ekanayake. Behavior-based robotics and the reactive paradigm a survey. In *Computer and Information Technology, 2008. ICCIT 2008. 11th International Conference on*, pages 36–43. IEEE, 2008.

[10] Mathew DeDonato, Velin Dimitrov, Ruixiang Du, Ryan Giovacchini, Kevin Knoedler, Xianchao Long, Felipe Polido, Michael A Gennert, Taşkın Padır, Siyuan Feng, et al. Human-in-the-loop control of a humanoid robot for disaster response: A report from the darpa robotics challenge trials. *Journal of Field Robotics*, 32(2):275–292, 2015.

[11] Eladio Domı, Beatriz Pérez, Ángel L Rubio, et al. A systematic review of code generation proposals from state machine specifications. *Information and Software Technology*, 54(10): 1045–1066, 2012.

[12] B Espiau, K Kapellos, and M Jourdan. Formal verification in robotics: Why and how? In *Robotics Research*, pages 225–236. Springer, 1996.

[13] Michalis Foukarakis, Asterios Leonidis, Margherita Antona, and Constantine Stephanidis. Combining finite state machine and decision-making tools for adaptable robot behavior. In *Universal Access in Human-Computer Interaction. Aging and Assistive Environments*, pages 625–635. Springer, 2014.

[14] Stephen Hart, Paul Dinh, John D Yamokoski, Brian Wightman, and Nicolaus Radford. Robot task commander: A framework and ide for robot application development. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1547–1554. IEEE, 2014.

[15] Rajibul Huq, George KI Mann, and Ray G Gosine. Behavior-modulation technique in mobile robotics using fuzzy discrete event system. *Robotics, IEEE Transactions on*, 22(5): 903–916, 2006.

[16] Matthew Johnson, Jeffrey M Bradshaw, Paul J Feltovich, Catholijn Jonker, Maarten Sierhuis, and Birna van Riemsdijk. Toward coactivity. In *Proceedings of the 5th ACM/IEEE international conference on Human-robot interaction*, pages 101–102. IEEE Press, 2010.

[17] Matthew Johnson, Jeffrey M Bradshaw, Paul J Feltovich, Catholijn M Jonker, Birna van Riemsdijk, and Maarten Sierhuis. The fundamental principle of coactive design: Interdependence must shape autonomy. In *Coordination, organizations, institutions, and norms in agent systems VI*, pages 172–191. Springer, 2011.

[18] Matthew Johnson, Brandon Shrewsbury, Sylvain Bertrand, Tingfan Wu, Daniel Duran, Marshall Floyd, Peter Abeles, Douglas Stephen, Nathan Mertins, Alex Lesman, et al. Team ihmc's lessons learned from the darpa robotics challenge trials. *Journal of Field Robotics*, 32(2):192–208, 2015.

[19] MJ Johnson. *Coactive Design: Designing Support for Interdependence in Human-Robot Teamwork*. PhD thesis, TU Delft, Delft University of Technology, 2014.

[20] PT Kidd. Design of human-centred robotic systems. In *Human Robot Interaction*, pages 225–241. London, UK: Taylor & Francis, 1992.

[21] Alexander Knapp and Stephan Merz. Model checking and code generation for uml state machines and collaborations. *Proc. 5th Wsh. Tools for System Design and Verification*, pages 59–64, 2002.

[22] Stefan Kohlbrecher, Alberto Romay, Alexander Stumpf, Anant Gupta, Oskar von Stryk, Felipe Bacim, Doug A Bowman, Alex Goins, Ravi Balasubramanian, and David C Conner. Human-robot teaming for rescue missions: Team vigir's approach to the 2013 darpa robotics challenge trials. *Journal of Field Robotics*, 2014.

[23] Szilveszter Kovács. A flexible fuzzy behaviour-based control structure.

[24] Adam Eric Leeper, Kaijen Hsiao, Matei Ciocarlie, Leila Takayama, and David Gossow. Strategies for human-in-the-loop robotic grasping. In *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction*, pages 1–8. ACM, 2012.

[25] Georgios Lidoris, Dirk Wollherr, and Martin Buss. *Bayesian Framework for State Estimation and Robot Behaviour Selection in Dynamic Environments*. INTECH Open Access Publisher, 2008.

[26] MajaJ. Matarić and François Michaud. Behavior-based systems. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, pages 891–909. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-23957-4.

[27] Stephen McGill, Seung-Joon Yi, Larry Vadakedathu, Qin He, Dennis Hong, and Daniel Lee. Team thorwin.

[28] Terrance Medina, Maria Hybinette, and Tucker Balch. Behavior-based code generation for robots and autonomous agents. In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, pages 172–177. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014.

[29] Robin R Murphy. Human-robot interaction in rescue robotics. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 34(2):138–153, 2004.

[30] Keiji Nagatani, Seiga Kiribayashi, Yoshito Okada, Kazuki Otake, Kazuya Yoshida, Satoshi Tadokoro, Takeshi Nishimura, Tomoaki Yoshida, Eiji Koyanagi, Mineo Fukushima, et al. Emergency response to the nuclear accident at the fukushima daiichi nuclear power plants using mobile rescue robots. *Journal of Field Robotics*, 30(1):44–63, 2013.

[31] Tim Niemüller, Alexander Ferrein, and Gerhard Lakemeyer. A lua-based behavior engine for controlling the humanoid robot nao. In *RoboCup 2009: Robot Soccer World Cup XIII*, pages 240–251. Springer, 2010.

[32] Donald A Norman. Cognitive engineering, 1986.

[33] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, pages 1043–1049, 1998.

[34] Nicolaus A Radford, Philip Strawser, Kimberly Hambuchen, Joshua S Mehling, William K Verdeyen, A Stuart Donnan, James Holley, Jairo Sanchez, Vienny Nguyen, Lyndon Bridgwater, et al. Valkyrie: Nasa's first bipedal humanoid robot. *Journal of Field Robotics*, 2015.

[35] Vasumathi Raman and Hadas Kress-Gazit. Automated feedback for unachievable high-level robot behaviors. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 5156–5162. IEEE, 2012.

[36] Vasumathi Raman, Nir Piterman, and Hadas Kress-Gazit. Provably correct continuous control for high-level robot behaviors with actions of arbitrary execution durations. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 4075–4081. IEEE, 2013.

[37] Philipp Schillinger. Development of an operator centric behavior control approach for a humanoid robot. Bachelor's thesis, Technische Universitaet Darmstadt, Fachbereich Informatik, 2013.

[38] Jean Scholtz. Theory and evaluation of human robot interactions. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 10–pp. IEEE, 2003.

[39] Julie Shah, James Wiken, Brian Williams, and Cynthia Breazeal. Improved human-robot team performance using chaski, a human-inspired plan execution system. In *Proceedings of the 6th international conference on Human-robot interaction*, pages 29–36. ACM, 2011.

[40] Aaron Steinfeld, Terrence Fong, David Kaber, Michael Lewis, Jean Scholtz, Alan Schultz, and Michael Goodrich. Common metrics for human-robot interaction. In *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*, pages 33–40. ACM, 2006.

[41] Fernando Trasvina, Eric Shepherd, et al. Introduction to Object-Oriented JavaScript. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript`, 2007. [Online; accessed 19-March-2015].

[42] Eiji Uchibe, Minoru Asada, and Koh Hosoda. Behavior coordination for a mobile robot using modular reinforcement learning. In *Intelligent Robots and Systems' 96, IROS 96, Proceedings of the 1996 IEEE/RSJ International Conference on*, volume 3, pages 1329–1336. IEEE, 1996.

[43] Holly A Yanco, Adam Norton, Willard Ober, David Shane, Anna Skinner, and Jack Vice. Analysis of human-robot interaction at the darpa robotics challenge trials. *Journal of Field Robotics*, 2015.

[44] Matt Zucker, Sungmoon Joo, Michael X Grey, Christopher Rasmussen, Eric Huang, Michael Stilman, and Aaron Bobick. A general-purpose system for teleoperation of the drc-hubo humanoid robot. *Journal of Field Robotics*, 2015.

Appendix

A.1 List of Developed States

The following tables list all state implementations currently available in *FlexBE*. The states are split into different packages regarding the system they can be used with. All states with no external dependencies are part of the `flexbe_states` package and can be used along with any system. Further states have been developed to be used with *ATLAS* and the *Team ViGIR* software, these are listed in the `flexbe_atlas_states`. However, although developed to be used with *ATLAS*, they can also be used with *THOR-MANG* in conjuction with the *Team ViGIR* software. A more detailed documentation including the interface of each state is available in the *FlexBE Statemachine Editor* or in the source code of each state.

A.1.1 flexbe_states

State Class	**Description**
`CalculationState`	Implements a state that can perform a calculation based on userdata. The calculation is a function which takes exactly one parameter, input_value from userdata, and its return value is stored in output_value after leaving the state.
`CheckConditionState`	Checks if the given condition is true and returns the corresponding outcome. This state can be used if the further control flow of the behavior depends on a simple condition.
`ConcurrentState`	Implements concurrent execution of states. Depending on the outcomes of the single states, the outcome of this state is determined according to the given set of rules.
`DecisionState`	Evaluates a condition function in order to return one of the specified outcomes. This state can be used if the further control flow of the behavior depends on an advanced condition.
`FlexibleCalculationState`	Implements a state that can perform a calculation based on multiple userdata inputs provided as a list to the calculation function.
`InputState`	Implements a state where the state machine needs an input from the operator. Requests of different types, such as requesting a waypoint, a template, or a pose, can be specified.
`LogState`	A state that can log a predefined message to precisely inform the operator about what happened to the behavior.
`OperatorDecisionState`	Implements a state where the operator has to choose an outcome manually. Autonomy Level of all outcomes should be set to Full, since this state is not able to choose an outcome on its own. Only exception is the suggested outcome, which will be returned immediately by default. This state can be used to create alternative execution paths by setting the suggestion to High autonomy instead of Full.

`StartRecordLogsState`	A state that records the contents of the specified ROS topics in a bag file. Logging is done by creating a rosbag subprocess which is afterwards accessible using the output key rosbag_process. This state is typically combined with a StopRecordLogsState which gets the subprocess in order to stop logging.
`StopRecordLogsStates`	A state that records the contents of the specified ROS topics in a bag file.
`WaitState`	Implements a state that can be used to wait on timed process.

A.1.2 flexbe_atlas_states

State Class	**Description**
`ChangeControlModeActionState`	Implements a state where the robot changes its control mode using the action.
`CheckCurrentControlModeState`	Implements a state where the robot checks its current control mode.
`CurrentJointPositionsState`	Retrieves the current positions of the joints of the specified planning group.
`ExecuteStepPlanActionState`	Implements a state to execute a step plan of the footstep planner. This state will change the control mode of the robot to STEP during execution and expects to be in STAND when entered.
`ExecuteTrajectoryBothArmsState`	Executes trajectory(ies) passed from userdata.
`ExecuteTrajectoryMsgState`	Executes a given joint trajectory message.
`ExecuteTrajectoryState`	Executes a custom trajectory.
`FootstepPlanRealignCenterState`	Implements a state where the robot plans to centers its feet.
`FootstepPlanRelativeState`	Implements a state where the robot plans a relative motion, e.g., 2m to the left. Please note that the distance is only approximate, actual distance depends on exact step alignment.
`FootstepPlanTurnState`	Implements a state where the robot plans a motion to turn in place, e.g. 90 degree to the left. Please note that the angle is only approximate, actual distance depends on exact step alignment.
`FootstepPlanWideStanceState`	Implements a state where the robot plans to move into wide stance.
`GetCameraImageState`	Grabs the most recent camera image.
`GetLaserscanState`	Grabs the most recent laserscan data.
`GetTemplateGraspState`	Requests a grasp for a template from the template server.

`GetTemplatePoseState`	Requests the pose of a template from the template server.
`GetTemplatePregraspState`	Requests pregrasp information for a template from the template server.
`GetTemplateStandPoseState`	Requests the pose where to stand in front of a template from the template server.
`HandTrajectoryState`	Executes a given hand trajectory, i.e., a request to open or close the fingers.
`LoadTrajectoryFromBagfileState`	Implements a state that loads (arm) trajectories stored in a bagfile.
`LocomotionPosePerceptionState`	Extracts a pose of interest to walk to from environment data.
`MotionServiceState`	Implements a state where a certain motion is performed. This state can be used to execute motions created by the motion editor.
`MoveitCommanderMoveGroupState`	Uses moveit commander to plan to the given joint configuration and execute the resulting trajctory.
`MoveItMoveGroupPlanState`	Employs moveit Move Group to get a plan to a goal configuration, but does not execute it.
`MoveitMoveGroupState`	Uses moveit to plan to the given joint configuration and executes the resulting trajctory.
`MoveitStartingPointState`	Uses moveit to plan and move to the first point of a given arm trajectory.
`PickupObjectState`	Uses moveit to perform a pickup action.
`PlanEndeffectorCartesianState`	Plans a cartesian endeffector trajectory passing all given waypoints.
`PlanEndeffectorCircularState`	Plans a circular motion of the endeffector.
`PlanEndeffectorPoseState`	Plans to reach the given pose with the specified endeffector.
`PlanFootstepsState`	Creates a footstep plan to reach the specified waypoint from the current position.
`ReadDynamicParameterState`	Reads a given trajectory controller parameter set.
`SendToOperatorState`	Sends the provided data to the operator using a generic serialized topic.
`UpdateDynamicParameterState`	Updates a given trajectory controller parameter set.
`UpdateJointCalibrationState`	Updates the calibration offsets for the given joints.
`VideoLoggingState`	A state that controls video logging.

A.2 Evaluation of Behavior Logfiles

Each behavior execution automatically creates a new log file. Default location for these files is the `logs` folder inside the `home` folder. Independently from executing a behavior, these log files can be evaluated in order to derive information about execution performance regarding various aspects. An example evaluation is given in chapter 6.2. The evaluation script can be executed by running:

```
rosrun flexbe_widget evaluate_logs [logfile] [key] [value1, value2, ...]
```

`[logfile]` can be used to specify any log file. It can also be set to an empty string which will result in using the most recent log file in the default folder.

`[key]` refers to one of the choices listed in the table below and is used to accumulate the data of the single log entries in order to evaluate regarding specific categories.

Keys for Accumulation	
`total`	Accumulates all entries.
`state_path`	Uses the unique path of a state.
`state_name`	The name of each state is the keys, can be ambiguous.
`state_class`	Accumulate entries according to their implementation.
`transition`	Each transition is evaluated separately.

`[value1, value2, ...]` is any combination of the available indicators as listed below. Each of these indicators can be derived from the logged data and will be accumulated according to the specified key.

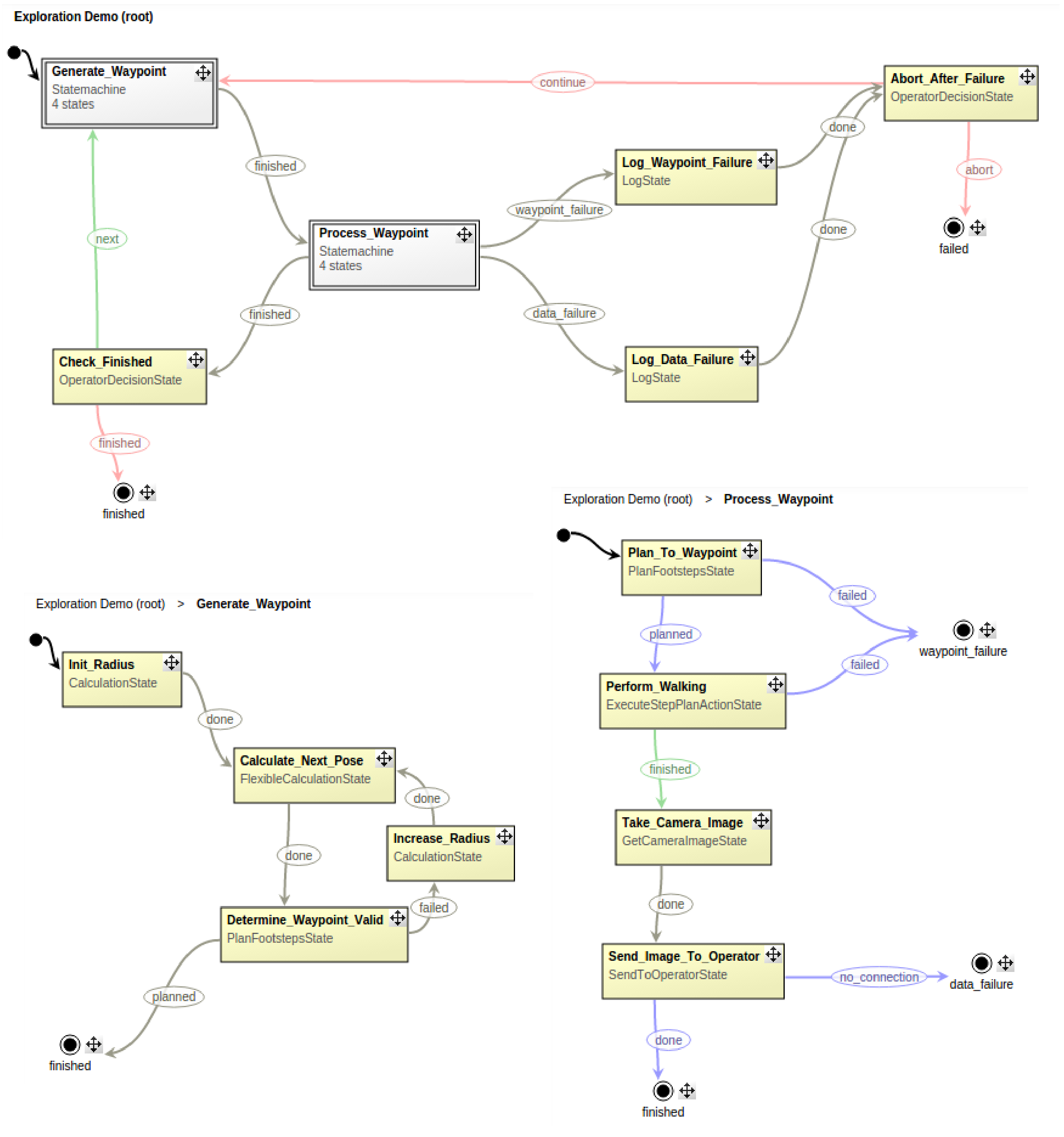
Available Performance Indicators	
`count_interventions`	Amount of manual interaction by the operator.
`count_wait`	How often execution has been stopped at this point.
`time`	Total time spent at this entry.
`time_wait`	Amount of time spent on waiting for the operator.
`time_run`	Effective amount of execution time.
`wrong_decisions`	How often the suggested outcome was declined by the operator.
`right_decisions`	How often the correct outcome has been taken.
`approved_decisions`	How often the suggested outcome was approved by the operator.
`visits`	Amount of visits of this entry.

A.3 Example State Implementation

```python
#!/usr/bin/env python

import actionlib
import rospy

from vigir_be_core import EventState, Logger
from vigir_be_core.proxy import ProxyServiceCaller, ProxyActionClient

from vigir_footstep_planning_msgs.msg import *
from vigir_footstep_planning_msgs.srv import *
from std_msgs.msg import *

'''
Created on 10/22/2014

@author: Philipp Schillinger
'''

class PlanFootstepsState(EventState):
    '''
    Creates a footstep plan to reach the specified waypoint from the current position.

    -- mode             string          One of the available planning modes (class constants).
    -- pose_is_pelvis   boolean         Set this to True if the waypoint is given
                                        as pelvis pose and not on the ground.

    ># target_pose      PoseStamped     Waypoint to which the robot should walk.

    #> footstep_plan    StepPlan        Footstep plan to reach the waypoint.

    <= planned                          Successfully created a plan.
    <= failed                           Failed to create a plan.

    '''

    MODE_STEP_2D = 'drc_step_2D'
    MODE_STEP_3D = 'drc_step_3D'
    MODE_STEP_NO_COLLISION = 'drc_step_no_collision'
    MODE_WALK = 'drc_walk'


    def __init__(self, mode, pose_is_pelvis = False):
        '''
        Constructor
        '''
        super(PlanFootstepsState, self).__init__(outcomes=['planned', 'failed'],
                                                 input_keys=['target_pose'],
                                                 output_keys=['footstep_plan'])

        self._action_topic = "/vigir/footstep_planning/step_plan_request"
        self._feet_pose_srv = "/vigir/footstep_planning/generate_feet_pose"

        self._client = ProxyActionClient({self._action_topic: StepPlanRequestAction})

        self._srv = ProxyServiceCaller({self._feet_pose_srv: GenerateFeetPoseService})

        self._failed = False
        self._mode = mode
        self._pose_is_pelvis = pose_is_pelvis



    def execute(self, userdata):
        '''
        Execute this state
        '''
        if self._failed:
            userdata.footstep_plan = None
            return 'failed'

        if self._client.has_result(self._action_topic):
            result = self._client.get_result(self._action_topic)
            if result.status.warning != 0:
                Logger.logwarn('Planning footsteps warning:\n%s' % result.status.warning_msg)

            if result.status.error == 0:
                userdata.footstep_plan = result.step_plan
                return 'planned'
            else:
                userdata.footstep_plan = None
                Logger.logerr('Planning footsteps failed:\n%s' % result.status.error_msg)
                return 'failed'


    def on_enter(self, userdata):
        self._failed = False

        current_feet = None
        target_feet = None

        # get start
        try:
            msg_start = FeetPoseRequest(flags=FeetPoseRequest.FLAG_CURRENT)
            msg_start.header = Header(frame_id='/world')
            current_feet = self._srv.call(self._feet_pose_srv, msg_start)
        except Exception as e:
            Logger.logwarn('Failed to get current feet poses:\n%s' % str(e))
            self._failed = True
            return

        # get goal
        try:
            header_value = userdata.target_pose.header
            pose_value = userdata.target_pose.pose
            flags_value = FeetPoseRequest.FLAG_PELVIS_FRAME if self._pose_is_pelvis else 0
            msg_goal = FeetPoseRequest(header=header_value, pose=pose_value, flags=flags_value)
            target_feet = self._srv.call(self._feet_pose_srv, msg_goal)
        except Exception as e:
            Logger.logwarn('Failed to get target feet poses:\n%s' % str(e))
            self._failed = True
            return

        # create action goal msg
        action_goal = StepPlanRequestGoal()
        request_msg = StepPlanRequest()
        request_msg.start = current_feet.feet
        request_msg.goal = target_feet.feet
        request_msg.parameter_set_name = String(self._mode)
        request_msg.header = Header(frame_id='/world')
        action_goal.plan_request = request_msg

        try:
            self._client.send_goal(self._action_topic, action_goal)
        except Exception as e:
            Logger.logwarn('Was unable to create footstep planner plan request:\n%s' % str(e))
            self._failed = True
```

Listing 7.1: Implementation of the state to plan footsteps using *Team ViGIR*'s footstep planner.

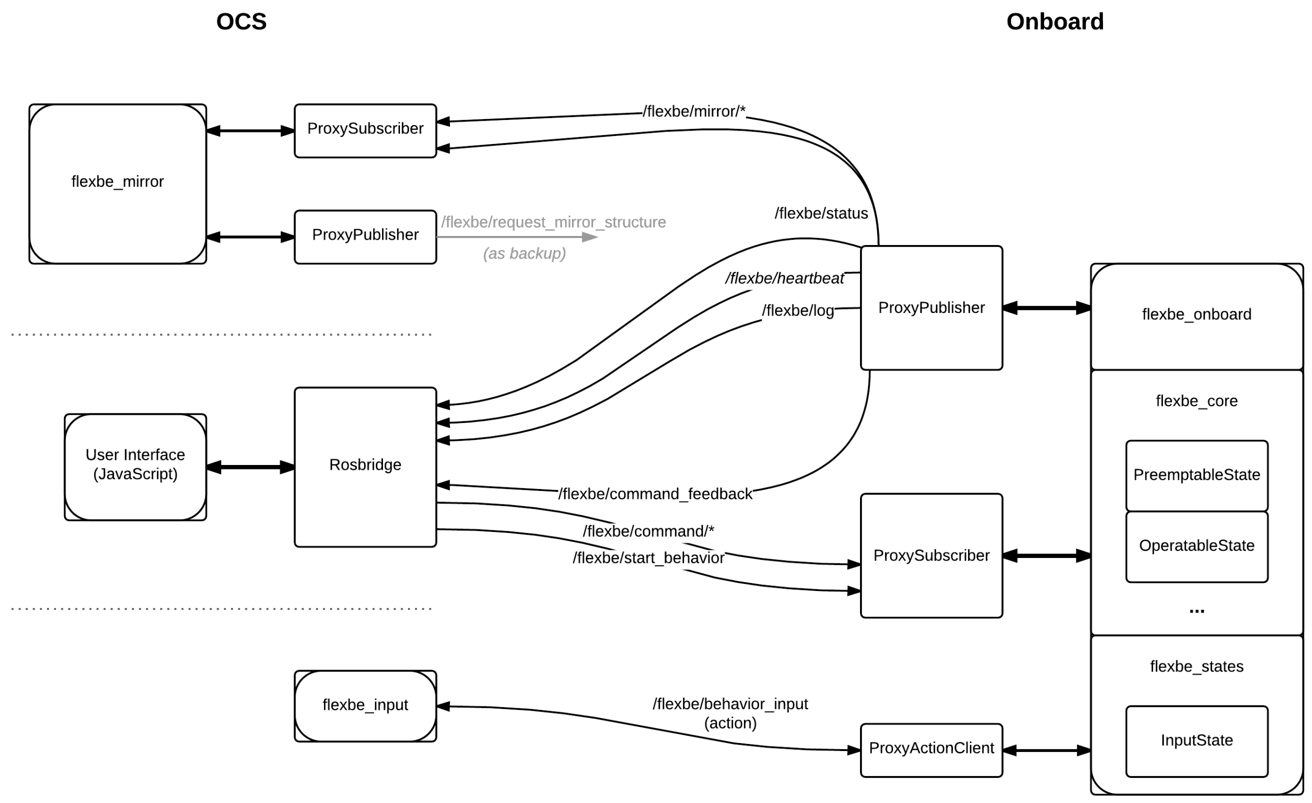
A.4 Exploration Demo Behavior



The *Exploration Demo* behavior including its two inner state machines as displayed in *FlexBE*. This is the original version where the robot would take camera images and send these to the operator. During execution in the scenario as described in chapter 6.2, the operator replaced the `Take_Camera_Image` state by a state to record a LIDAR scan.

A.5 Communication Channel Topics

The following figure gives an overview of all ROS topics involved in communicating between the onboard behavior engine and the ROS nodes running at the operator control station.



The following command and mirror topics are in use:

`/flexbe/command/*`	`/flexbe/mirror/*`
`autonomy`	`outcome`
`lock`	`preempt`
`preempt`	`structure`
`transition`	`sync`
`unlock`	

`/flexbe/heartbeat` is the only topic which is published at a certain rate. One empty message per second is sent in order to let the operator know that the behavior engine is still running. All other topics are only used if a command is sent (`/flexbe/command/*`), a message is logged onboard and should be displayed to the operator (`/flexbe/log`), the behavior starts, finishes, or fails (`/flexbe/status`), or a transition is executed (`/flexbe/mirror/outcome`), to list the most common ones. Each sent data only contains minimal information. A transition notification for example only provides the index (`uint8`) of the returned outcome. This information can be interpreted by the *Behavior Mirror* in order to provide context information. The command messages, to give another example, only need to contain their required arguments. The type of command is implicitly specified by using separate topics.

A.6 FlexBE Tutorials

The following sections contain basic tutorials for getting started with using the *FlexBE* user interface. As soon as *FlexBE* is publicly released, it is planned to add more detailed and in-depth tutorials to the *FlexBE* website, as well as an online sandbox version of the user interface to play around with. However, since the *FlexBE* user interface is designed to be usable for both, non-expert developers and operators, exploring further features of *FlexBE* after completion of the given basic tutorials is a viable option.

Installation: Behavior Engine

In order to install the ROS packages forming the behavior engine, create a new folder inside your ROS workspace and clone the *FlexBE* repository to this folder. Next, you need to create at least one additional ROS package and most likely more. It is recommended, although not required, to create a second folder next to the folder containing the content of the *FlexBE* repository and make it a repository on its own.

Now create a new ROS package called `flexbe_behaviors` and add two folders. The first folder is called `behaviors` and will later contain all behavior manifests, the second folder is called `config` and is the recommended location for storing configuration files.

In addition, you can create further ROS packages containing newly developed states. The naming convention for these packages is `flexbe_custom_states` where `custom` is replaced by your system identifier, e.g., the robot type or required software.

Finally, it is recommended to add a folder to your repository where you can later store all ROS packages implementing behaviors.

Installation: FlexBE Chrome App

Make sure you have *rosbridge* installed and install the *FlexBE Chrome App* just like any other *Chrome App*[1]. If you use the app in development mode instead of the compiled version, make sure to add `export FLEXBE_ID="[app_id]"` to your `.bashrc` file and substitute `[app_id]` with the unique ID assigned to this app by *Chrome* as displayed in the *Extensions* page in *Chrome*.

When you use the app for the first time, you need to configure your working environment. Navigate to the *Configuration* view and set the workspace references in the *Workspace* panel. The entry *Behaviors folder* should point to the folder where you want to store all ROS packages implementing behaviors. The entry *flexbe_behaviors folder* should point to your newly created `flexbe_behaviors` ROS package.

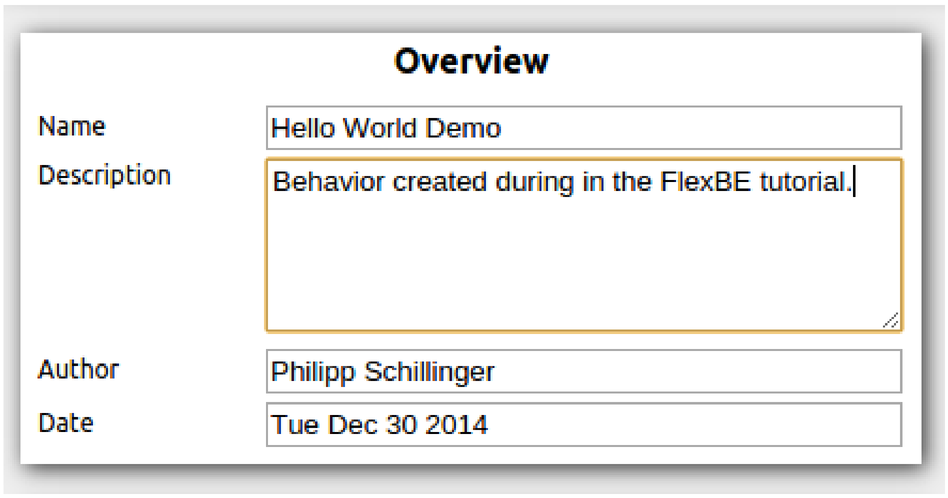
Furthermore, add folders to the state library of the editor by clicking *Add Folder* in the panel *State Library*. You may want to add at least the folder inside the `flexbe_states` package containing all the generic states. Make sure to add the folder which contains the states, not the ROS package itself.

[1] `https://support.google.com/chrome_webstore/answer/1053369?hl=en` *(04/08/2015)*

A.6.1 Creating a New Behavior

In this tutorial, and in the next one, we will create a very simple demo behavior to demonstrate the steps required for behavior creation.

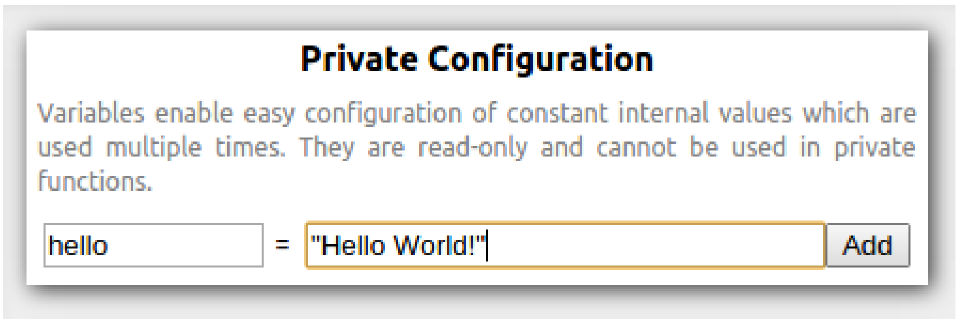
Launch the *FlexBE Chrome App* and start with giving your new behavior a name, describe it, and enter your own name as author.



Overview

Name	Hello World Demo
Description	Behavior created during in the FlexBE tutorial.
Author	Philipp Schillinger
Date	Tue Dec 30 2014

We would like our new behavior to print a `"Hello World!"` message when executing it. In the *Private Configuration* section of the dashboard, constant values of the behaviors can be defined. This serves as a single point of change for any value required by the behavior. So let's define a new string named hello for our message.



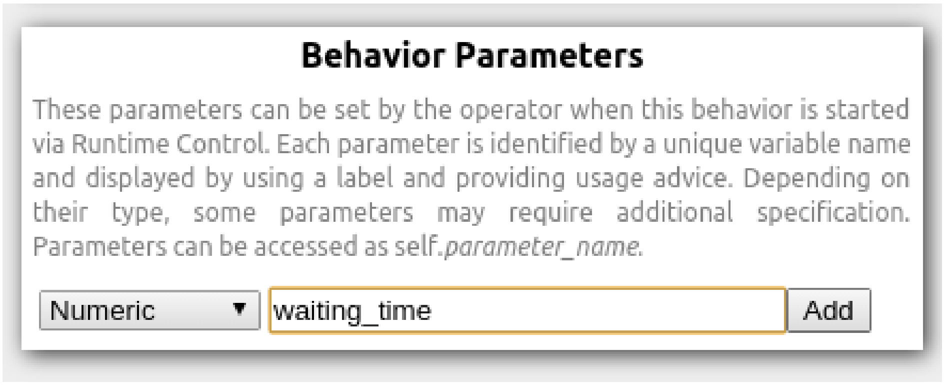
Private Configuration

Variables enable easy configuration of constant internal values which are used multiple times. They are read-only and cannot be used in private functions.

hello = "Hello World!"

Add the new string by clicking the Add button. We will use it later.

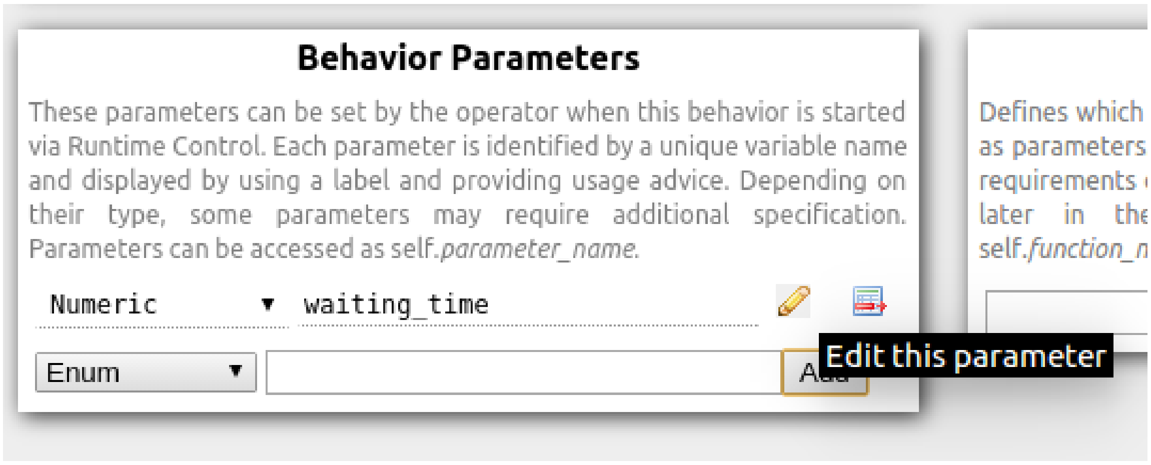
Also, we want our behavior to wait some time before printing the message. However, the time to wait should not be constant. Instead, we want the operator to select a value when he starts execution of the behavior. For this purpose, *Behavior Parameters* are available. Select `Numeric` as type of the new parameter and name it `waiting_time`.



Behavior Parameters

These parameters can be set by the operator when this behavior is started via Runtime Control. Each parameter is identified by a unique variable name and displayed by using a label and providing usage advice. Depending on their type, some parameters may require additional specification. Parameters can be accessed as self.*parameter_name*.

Add the parameter and edit its properties by clicking on the pencil icon.

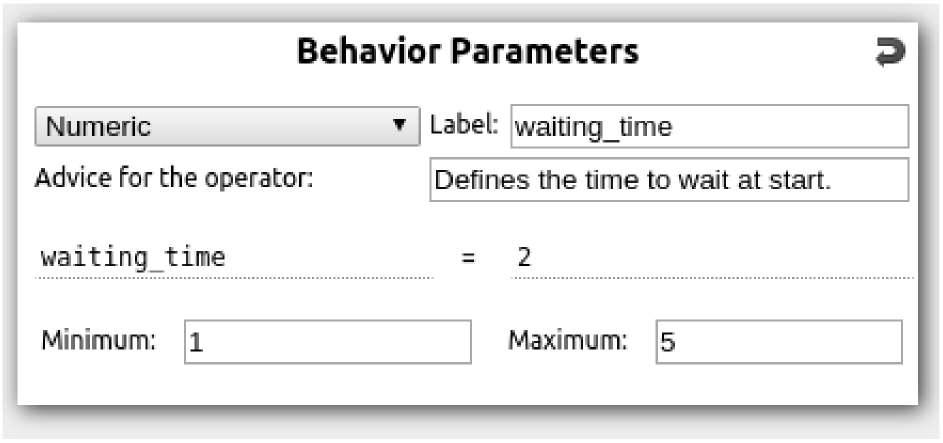


Behavior Parameters

These parameters can be set by the operator when this behavior is started via Runtime Control. Each parameter is identified by a unique variable name and displayed by using a label and providing usage advice. Depending on their type, some parameters may require additional specification. Parameters can be accessed as self.*parameter_name*.

Defines which as parameters requirements later in the self.*function_n*

You can now enter reasonable values for the waiting time as shown below. Unit is seconds, because we will use this value later to pass it to a state which waits for the given amount of seconds.



Behavior Parameters

Numeric Label: waiting_time

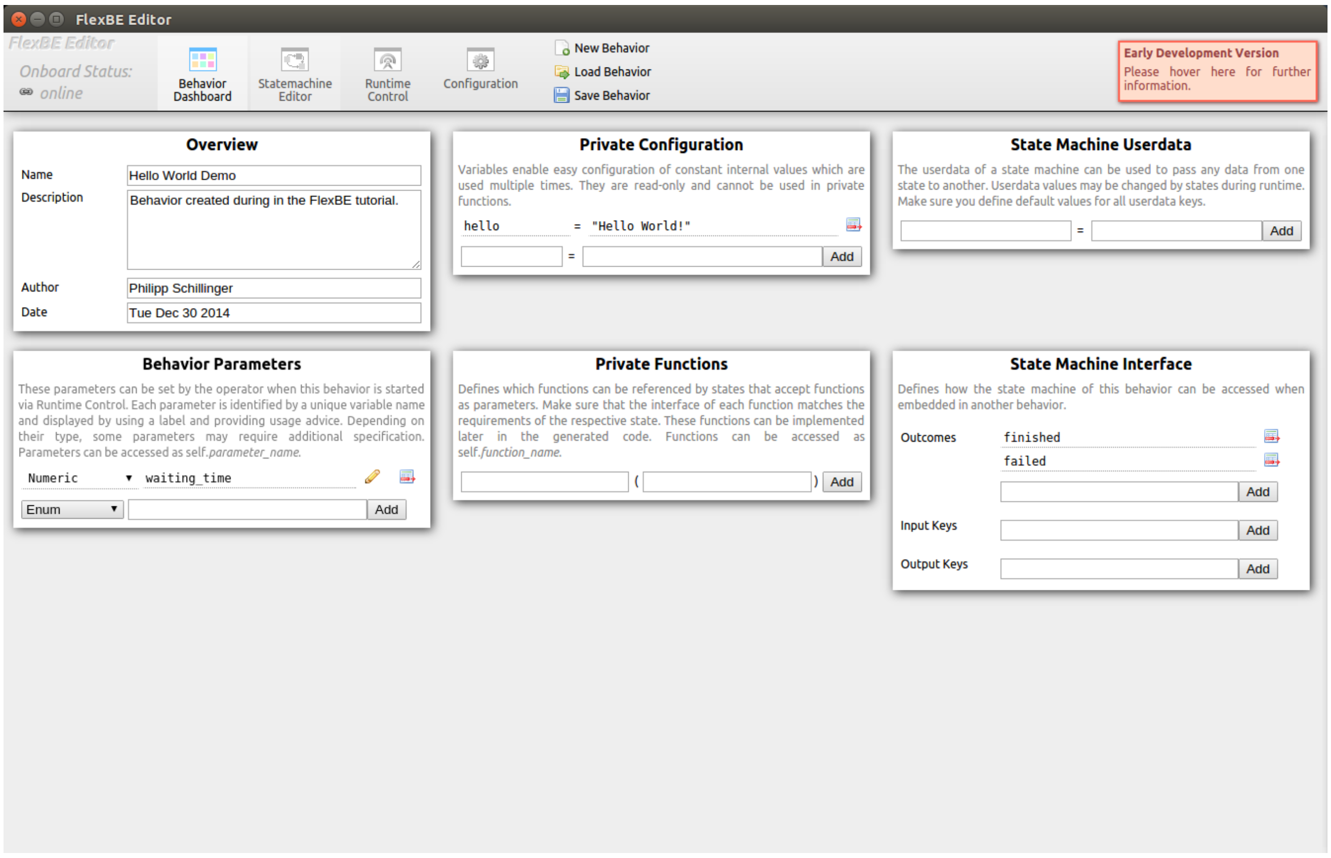
Advice for the operator: Defines the time to wait at start.

waiting_time = 2

Minimum: 1 Maximum: 5

To leave the property view of the parameter and have the list of all parameters displayed again, click on the arrow icon at the top of this box.

We started creating our new behavior by defining some values (one constant, one variable) and can now proceed to define the state machine of the behavior. Your *Behavior Dashboard* should now look like shown below. Click *Statemachine Editor* at the top to switch to the editor and continue with the next tutorial.



Early Development Version
Please hover here for further information.

Overview

Name	Hello World Demo
Description	Behavior created during in the FlexBE tutorial.
Author	Philipp Schillinger
Date	Tue Dec 30 2014

Private Configuration

Variables enable easy configuration of constant internal values which are used multiple times. They are read-only and cannot be used in private functions.

```
hello = "Hello World!"
```

State Machine Userdata

The userdata of a state machine can be used to pass any data from one state to another. Userdata values may be changed by states during runtime. Make sure you define default values for all userdata keys.

Behavior Parameters

These parameters can be set by the operator when this behavior is started via Runtime Control. Each parameter is identified by a unique variable name and displayed by using a label and providing usage advice. Depending on their type, some parameters may require additional specification. Parameters can be accessed as self.*parameter_name*.

```
Numeric ▾ waiting_time
```

Enum ▾

Private Functions

Defines which functions can be referenced by states that accept functions as parameters. Make sure that the interface of each function matches the requirements of the respective state. These functions can be implemented later in the generated code. Functions can be accessed as self.*function_name*.

State Machine Interface

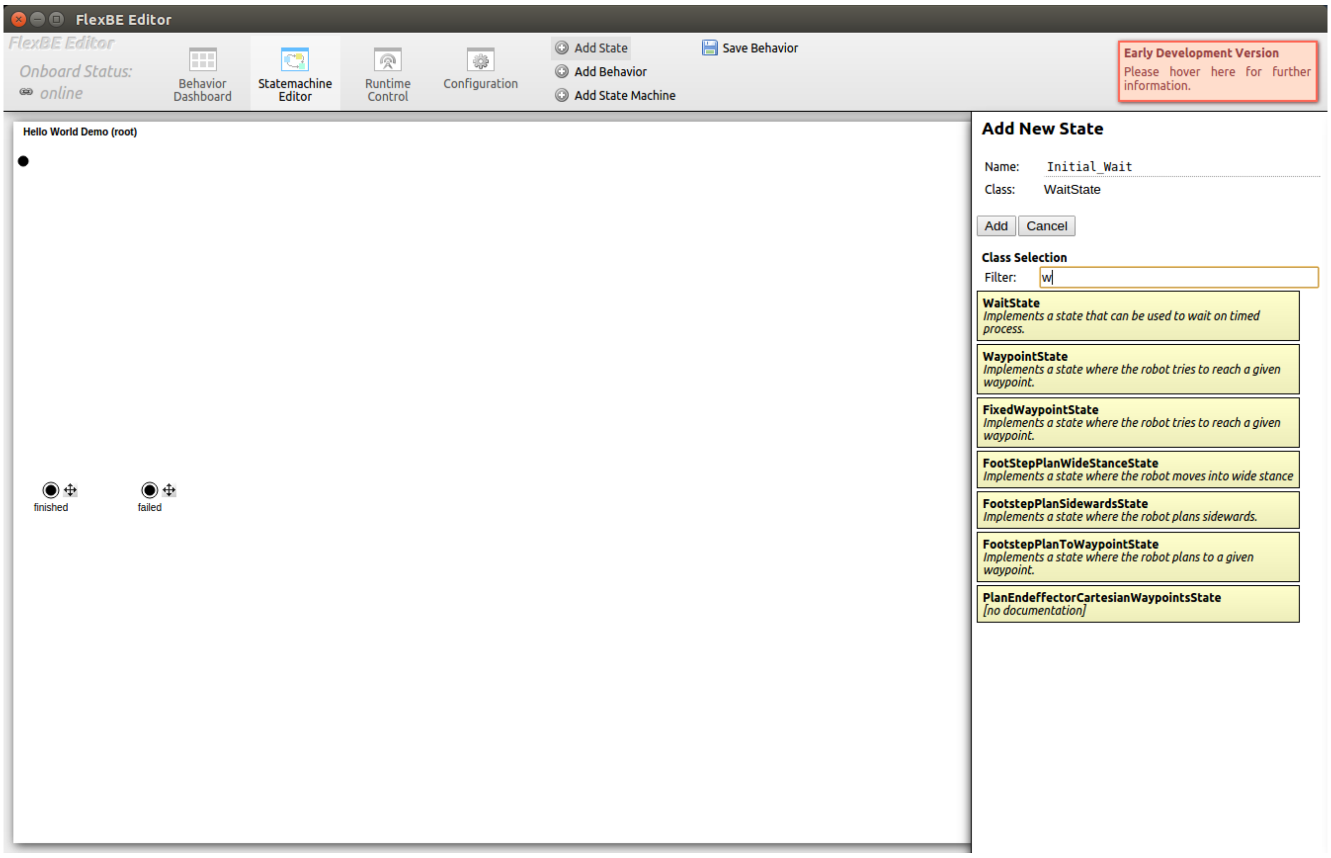
Defines how the state machine of this behavior can be accessed when embedded in another behavior.

Outcomes	finished
	failed
Input Keys	
Output Keys	

A.6.2 Using the Statemachine Editor

Start implementing the behavior by adding its first state. As written in the first tutorial, we want our behavior to wait for some time and then print a message. So our first state will be a state which waits a given amount of seconds.

Click *Add State* at the top, enter a name for the new state and select `WaitState` from the list below. Entering something like "wait" in the class filter will help to quickly find the state and would also help if we didn't know the exact name and just need any state for waiting.



Early Development Version
Please hover here for further information.

Add New State

Name: Initial_Wait
Class: WaitState

Class Selection
Filter: w

WaitState
Implements a state that can be used to wait on timed process.

WaypointState
Implements a state where the robot tries to reach a given waypoint.

FixedWaypointState
Implements a state where the robot tries to reach a given waypoint.

FootStepPlanWideStanceState
Implements a state where the robot moves into wide stance

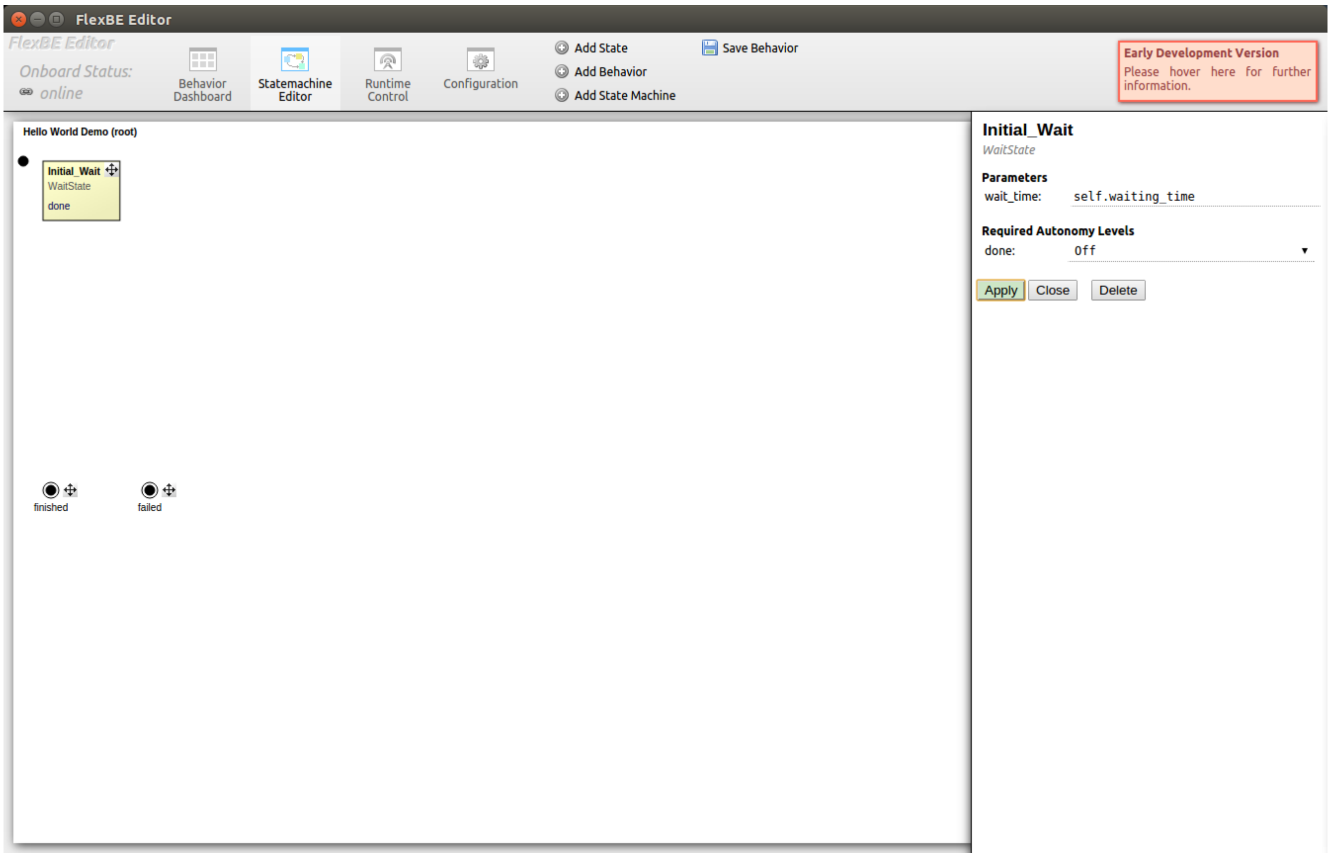
FootstepPlanSidewardsState
Implements a state where the robot plans sidewards.

FootstepPlanToWaypointState
Implements a state where the robot plans to a given waypoint.

PlanEndeffectorCartesianWaypointsState
[no documentation]

Now click *Add* to add the new state to the behavior state machine.

The properties of the newly added state will pop up. This panel can also be displayed by clicking on the state. Since we already defined a parameter for the waiting time, we can use it now. References to parameters always start with a `self.` in front of the parameter name. This helps to distinguish between parameters and constants. However, you can just start typing `waiting_time` and let autocompletion do the rest.



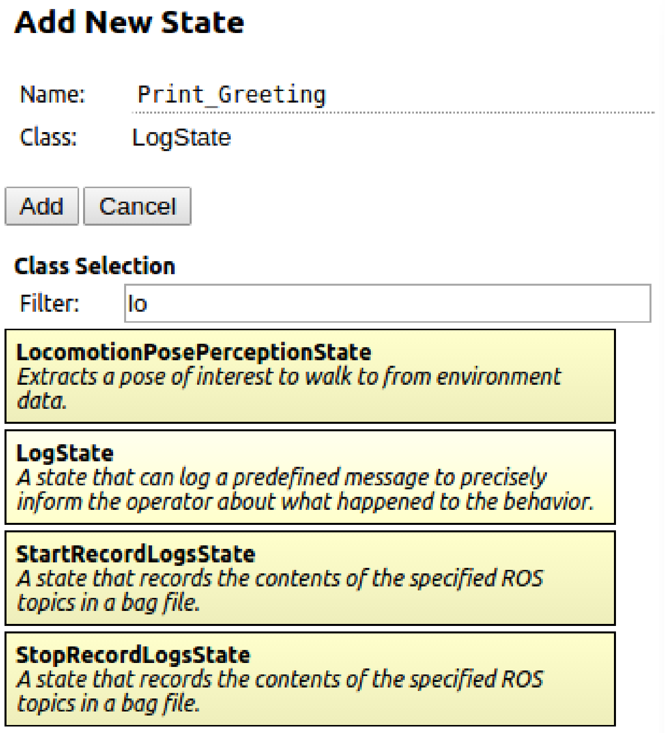
Initial_Wait
WaitState

Parameters
wait_time: self.waiting_time

Required Autonomy Levels
done: Off

Click *Apply* to make this change permanent.

Next, we add our second state the same way. This time, it's a `LogState` as shown below because now we want to print our message.


Add New State

Name: Print_Greeting
Class: LogState

Class Selection
Filter: lo

LocomotionPosePerceptionState
Extracts a pose of interest to walk to from environment data.

LogState
A state that can log a predefined message to precisely inform the operator about what happened to the behavior.

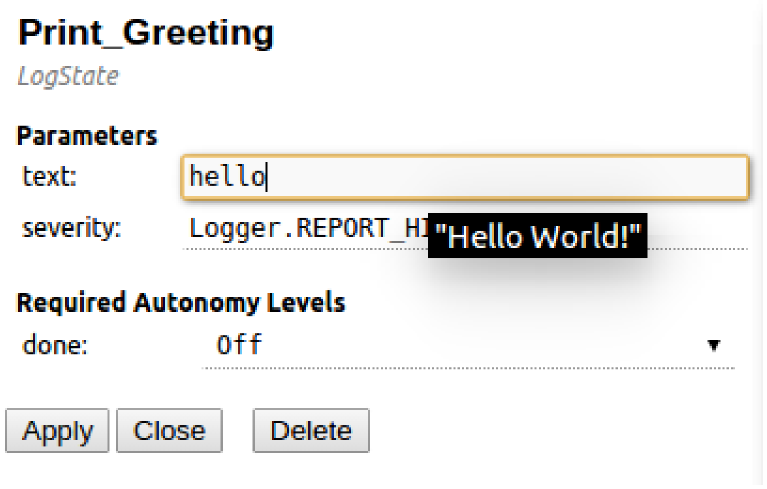
StartRecordLogsState
A state that records the contents of the specified ROS topics in a bag file.

StopRecordLogsState
A state that records the contents of the specified ROS topics in a bag file.

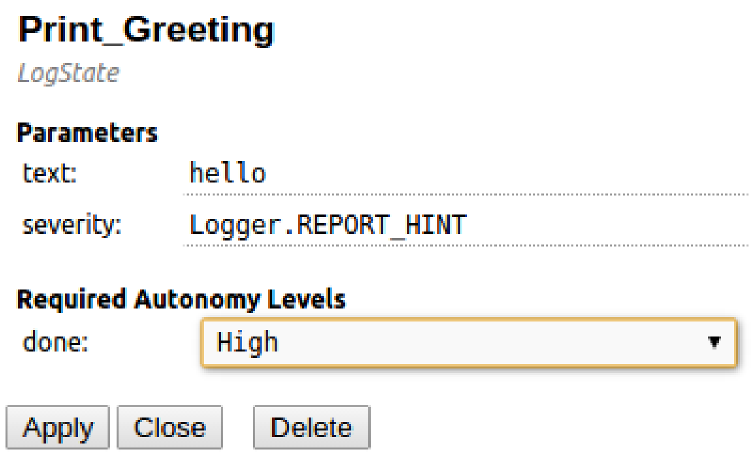
This state has two parameters which we can now edit in the properties panel.

The first one is the text to be printed. As we did for the waiting state, we won't enter an explicit value, but instead refer to one of our already defined values. So, just enter the name of our message string, `hello`, as value of the text parameter. As soon as you click *Apply*, you will also see its value as tooltip when hovering over the text field.

The second parameter defines the type of the message such as error or warning. The default value should be fine this time. Normally, behavior hints are used to give explicit instructions or reminders to the operator when he is expected to do something.


Print_Greeting
LogState
Parameters
text: hello
severity: Logger.REPORT_HI "Hello World!"
Required Autonomy Levels
done: Off

This time, we will also change the required level of autonomy of this state's outcome. Select `High` and apply the change. The meaning of this *Autonomy Level* is explained in the next tutorial.

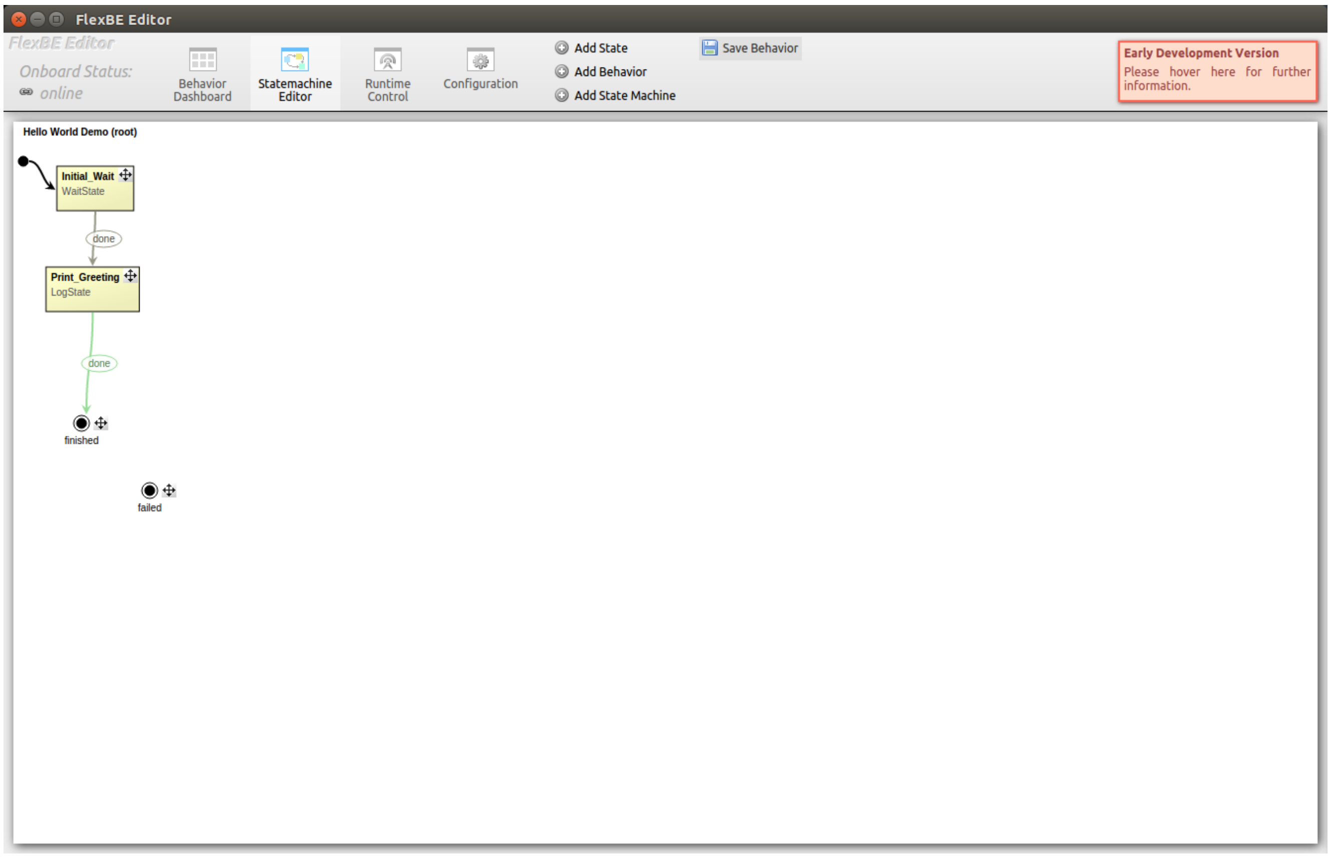

Print_Greeting
LogState
Parameters
text: hello
severity: Logger.REPORT_HINT
Required Autonomy Levels
done: High

Now connect the added states.

Start with setting the initial state by clicking on the black bullet next to the waiting state first, and then clicking on the waiting state to connect it to the bullet. You may also change the position of the states by dragging them at their top right corner (indicated by a move icon).

The unconnected outcomes of each state are listed at the bottom of each state's visualization. First, click on the outcome you want to connect and then click on the target. The two bullets at the bottom of the state machine are the outcomes of the state machine itself. You can ignore the outcome `failed` for now.

Your state machine should now look similar to the one shown below.



The green color of the second transition refers to its required autonomy level.

Now save the behavior. This will generate all files required for executing the new behavior.

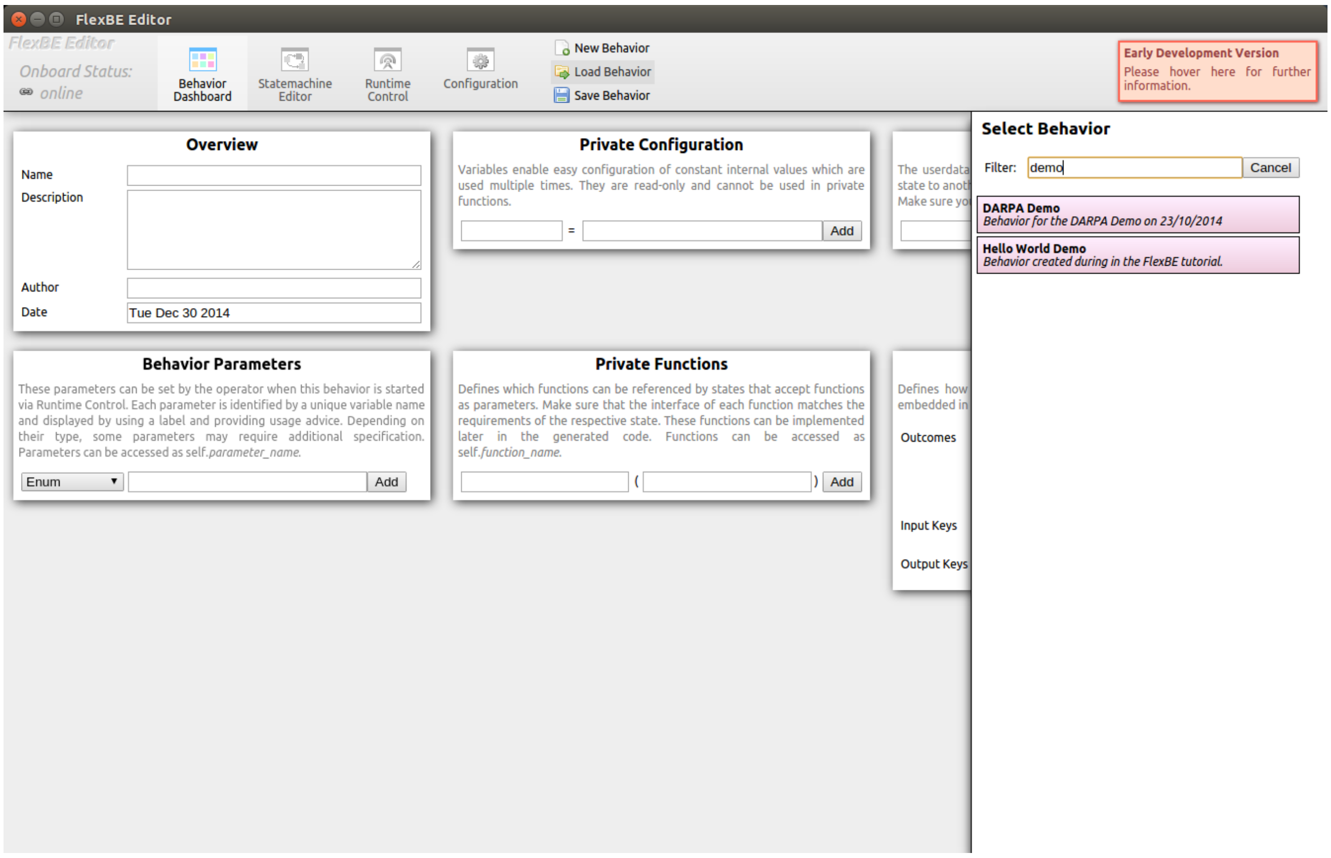
Since you added a new ROS package by creating a new behavior, *catkin* needs to configure this new package. In order to do so, close the *Chrome App* after saving and run `catkin_make`. This step is only required when adding a new behavior. Changes to a behavior do not require this process.

A.6.3 Execution of a Behavior

You should have completed the first two tutorials or at least have access to the created behavior. Now we want to execute a behavior. This means, it is no longer sufficient to just launch the *FlexBE Chrome App*. Instead, we need to launch the whole behavior engine. This can be done by executing the following command:

```
roslaunch flexbe_onboard behavior_testing.launch
```

First, you have to load the previously created behavior. This is done by clicking on the *Load Behavior* button at the top and then selecting your `Hello World Demo` in the list of available behaviors. Similar to choosing a class when adding a new state, you can again start typing parts of the name into the filter text field to quickly find what you are looking for.



Date: Tue Dec 30 2014

Variables enable easy configuration of constant internal values which are used multiple times. They are read-only and cannot be used in private functions.

The userdata state to anot Make sure yo

These parameters can be set by the operator when this behavior is started via Runtime Control. Each parameter is identified by a unique variable name and displayed by using a label and providing usage advice. Depending on their type, some parameters may require additional specification. Parameters can be accessed as self.*parameter_name*.

Defines which functions can be referenced by states that accept functions as parameters. Make sure that the interface of each function matches the requirements of the respective state. These functions can be implemented later in the generated code. Functions can be accessed as self.*function_name*.

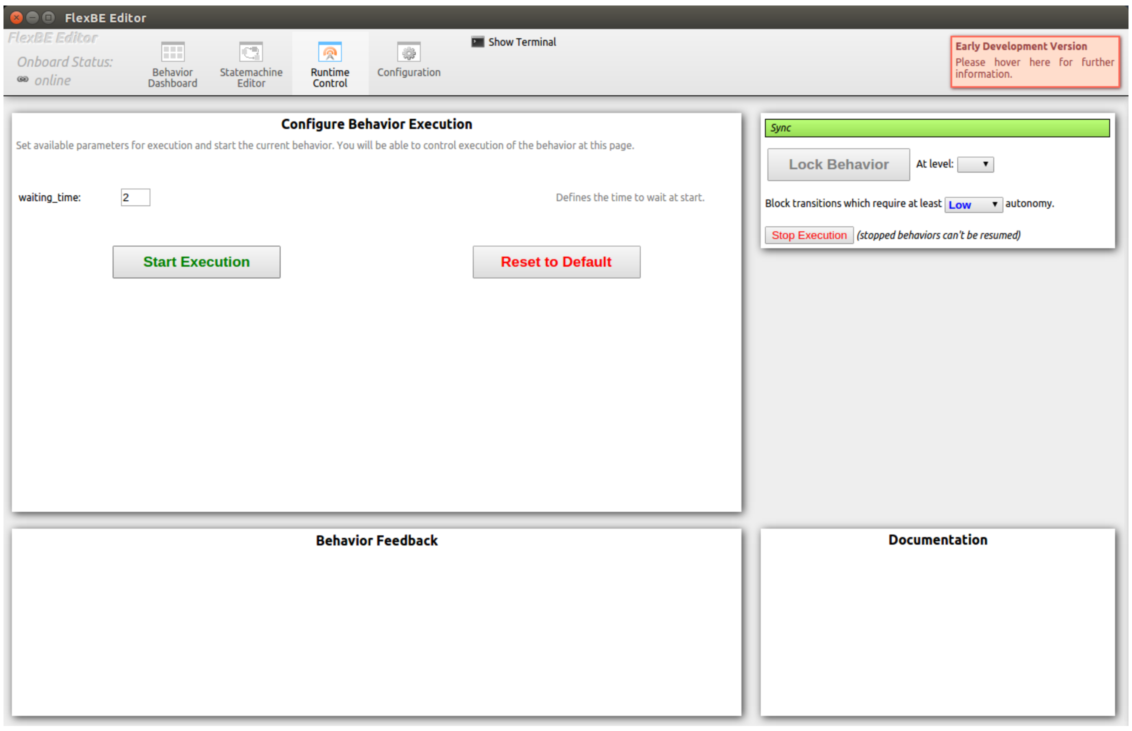
Defines how embedded in

Filter: demo

DARPA Demo
Behavior for the DARPA Demo on 23/10/2014

Hello World Demo
Behavior created during in the FlexBE tutorial.

Now, switch to the *Runtime Control* view of the user interface. It should look like shown below.



Early Development Version
Please hover here for further information.

Configure Behavior Execution

Set available parameters for execution and start the current behavior. You will be able to control execution of the behavior at this page.

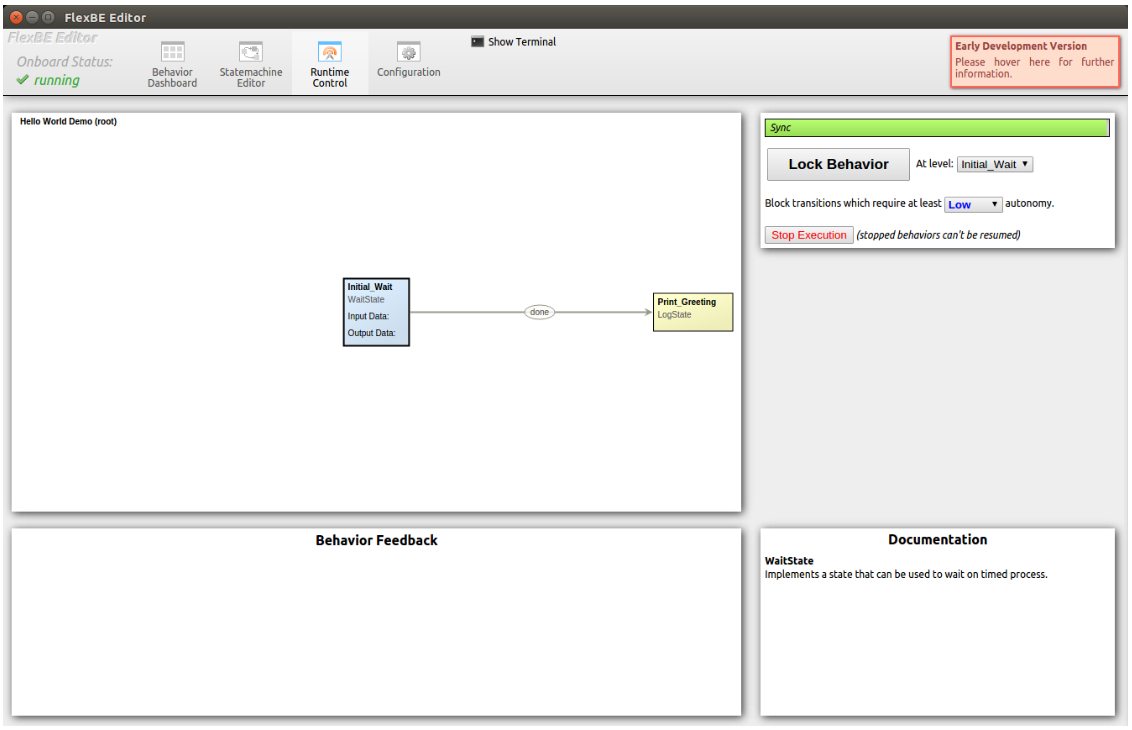
waiting_time: 2 Defines the time to wait at start.

At level:

Block transitions which require at least Low autonomy.

(stopped behaviors can't be resumed)

In order to run the behavior, now click the green *Start Execution* button. As soon as you click the button, several things will happen. The user interface informs the onboard engine as well as the behavior mirror. The mirror will create a behavior with the same structure as the onboard behavior in order to provide meaningful information by just receiving minimal data from onboard, while the onboard engine now imports the sent behavior, builds it, and starts execution. When everything is running as it should, the main panel will display the first state of our behavior as the currently active state.



Early Development Version
Please hover here for further information.

At level: Initial_Wait

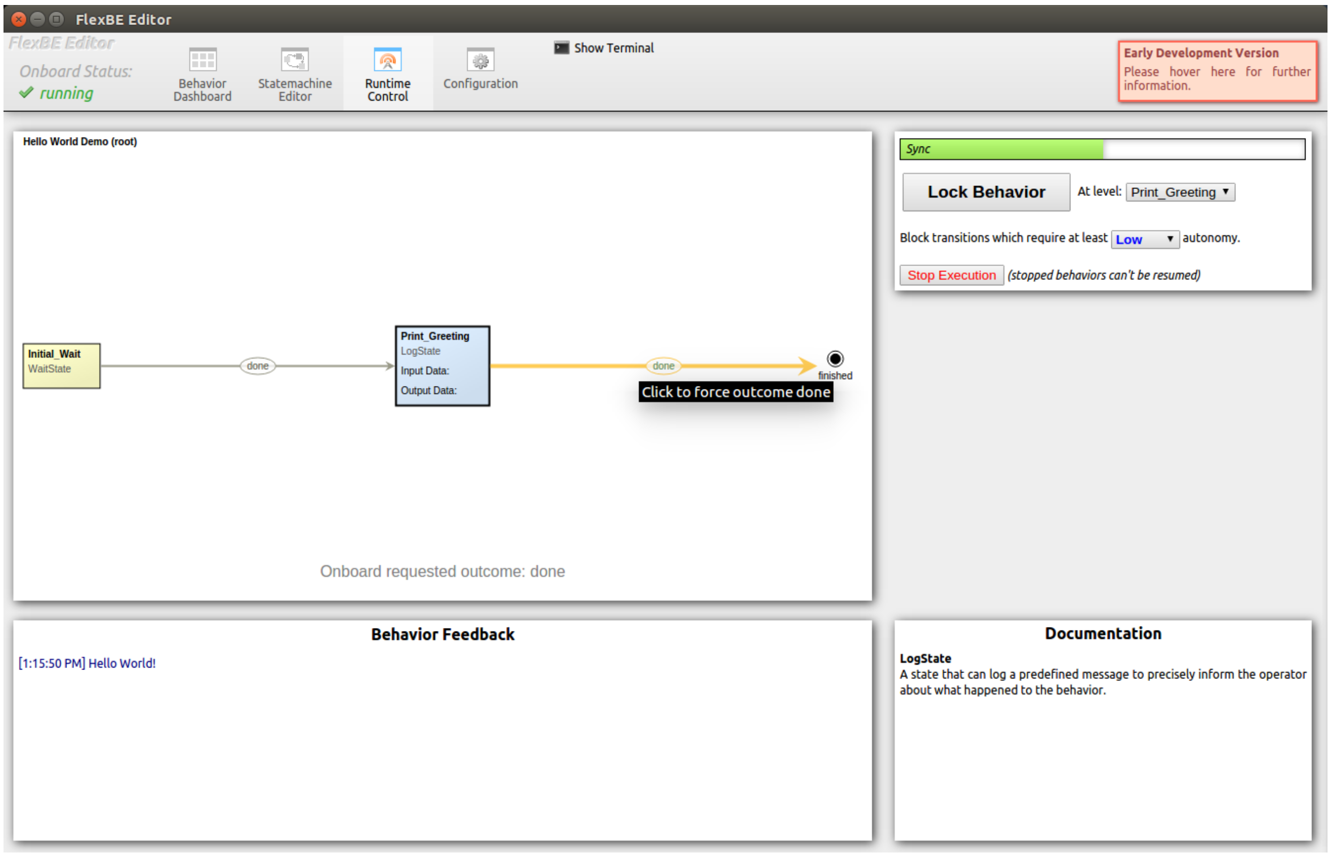
Block transitions which require at least Low autonomy.

(stopped behaviors can't be resumed)

WaitState
Implements a state that can be used to wait on timed process.

The active state is always displayed in the center of the panel in blue color. On the right, you can see possible outcomes of the state including the state which would be executed next.

After two seconds, the behavior continues with the next state.



At level: Print_Greeting

Block transitions which require at least Low autonomy.

(stopped behaviors can't be resumed)

Onboard requested outcome: done

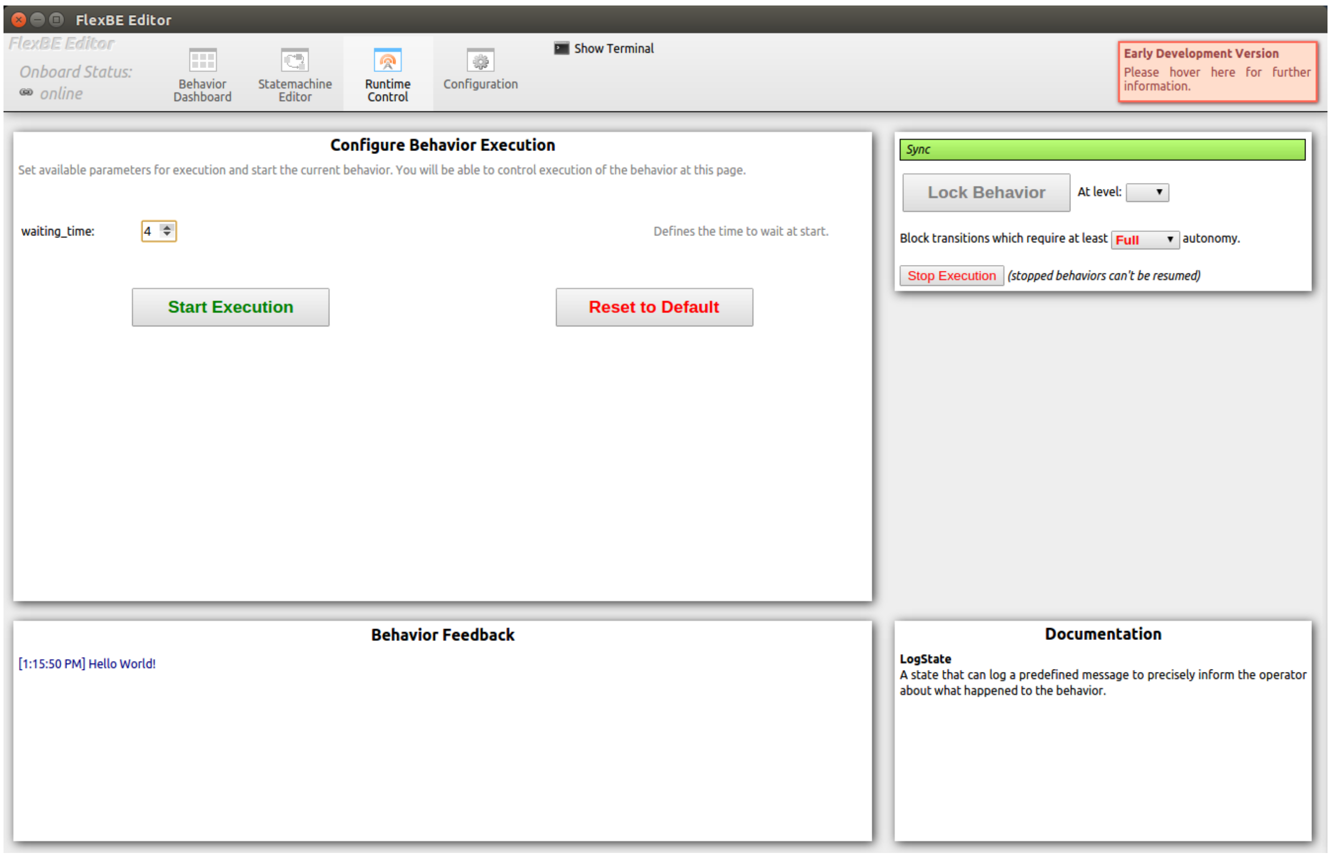
[1:15:50 PM] Hello World!

LogState
A state that can log a predefined message to precisely inform the operator about what happened to the behavior.

As you can see now, the behavior does not continue after reaching the second state. Instead, it highlights the outcome and asks for permission to execute it. We will allow it in a moment, but first, take a look at the bottom panels. You can now see the *Hello World* message displayed in the feedback panel, as well as a documentation of the `LogState` in the documentation panel.

Allow the requested outcome by clicking on the highlighted arrow. You can click on outcomes at any time, even if they are not highlighted, in order to force them. The behavior will immediately return outcomes requested in this way, so it is your responsibility to decide whether an outcome should be forced. After permitting the outcome, the demo behavior finishes.

Let's do a second run of this behavior, but this time, we change the default settings. Increase the waiting time to four seconds and change the *Autonomy Level* to *Full* by selecting the respective entry at the control panel. Changing the *Autonomy Level* can be done at any time, no matter if a behavior is running or not.



Early Development Version
Please hover here for further information.

Configure Behavior Execution

Set available parameters for execution and start the current behavior. You will be able to control execution of the behavior at this page.

waiting_time: 4 Defines the time to wait at start.

At level:

Block transitions which require at least Full autonomy.

(stopped behaviors can't be resumed)

Behavior Feedback

[1:15:50 PM] Hello World!

Documentation

LogState
A state that can log a predefined message to precisely inform the operator about what happened to the behavior.

The *Autonomy Level* was the reason why behavior execution stopped during the first run. We just allowed transitions requiring less than low autonomy which only applies for the most basic transitions. Each transition that required at least low autonomy was blocked and our second transitions requires high autonomy as we configured in the last tutorial. This means, as observed, the behavior will not execute the transition autonomously, but instead indicates that it is ready to be executed and waits for permission. Since we now set the autonomy to *Full*, behavior execution does not stop at the transition this time.