MASTER

RoboSC
a domain-specific language for supervisory controller synthesis of ROS-based applications

Wesselink, Bart B.A.

Award date:
2022

Department of Mathematics and Computer Science
Software Engineering and Technology (SET)

RoboSC: a domain-specific language for supervisory controller synthesis of ROS-based applications

Master Thesis

Bart Wesselink
(b.b.a.wesselink@student.tue.nl)

Supervisors:
dr. Ivan Kurtev
dr. Elena Torta

Version 1.0

Eindhoven, July 2022

Abstract

Safety is a key concern for the development of robots. They should operate as expected, not have any deadlocks, livelocks or cause accidents. In robotic applications, supervisory controllers determine which discrete actions a robot should execute based on the current state and data. The process of manually creating supervisory controllers makes it very difficult to verify their correctness with respect to requirements, like the absence of deadlocks and livelocks. In this thesis, a domain-specific language (DSL), RoboSC, is devised that can be used to develop a robotic supervisory controller for a robotic middleware. The language allows users to model concepts from supervisory control theory together with communication concepts from the robotic middleware ROS. This, in turn, allows users to generate models for supervisory control theory to which controller synthesis can be applied, as well as the platform binding code for both ROS as well as ROS2, which allows users to integrate the controller into ROS-based applications, without the user having to know anything about it. This speeds up the entire controller development chain while avoiding errors. An evaluation of the language is presented by simulating eight robotic scenarios for which a supervisory controller was developed using the DSL. The execution of the controller, as well as the generated bindings, were verified on both simulated as well as physical hardware.

Acknowledgements

This thesis marks the end of my Master in Computer Science at the University of Eindhoven. Although last two years meant that on-campus activities were limited due to all restrictions, it still has been a very joyful and educational experience.

First of all, I would like to thank my supervisors: Ivan Kurtev and Elena Torta. Their guidance has been excellent, and they provided me with an extremely collaborative environment, always open to new ideas. Their insights were key to what is presented in this thesis, and I would like to thank them for that.

Furthermore, special thanks to everyone that helped me in any way during the project. Sometimes by providing me with new insights, or sometimes by just listening to problems that I encountered, and helping to solve them.

Lastly, I thank my family, friends and colleagues for all their support over the years. They always had faith that everything would be alright with respect to my education, and let me do my thing. Thank you for that.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

There are many different applications for robotics. Robots can be found in places like factories, hospitals and nursing homes. It is very important that these robots function correctly, and behave as intended. Failing to do so might cause huge safety problems, as these robots often work in dynamic environments with people around. The development cycle of such a robot frequently starts with the development of a model, which is transformed into software. Then, the software is verified after implementation. If something was found to be incorrect, the model is altered and the process is starting all over again.

To supervise the behaviour of a robot, engineers can make use of supervisory controllers. These supervisory controllers update their state based on individual events coming from the different components of the robot, like distance sensors. Depending on the state that the robot is in, supervisory controllers restrict the allowed behaviour of such a robot.

Software for robots can be developed using a robotic middleware like ROS, the robotic operating system. This middleware enables communication between different parts of the robot. This has several advantages, as it allows for the individual development of robotic components. Such a component (or node) could contain the logic for communicating with a motor or a distance sensor, for example.

One of the ways to develop a supervisory controller, is to apply controller synthesis, as described in supervisory control theory [43]. The behaviour and communication of all the different systems are modelled as plants (which are basically just automata). The events to these systems are split into two sets: controllable and uncontrollable events. Controllable events are events that can be triggered by the supervisor, whereas uncontrollable events are originated externally. Each of these plants specifies the uncontrollable events, as well as the controllable events. Constraints on the controllable events can be specified using requirements. These requirements restrict events that can happen based on conditions that should be satisfied in order for controllable events to occur.

Existing work has been done to use tools that support supervisory controller synthesis (like CIF [29], more information can be found in the chapter on related work, chapter 3), and convert the supervisor to code, and construct supervisor controller in ROS from it. Although the supervisory controller itself is generated automatically, it still requires modelling all communication individually, create a ROS node manually and keeping the bindings between the robotic middleware and the supervisory controller up-to-date. This process can be very time-consuming, repetitive and error-prone.

1.2 Approach

Within this thesis, the approach by Kok et al. [29] is taken a step further by making use of model-driven engineering. The goal is to create a domain-specific language (DSL) that allows language users to define the middleware communication, model the different components and specify requirements in order to extract this a supervisory controller from it. By using a DSL, the idea is that users spend less time and not make mistakes when creating a supervisory controller as the controller is synthesised automatically using an external tool, and the ROS-code with bindings to the controller is generated as well, which prevents mistakes. The generated code supports multiple versions of ROS. ROS is software that can be installed onto a robot, and acts as a robotic middleware. This middleware enables communication between different parts of the robot using various means of communication. Section 2.3 contains more information about this.

Although multiple robotic middlewares exist [15], and different tools can be used to create supervisory controllers, the domain-specific language as devised in the thesis introduces concepts from ROS (both versions, ROS1 and ROS2) and makes use of the concepts of supervisory control theory [43]. In this thesis, CIF (see section 2.5) will be used, but any other tool that supports supervisory controller synthesis could be used.

The focus, and therefore the concepts, for the DSL in this thesis has been on the ROS middleware, to ensure familiarity with the concepts of ROS in terms of communication.

1.3 Research Questions

For this thesis, some guiding research questions have been formulated upfront. These questions help answer sub-problems that lead to the development of a domain-specific language for a robotic supervisory controller.

RQ1. What main approaches are used for modelling robots with DSLs?

The first research question mainly focuses on generic domain-specific-languages for robots. It is used to find answers to the different ways of modelling robotic communication, software environment and generic robot concepts.

RQ2. What concepts can be identified in the supervisory control domain for ROS?

In the second research questions the focus shifts to the supervisory control domain. It focuses on the different concepts that are defined within the domain, especially the ones that are relevant to the language that is developed.

RQ3. What kind of requirements should a supervisory controller be able to satisfy?

The third question focuses on the requirements that a supervisory controller should support and to adhere to. These requirements are specified to support the supervisory controller synthesis process, and it is important to check which kind of requirements should be supported.

RQ4. Does the proposed DSL-solution work in one or more case studies?

In the last question, the focus lies on the evaluation of the developed solution. It is important that the language functions as expected, and that it is expressive enough to be useful in a wide variety of scenarios.

1.4 Structure of the Thesis

This thesis structured as follows:

- **Chapter 1** contains an overall introduction to the topic, research and the structure of the thesis.
- **Chapter 2** describes some of the core preliminaries which are required to understand the ideas and concepts that the thesis describes.

- **Chapter 3** focuses on the related work that has already been done and is (partially) used or relevant to this thesis.
- **Chapter 4** outlines a general idea of the developed language, its concepts and core ideas.
- **Chapter 5** describes all the logic around the generators for supervisory control theory (SCT) and the Robot Operating System.
- **Chapter 6** evaluates the language based on a set of scenarios, contains measurements for each of the scenarios and discusses some limitations.
- **Chapter 7** discusses the work that was done in the thesis and how it relates to the original research questions specified in the introduction.
- **Chapter 8** contains a conclusion of the thesis and recaps the work that was done to solve the original problem as written down in the introduction.

Chapter 2

Preliminaries

2.1 Introduction

This chapter covers some basic preliminaries that are required to understand the goal and research that was done within this thesis.

2.2 Model-Driven Engineering (MDE)

Model-Driven Engineering (MDE) is a software development methodology that aims to increase compatibility between systems and simplifying the design process. The model can be defined as an abstraction of the system under study, where the abstraction is often replacing the model [11]. By incorporating the domain of the software project, model-driven engineering aims to promote communications between all stakeholders in a single project [18]. The goal of using model-driven engineering is to save time by using concepts closer the domain, which increases productivity [50].

The first tools to support the MDE-methodology date as far back as the 1980s [6]. In the 90s, this resulted in the creation of the Unified Modeling Language (UML), which was adopted as a standard by the Object Management Group (OMG) in 1997 [6]. UML is very powerful for creating complex models of a software project. These models are the single source of truth in a model-driven engineering project for all of the stakeholders. Code that is written is based on these models and verification of the software happens from the developed model as well.

What can make model-driven engineering especially powerful is the application of model-to-model (M2M) or model-to-text (M2T) transformations [50]. These transformations are used to extract different artifacts from the set of developed models. For example, models can be transformed to code using automatic code generation as an model-to-text transformation. Model-to-model transformations can be used to convert models to new models that can act as an input for different software, like formal verification tools.

2.2.1 Domain-Specific Languages (DSL)

A domain-specific language is, contrary to a general-purpose language like Java, a programming language that is developed for a specific domain. It can be seen as a high level software language implementation that introduces concepts and abstractions related to the current domain [59]. It also limits the freedom of what can be done, which can prevent mistakes. A domain-specific language can be used to instantiate entities within the model of a domain.

A domain-specific language is defined using syntax and semantics. The syntax defines the structure of a language, whereas the semantics define the actual meaning of the code. DSLs can be distinguished into two categories: *external* and *internal* languages [34]. Where external DSLs define their own syntax and semantics, internal DSLs re-use part of the syntax and semantics of the host language.

This means that these languages are somewhat restricted in the expressiveness of the DSL itself, but do add the power of a domain-specific language to a general purpose language.

Multiple workbenches exist that can support the development of DSLs [17]. A popular tool is XText [1], which is a framework that supports the development of domain specific languages. It allows defining a grammar for a DSL, and get a parser, linker, typechecker and compiler as a result. Another option is JetBrains MPS [2] which takes a different approach by using a projectional editor. With a projectional editor, developers are programming in the model tree rather than in a concrete syntax (a concrete syntax is the textual representation of the syntax). This has some advantages, as it allows language designers to add complex elements in the editor. For example, the language could introduce a graphical representation of a matrix or a vector.

2.3 Robot Operating System

Middlewares in robotics were created to make it easier to create different processes and enable communication between these processes. Robots are often highly asynchronous with a lot of hardware that requires dedicated drivers. A generic middleware can be defined as a software glue, which takes care of communication between different applications within a system. Using a robotic middleware can allow robotic developers to focus on specific parts of a robot and re-use existing solutions as the parts and components are developed individually. Several middlewares exist [15], like the Robot Operating System (ROS), which is the only middleware that is within the scope of this thesis, to ensure familiarity with the concepts and communication of ROS. There are other middleware s[15], for example Miro or Player. Miro is a distributed object oriented framework using a generic broker architecture, allowing developers to build robotic applications in their own programming language [38]. The Player Project enables the creation of free software to be used in research and education [42].

The Robot Operating System (ROS) can be defined as a set of software libraries and tooling that assists in developing the software of a robot [56]. It is an open-source library that comes with an ecosystem that allows re-using pre-existing modules like state-of-the-art algorithms for navigation or drivers for certain motors and sensors. These modules are distributed using Github and can be found using the ROS website. Although the name of ROS contains operating system, it runs as a program on a host system. The current version of ROS is 2. The previous version, ROS1, defines the same concepts [54].

The base of ROS consists of the ROS graph that defines all nodes and the communication between them. A node is a process that performs computation [52] and can communicate with other nodes by using messages, services and actions. ROS provides client and server libraries that can facilitate this communication.

- **Messages:** messages have a specific data structure, and can be published to a topic using the publisher/subscriber model. Other nodes can listen to these messages by subscribing to the same topics.
- **Services:** services are based on a call-and-response model, similar to that of HTTP. Services that are provided by a node return a response when invoked from a client.
- **Actions:** the intention of actions is that they support longer running tasks. They consist of a goal, feedback and a result. A node can offer an action server, to which an action client node can connect. The action is initiated by the client, by supplying the server with a goal. The server acknowledges this and provides the client with feedback during the execution of the action. When the action is finished, the server returns a response to the client. The goal and response are similar to a service, whereas the feedback is provided using a message.

An example of the communication within ROS can be found in figure 2.1. It consists of three nodes, who communicate with each other using messages and services. Arrows represent communication between two entities. This example is based on the graph from the ROS website [52].

[1] https://www.eclipse.org/Xtext/

[2] https://www.jetbrains.com/mps/

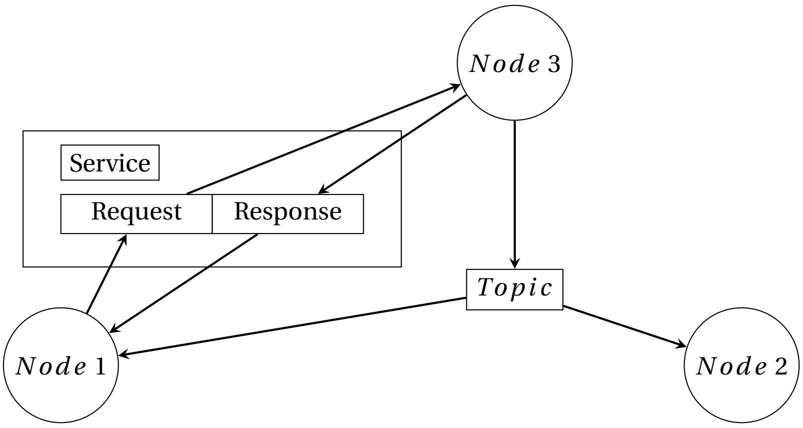


Figure 2.1: Example of a ROS graph with three nodes.

The detection and discovery of nodes within the same network happens automatically in ROS2, as long as they run in the same network. Every ROS2 setup has a domain identifier, which is a number that all nodes use to communicate with each other. As long as nodes communicate with the same domain identifier, they will be discovered. In ROS1, discovery happens using a master node. The discovery continues, even after the initial setup phase, to ensure that the presence of new nodes is periodically checked. Furthermore, when nodes go offline, they have the option to advertise their shutdown.

To launch a program in ROS, one can start a node directly, or make use of a launch file [55]. These launch files contain information about how to start one or multiple nodes, allowing users to quickly start a set of nodes that control all parts of the robot. It also supports the re-use of existing launch files, which could be used to start all nodes required on a specific robot simultaneously.

Currently, there are two main versions of ROS: ROS1 and ROS2. Software updates for both versions are released in the form of distributions. There are some differences between ROS1 and ROS2, as described in [54]. The most notable changes are:

- ROS2 supports multiple platforms (Ubuntu, MacOS and Windows) out-of-the-box
- Support for real-time nodes was added in ROS2.
- Actions are part of the main ROS project in ROS2, instead of requiring an additional library.
- Opposite to version 1 of ROS, ROS2 does not need a central ROS master. All nodes have the capability to discover other nodes without using a centralized system.
- Transport of data happens using the DDS standard, instead of a custom protocol [54].

2.4 Supervisory Control Theory

The Supervisory Control Theory, abbreviated as SCT, was proposed by Ramadge and Woham in [43]. The idea is to restrict the behaviour of a discrete system by synthesizing a supervisor. A discrete system is a system that has a countable number of states, contrary to a continuous system that has an infinite amount of states.

In supervisory control theory, Ramadge and Woham first define the concept of a generator, which is similar to what an automaton is. The generator will play the role of a plant, which is a representation of an individual system and its states and is the object to control. The formal definition of a generator can be found in [43].

States of the generator can be labeled as marked. If a state is marked, it means that the state has some special significance [62]. An example of such a significance is that they represent a finished task,

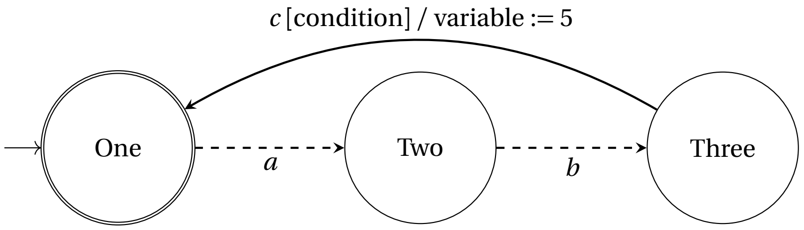


Figure 2.2: Example of a visual representation of a plant.

where a finite amount of events can occur after which the generator will return to the marked state. Events can either originate from the plant, or from the outside. If such a process returns to the initial state after a while, the initial state can be defined as a marker state. To apply controller synthesis, it should be possible to reach a marked state from all reachable states in the system [36].

The generator will control a system and act as a plant. To do so, supervisory control theory defines Σ as the set of all events, and Σ_c as the set of controllable events, or the events over which the supervisor has control. Such a controllable event can either be enabled or disabled. If it is disabled, the supervisor will never allow the event to occur, whereas if it is enabled it is not forced to occur.

Next to the controllable events, there are also events that the supervisor has no control over: the uncontrollable events. These events update the active state of a plant, and can occur at any time. The supervisor observes the string of uncontrollable events and can restrict a plant to execute controllable behaviour.

A plant can be visually represented in a diagram. This can be done in different ways. In this thesis, all of the states of such a plant are represented by circles. If a state is also an initial state, it is annotated with an incoming arrow. Double circles represent a state that is marked. Transitions between states are visualized using arrows. Solid arrows stand for controllable events, whereas dashed arrows represent uncontrollable events. In some cases, a transition (of a controllable event) can have guard, which only enables the transition if the guard holds. This is represented by an edge label which contains brackets around the guard expression. It is also possible that a transition results in assignments to variables. This is represented as a suffix to the edge label in the form of */ <variable> := <value>*. An example of an arbitrary plant is shown in figure 2.2.

Lastly, there is the concept of requirements. A requirement can require a plant to be in one or more specific states in order to enable a controllable event. These requirements are used to synchronize a supervisory controller that adheres to these requirements and will never allow controllable events to occur that are disallowed by one or more of the requirements. Thus, the requirements specify the behaviour that is allowed in for the supervisory controller. The requirements can be used to synthesize a supervisory controller which: prevents blocking, only disables controllable events and which also does not restrict the system any more than is strictly needed.

2.5 CIF

CIF stands for the Compositional Interchange Format and was developed at the University of Eindhoven [35]. The goal of CIF is to allow the modeling of discrete event systems, timed systems and hybrid systems using automata. It contains a large set of functionalities that can be used in the development of controllers, including supervisory controllers. CIF has support for the following features for the development of controllers [35]:

- Specification
- Verification
- Supervisory controller synthesis
- Simulation-based validation

- Simulation-based visualization
- Real-time testing
- Code generation

In this thesis, CIF will only be used to perform supervisory controller synthesis and generate code from this supervisory controller.

Chapter 3

Related work

3.1 Introduction

This chapter contains related work for all that is done in the thesis. It contains literature on existing robotic domain specific languages and their approaches, how they are evaluated, their development process and their ecosystem. Furthermore, the chapter highlights work done in supervisory controller synthesis in robotics, especially in combination with ROS.

3.2 Robotic Domain-Specific Languages

This section contains the state-of-the-art work that was already done with respect to domain-specific languages for robots. Section 3.2.1 highlights the different approaches taken by the languages, of which some ideas, concepts or insights can be introduced into the language developed in this thesis. It also discusses the different approaches taken with respect to evaluation of the developed languages in section 3.2.3. Furthermore, the different artifacts and the ecosystems around the developed solutions are discussed in section 3.2.3 and 3.2.4 respectively. The insights from these sections are used throughout the thesis.

3.2.1 Approaches

There are a lot of languages that apply model-driven engineering in robotics with domain-specific languages [33]. One of the approaches that can be taken, is the task-based approach as done by Heinzemann and Lange [23]. They developed the vTSL-language: a formally verifiable (using an external model checker) DSL for specifying robotic tasks. This is a text-based language that defines the behavior of the robot using specifications of different tasks. These tasks are then verified using a separate model checker, Spin [1].

Tasks are specified using so-called actions. Actions define the behaviour of a robot and can run concurrently with other actions based on an optional set of parameters as an input. Furthermore, actions have the option to call and start different actions. An interesting point, is that each action is defined as a set of behaviors that each specify under which conditions the behaviour should be active.

The definition of the behaviour itself uses a C-like syntax. A key part of the DSL are sub-actions that allow the re-use of logic between different behaviour definitions. These calls to sub-actions can be executed in parallel if required.

vTSL also has built-in support for communicating with a middleware like ROS. Using keywords like *connect, read, write* and *disconnect,* the DSL adds support for subscribing, listening and publishing to topics in the middleware. The primary focus of the paper is to support the ROS middleware. Support

[1] http://spinroot.com/spin/whatispin.html

for synchronuous request and reply communication is built using the *query* keyword. In ROS, communications adhere to a data type. vTSL allows the definition of custom data structures from within the language.

The language is implemented using Jetbrains MPS [2], a language workbench that supports the creation of domain-specific languages. One of the unique features of MPS, is its projectional editor. The projectional editor supports editing files using non-textual syntax. As an example, MPS allows writing mathematical representations of vectors from within the code. This is done by saving the entire file as an abstract syntax tree, instead of a plain textual notation. The projectional editor only supports typing commands that adhere to the grammar of the language, resulting in no more syntax errors.

Actions and behaviour of the language are transformed to code using a model-to-text (M2T) transformation. However, the paper does not describe the methodology for this.

One of the use cases of the constructed model of the robotic behaviour, is performing a model-to-model (M2M) translation to a model-checker called Spin. The model is converted to a state machine, which is done using component stubs. As the language has no concept of the possible states and interactions of a middleware node, these need to be modelled as well. Heinzemann et al. therefore introduce the skill layer that models the topics and services of the middleware. These definitions do not necessarily correspond to a single ROS-node, but may describe the behaviour of a group of nodes.

The authors opted to let user specify assertions on the value of variables from within the domain-specific language, instead of using a separate tool for this. This yields some advantages, as variables can easily be mapped to states in the resulting model. However, the assertions that can be performed are still very limited. Another weakness of vTSL is that it is still rather vague and undefined how the logic of a program should be split up into different actions, behaviours or functions. Also, communicating with the middleware results in the use of ambiguous keywords. Lastly, the implementation in the paper still misses the approach of code generation.

An example of a program that is written in the vTSL-language is shown in listing 3.1. Note, that the skill layer and ROS-bridge data types have been omitted.

```
action MoveUntilObstacle(double speed)
    behavior normal
        connect DistanceSensor.distanceTopic as distanceSensor with queue size 1;
        DistanceMessage curMsg = read distanceSensor;

        while (curMsg.distance < 100)
            curMsg = read distanceSensor;
            setSpeed(speed)
        end on abort
            disconnect distanceSensor;
            setSpeed(0);
        end

        disconnect distanceSensor;
        setSpeed(0);

        return;
    end

    function setSpeed(double speed)
        Float64 speedMsg;
        speedMsg.data = speed;

        write speedMsg to Motor.speedTopic;
    end
end
```

Listing 3.1: Sample of a robot that moves until it reaches an obstacle in vTSL.

Another text-based approach was proposed by Ringert et al. in [46], namely MontiArcAutomaton [3]. MontiArcAutomaton is a framework that adheres to the component and connector (C & C) architec-

[2] https://www.jetbrains.com/mps/

[3] https://monticore.github.io/monticore/docs/DevelopedLanguages/

ture. In this architecture, the composition of all components is separated from the individual behaviour of a single component. Communications between these components are defined as ports. The behaviour of a component can be defined as an automaton or using a problem-specific language. Support for these extensions is developed in MontiArcAutomaton ADL.

The MontiArcAUtomaton language is based on the MontiCore Language workbench [4]. The language is defined as a context-free grammar, which MontiCore uses to generate a parser that can parse a concrete syntax into an abstract syntax tree (AST). Although MontiCore languages are text-based, an integration for the Eclipse Modeling Framework (EMF) is provided to support visual development of models.

MontiCore supports language composition based on three principles:

- **Embedding**: use parts of existing languages at pre-defined extension points.
- **Aggregation**: combine multiple, independent, languages into a single language family.
- **Extension**: re-use and extend existing grammars.

The paper shows the power of composition by embedding a domain-specific language for robot arms to allow modeling the behavior of a component that controls a robot arm. Communications take place by defining one or multiple ports.

Furthermore, the MontiCore workbench also has functionality to support in the development of code generators. These code generators are defined in the configuration of an application and can generate implementations for different general-purpose languages.

The MontiCoreAutomaton language is very well defined and is built on a solid base. The compositional nature of the language makes it very-well suited for a wide range of applications. However, the source of the language is only partially available. The source of MontiCoreAutomaton is available on the internet, but the source of the extensible MontiCoreAutomaton ADL is not. Furthermore, the starting point when developing a robotic application with MontiCoreAutomaton is not entirely clear, which makes the threshold to develop with the DSL relatively high.

The listing 3.2 contains an example of the controller of a robot that moves until it finds an obstacle.

```
component Controller {
    port
        in int distance,
        out int speed;

    automaton {
        state Moving, Halt;

        Moving -> Halt {distance < 100} / { speed = 0 },
        Moving -> Moving / { speed = 100 };
    }
}
```

Listing 3.2: Sample of a robot that moves until it reaches an obstacle in MontiCoreAutomaton.

A graphical-based approach was taken by Dhouib et al. in [13]. This approach was first published in 2012, but it still has some interesting approaches to consider. The approach is based on the Eclipse Modeling Framework [5] (EMF). EMF is an extension to Eclipse that facilitates in developing models and code generation for different sets of tools and applications based on a structured set of models. It also uses Papyrus [6] to support graphical editing of models.

The language was developed based on a set of requirements. These requirements included that the language had to be easy to use, it should support component-based architectures and that it should support multiple target platforms using code generation. Dhouib et al. do not explicitely evaluate the ease of use, but only reason about this. They state that the language does not require users to have programming knowledge, and that platform details are hidden in the models.

[4] https://monticore.github.io/monticore/

[5] https://www.eclipse.org/modeling/emf/

[6] https://www.eclipse.org/papyrus/

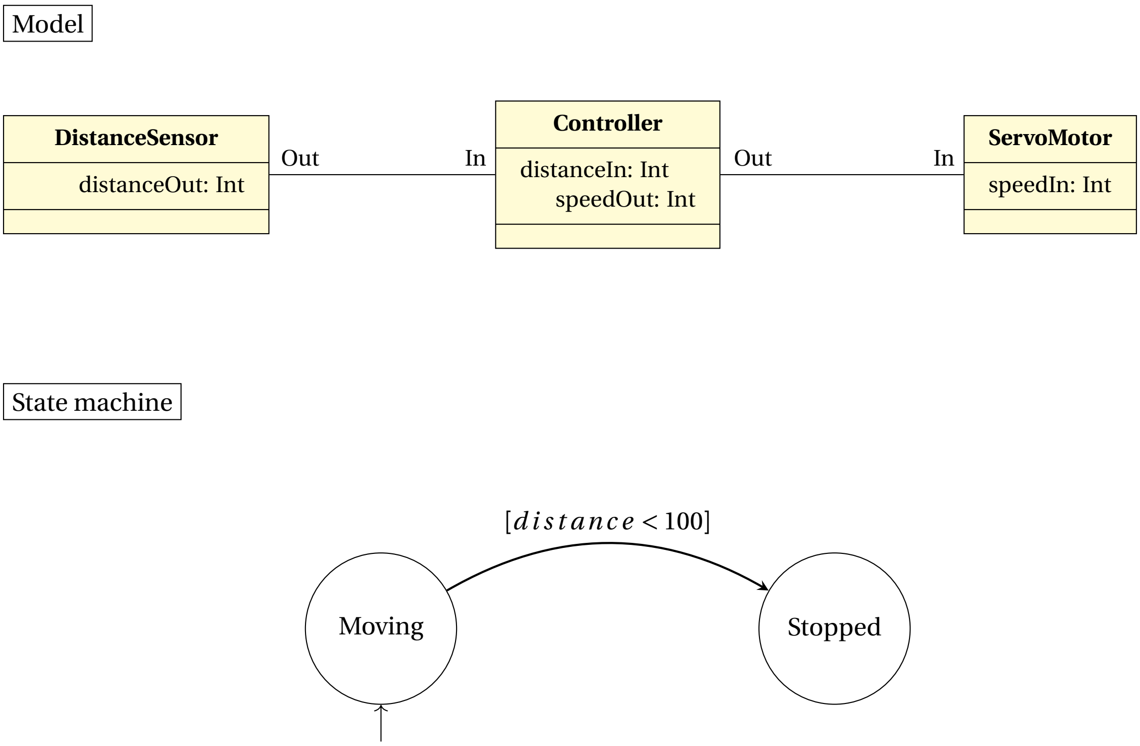


Figure 3.1: Sketched sample of a robot that moves until it reaches an obstacle in RobotML.

RobotML starts by designing a model that represents sensors, actuators and control systems of different robots. Communications are defined as ports and connectors. Both the publish and subscribe architecture, as well as the request and reply mechansims are supported through these ports. The behaviour of control systems is modelled in the form of state machines that represent the different states of each system and use the incoming ports and outgoing connectors. The next step for the development of a program consists of defining a deployment plan that specifies the robotic middleware being used.

The RobotML model was based on the ontology of a robot. The domain of robotics was studied in order to come to the design of the language. Concepts that were identified include a *Robot*, *SensorSystem*, *ActuatorSystem* and a *LocalizationSystem*.

The RobotML paper was published back in 2012, which may be the reason of issues that arise when attempting to install RobotML in newer versions of Eclipse and Papyrus. The last activity on the RobotML GitHub organisation page dates back to 2016 [40], which might indicate a lack of maintenance of the framework. Also, the language has options to deploy to one or more multiple (robotic) middlewares, but still requires the bindings to the platform to be defined manually.

As stated above, RobotML has been outdated and is not compatible anymore with newer versions of Eclipse and Papyrus. A sketch of a solution in RobotML is depicted in figure 3.1.

Another approach, developed by Elliott et al. in [16], is to use synthesis to extract a controller. This uses a *correct-by-design* approach contrary to the manual approach of constructing a robot controller that correctly (based on the specifications) reacts to changes in the dynamic environment because the controller is extracted using synthesis. The manual process can be very time consuming and error prone. The language uses Generalized Reactivity(1) (abbreviated as GR(1)) specifications to automatically generate controller designs from the specification. The paper proposes Salty: a domain-specific language that allows the definition of GR(1) specifications.

These GR(1) systems work as follows. A specification ϕ has the form of $\phi = \phi^e \implies \phi^s$, where ϕ^e contains assumptions about the environment, and ϕ^s encodes guarantees that the system should make under these environmental conditions. The language lets users create those specifications, and uses external tools to apply the actual synthesis of these GR(1) specifications. The DSL adds support for complex or repetitive expressions using common patterns and macros. The synthesized controller (which is a mathematical representation of the controller) is converted to multiple general-purpose languages like Python, Java, and C++.

At a top level, Salty defines the concept of a controller which takes inputs and outputs that are defined on the controller. These types of inputs and outputs can be enums, which are represented as bit vectors. Specification properties can be added in their corresponding block. System initialization and environment initialization have their own block, the same holds for transition properties in the system and environment as well as liveness properties for both the system and the environment. Furthermore, Salty adds support for features like sanity checking (approximating whether the specifications are realisable), debugging and optimization.

3.2.2 Evaluation

The evaluation of a language is key when deciding whether or not a language is suitable for a given problem, not only from the side of the language user, but also from the side of a language developer. Nordmann et al. show that evaluation either takes place using quantitative or qualitative evaluation [33]. Qualitative evaluations focus on the performance of a language in terms of execution time or build time, whereas the latter is more focused on how well concepts can be modelled with the DSL approach.

Some papers choose to evaluate the capabilities of their language by letting them perform a dedicated task, and execute it on different physical platforms [51] [61] [27] [25] [49] [44]. During the case study, different types of robots were chosen. In some cases, these robots had different sensors and actuators to highlight the handling of robots with different capabilities. There are also papers that rely solely on the evaluation in a simulator [45], which has some advantages in terms of reproduction, but it would still have been better to see a real-world example. In other cases, no evaluation has taken place at all, but was rather mentioned as a point of improvement [16].

A different approach of evaluating the language was taken by Sutherland et al. [51]. In order to evaluate the simplicity of the language, Sutherland et al. demonstrated the language using a live demonstration to an audience, and applied user-suggested changes in a short time frame.

Next to the qualitative approaches to evaluation, there are also quantitative ways. Examples of such metrics are the performance-per-watt metric [49] or the time it takes multiple groups to solve a given task using the language [27]. There are also papers that evaluate the portability of a language. There are two definitions for portability. First, there is portability in terms of the possibility for the language to run on different robotic platforms [51]. Then, there is also portability in terms of the amount of languages that the model can be transformed into [16].

3.2.3 Artifacts

In the context of model-driven engineering, there are different kinds of artifacts that are generated from a model. The most common approach, at least within robotics, is code generation from a model [33]. The model is fed to a generator, which generates one or more files with code. The use of multiple code generators highlights the power of model-driven engineering even more, as artifacts for multiple languages or tools can be generated from only a single model source [47].

As stated before, the most common scenario for using the model, is code generation. This code can be code for languages like Python [45] or Java [10] [60], or even a specific framework that handles a dedicated task like scheduling [61]. There is also a language, Salty (by Elliott et al. [16]), that produces a controller for different languages (Python, C++ and Java) as an artifact.

There are also opposite approaches, where the model is an artifact rather than a source. By combining information that is gathered from applying statical code analysis together with runtime monitoring of a ROS node, a model is constructed [19]. In this case, model-driven engineering for robotics is used as a complement, rather than having it as a separate solution.

Some of the papers describe approaches that allow feeding the model into a model verification tool. Desai et al. [12] describe how state machines for the model are generated and fed into a state-of-the-art prioritization tool, Dona. Some approaches allow modelling tasks as a task tree, after which they are transformed using a model-to-model transformation into models that can be verified using tools like Promela [23].

3.2.4 Ecosystem

Although there are lots of languages proposed within the domain of robotics [33], some of them only provide a definition of the language in the paper, leaving potential language users with little information about how to start using the language. It is interesting to consider what the ecosystem around existing languages looks like in terms of the community around a DSL, the availability of documentation and instructions for developers to get started easily.

There are papers that provide users with the entire grammar of the language [27] [51] [1] [24]. Providing this grammar gives users a quick insight into all the possibilities of the language, but still requires technical knowledge and a more in-depth read of all structural concepts that the language defines.

Some of the papers illustrate the workings of languages by going through one or more example scenarios that illustrate the concepts that are defined within the model of a language [23] [16] [26]. Providing these examples can increase the adoption of such a language, as it provides good insights in how specific problems should be handled, and shows the expressiveness of a language. Some papers choose to demonstrate small examples [23], whereas others highlight more complex scenarios [16].

Documentation of the language and its concepts alone is not sufficient in most cases, as users often require (installation) instructions to help them get started. There are papers that refer users to external sites like GitHub for providing these instructions [16] [61]. Some store all of this information (workbench, tooling, robotic models, installation, architectures and concepts) in a separate website that is publicly accessible [61]. Contrary to this, there are also languages that provide little to no information about how to get started, making it difficult to start using them [60] [49].

3.3 Supervisory Controllers

The main idea that the work in this thesis is based on, was done by Kok et al. in [29], where they apply a synthesis-based engineering approach to create a discrete event controller for a mobile robot platform. They describe the process of applying synthesis to discrete event controllers for complex navigation tasks using CIF based on supervisory control theory. The synthesized controller is deployed using a ROS1 node, which is written in Python.

In the paper, they use state-of-the-art navigation modules which are provided by a ROS package. These interfaces are modelled within CIF as a plant and contain navigation modules, a laser scan distance sensor and a human machine interface. The invocation of the navigation modules presented by the ROS package happens in the form of actions. Each of these actions was modelled as a separate plant that contains two states, where it is idle or active. Controllable goal events take the plant to the active state, whereas the uncontrollable finish event and the controllable cancel event take it back to the idle state. Note that action feedback is omitted in this case.

Next to plants for each of the navigation actions, Kok et al. also define observers which update their state based on outputs from the navigation modules, laser scan events and human machine interface. Then, the paper defines requirements which are used in the controller synthesis process. These requirements specify conditions that either a controllable event needs in order to be enabled, or a condition that disables such a controllable event. The conditions are all defined as boolean expressions that need plants to be in a specific state. They are specified using both conjunctions, as well as disjunctions.

A controller is then synthesized from the set of specified requirements. This controller is synthesized using CIF, after which the code generator of CIF is invoked to generate C. Because CIF does not support Python code generation, a Python wrapper was built that allows Python 2.x modules to interact with the controller based on the generated C-code. The working of this code generation process was described in [28].

CIF3 has been used for supervisory controllers outside of robotics as well, for example in product line engineering [3]. They provide an example model of the plants of a coffee machine and provide the requirements that are used in synthesis, after which a supervisory controller is synthesized.

More work has been done to apply a supervisory controller in robotics [48], where they use a hybrid approach and model multiple robots with a discrete event model, and the control of a robot as a continuous time based model. Synthesis is applied on the discrete event model, and a supervisor is extracted from it.

Gleirscher et al. highlight the sound development of safe supervisors [22], where they combine the best of both worlds. They both use verified synthesis, as well as complete testing. They propose a workflow that bridges the gap between the synthesis process, the derivation of the test suites from the supervisor reference, the generation of code that is executed, the test and deployment control system and the integration into the wider system.

In [30] Kress-Gazit et al. review the state of formal synthesis within the field of robotics. As they mention, applying synthesis allows reasoning about the task specification, rather than its implementation. Manual implementation requires skilled programmers and extensive testing to verify the implementation of a task. Synthesis provides users with guarantees about the behaviour of the robot. Kress-Gazit et al. state that synthesis has a high potential, but was not yet widely used.

Gleirscher and Pelaska present work on testing the implementation of a synthesized supervisor in [21]. Although it is possible to verify the properties of the supervisor itself, according to Geirscher and Pelaska it is possibly not feasible to verify the actual generated code of the supervisor. They present work to generate an abstract test reference instead of the more complex semantics of the generated code. This test reference is generated in the form of a symbolic finite state machine, and is used to show the equivalence between the test reference and the generated concrete controller that is running on the control system platform.

Chapter 4

Language

4.1 Introduction

In this chapter, the *RoboSC (RSC)* language will be presented. It will allow the development of a supervisory controller that can be installed into a robotic middleware. The core idea of the language is based on work performed by Kok et al. [29], where they apply synthesis-based engineering to robotic supervisory controller, specifically in the field of automatic navigation. They develop supervisory control theory plants for each of the actions, and specify requirements to control the execution of them. After applying controller synthesis, the C code engine is extracted. Manually, this is integrated into a ROS-node, and bindings between the controller and the deployment platform are manually created.

In the language developed in this thesis, the main goal is to promote the adoption of supervisory controller synthesis in the domain of robotics, by providing a language that allows synthesizing supervisory controllers using concepts of supervisory control theory and the robotic middleware ROS. By providing code generation, users do not need to bother about bindings and deep understandings of C++.

In this thesis, the approach of Kok et al. is extended by:

- Reducing the amount of work required to construct models and update the links between the supervisory controllers and the robotic platform

- Decrease development time by automatically applying supervisor synthesis and C++ middleware code

- Extend approach to be useful beyond automatic navigation

- Allow for easier debugging of supervisory controllers

Within the DSL, the language user can specify all (virtual) components, and all communication that can take place. Similarly to the the terminology that supervisory control systems adhere to [62], a distinction is made between controllable and uncontrollable communication. Controllable communication is the data that is going from the supervisory controller to the robotic middleware, whereas uncontrollable communication is the data coming from the middleware. Based on the uncontrollable communication, language users model the state of the robot using automata. Then, they can specify requirements to restrict the execution of controllable communication. The data that is passed along is determined based on conditions specified by the user. Then, from the DSL, code is generated to a tool that can apply supervisory controller synthesis, in this case CIF. The code of this supervisory controller is glued to a node for a robotic middleware, where the bindings and code are automatically generated, hence not requiring any additional modifications.

The controller is created as a node, and is deployed into the robotic middleware. The node contains the synthesized supervisory controller and plants, and starts a timer that acts as a control loop

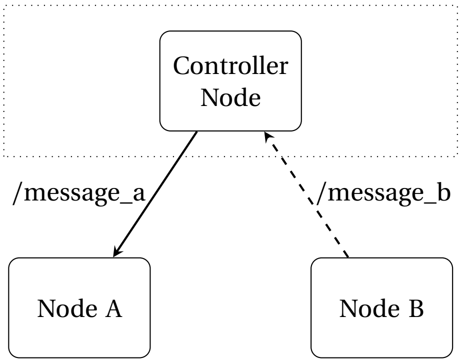


Figure 4.1: Example communication of the supervisory controller node. The dashed rectangle represents what is generated from the DSL.

and attempts to execute an enabled controllable event. The set of enabled controllable events depends on the uncontrollable events originating from the middleware. When the supervisory controller allows communication to take place, the node will publish messages, send requests to services or goals to actions. Every time new data is coming in from the middleware, the state of the supervisory controller is updated which updates the internal state of all plants. An example of the communication with the controller is shown in figure 4.1.

4.2 Concepts

This section contains all the concepts of the language and how they should be used. Throughout this section, a running example will be considered to make the concepts that are explained more concrete. The ontology of this robot consists of two motors that allow the robot to move in the two directions. Furthermore, it contains a sensor that can find a line, and if it has found one it can measure the offset of that line to the center of the robot. The robot is also equipped with a distance sensor which measures the distance to an object in front of the robot. If there is an obstacle, the robot can use its grabber to pick up the obstacle. Lastly, a light bulb is attached to the robot which can be used to represent the internal state of the robot.

The goal of the robot is to follow a line. If it finds something that is in front of the robot, it should stop moving and pick the obstacle up using the grabber. In the case that there is no line at all, the light bulb on the robot should be activated.

The full DSL model of this example can be found in appendix A.

4.2.1 Base

When a language user creates a model in the DSL, it can either start with a robot concept, or a library concept. The concept of a robot allows the definition of all components, their behaviour, their inputs and outputs, datatypes and the requirements that the controller should adhere to, whereas a library can only be used to define all aspects of one or multiple components. These can be re-used within the definition of a robot to accelerate development of different controllers for similar platforms.

A part of the metamodel of the base was added in figure 4.2. It contains the concepts of robot and library definitions, which allow the user to specify entities related to the library or robot. Libraries can only define interfaces, components and data types. Robot entities can also define provide statements and requirements.

For the example that was mentioned in the beginning of this section, the robot concept should be used. It receives a name, which is used to identify the robot. The code, as listed in listing 4.1, still looks pretty empty, but will take shape in the next sections.

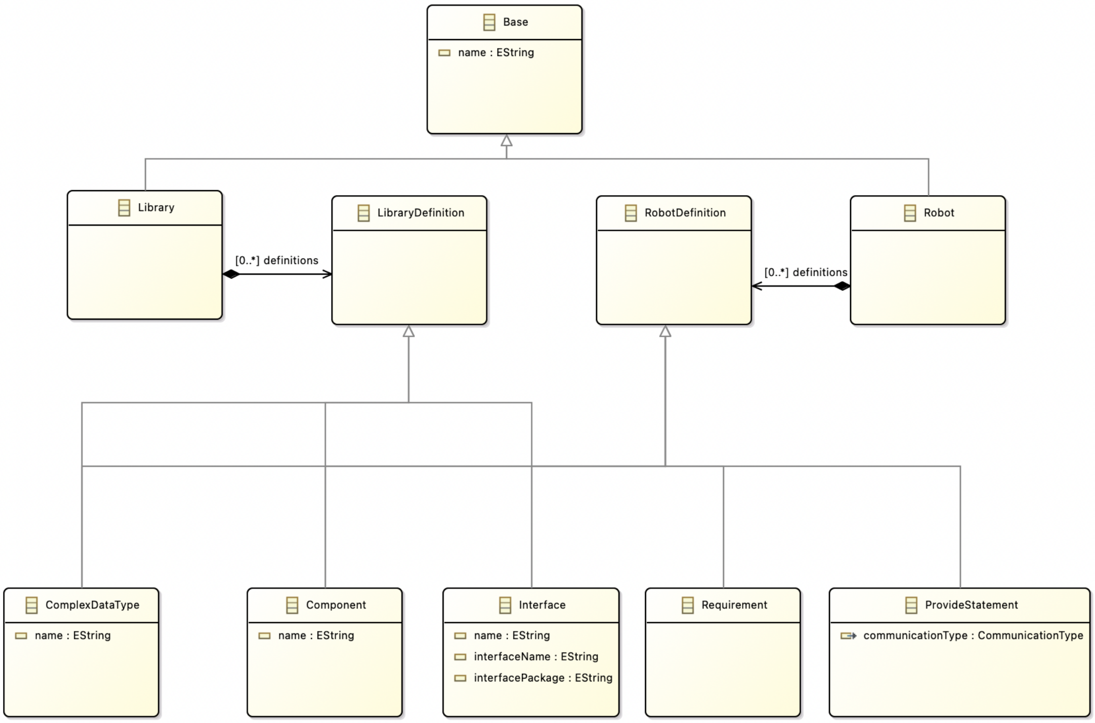


Figure 4.2: Part of the metamodel for the base concepts of the robot.

```
robot ThesisLineFollower {
    // Further robot code
}
```

Listing 4.1: Robot concept code for running example.

4.2.2 Components

A component can be defined as a virtual, software abstraction of a part of a robot. Although there is no explicit one-to-one mapping to a middleware node, this will often be the case as, at least for some robotic middlewares, it is expected that "each node in should be responsible for a single, module purpose" [58].

For each component the communication with the middleware can be specified. The types of communications that can be specified, are defined in section 4.2.6. Using this information, the internal state of the component can be modelled. This is referred to as the behaviour of a component, and can be specified using an automaton (see section 4.2.7). An example of this behaviour can be found in figure 4.3, where the internal state of a (simple) component is represented by two uncontrollable events: start and stop. These events are controllable events.

It is also possible to reference a component from a library by using an import. This is referred to as an imported component, and does not allow the specification of any additional inputs, outputs or behaviour. A component that is defined within a file itself is called a local component.

Figure 4.4 contains the concepts that are relevant to components. It highlights the relation to library components (which are imported), and to the communication types and behaviour that a component can define via a component definition.

For the example robot that should follow a line, five different components can distinguished. Each of the actuators and sensors is modelled as an individual component. For the actuators, this means

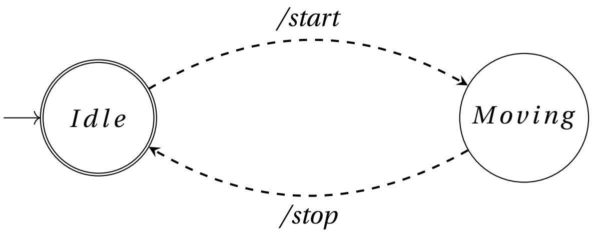


Figure 4.3: An example of the representation of the internal component state of a motor.

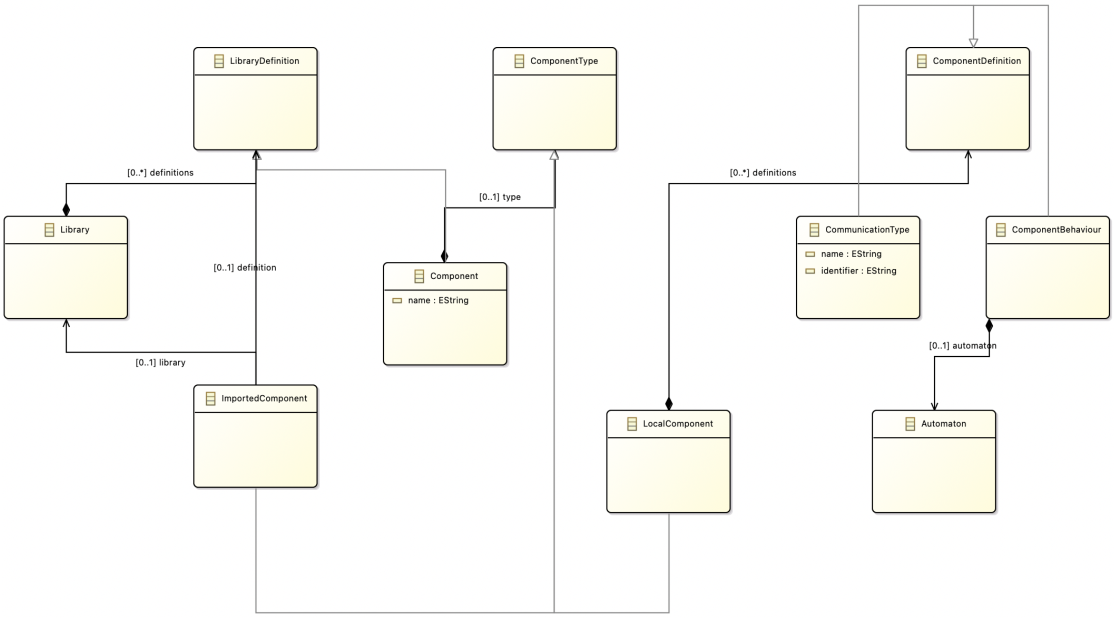


Figure 4.4: Part of the metamodel for the components concepts of the robot.

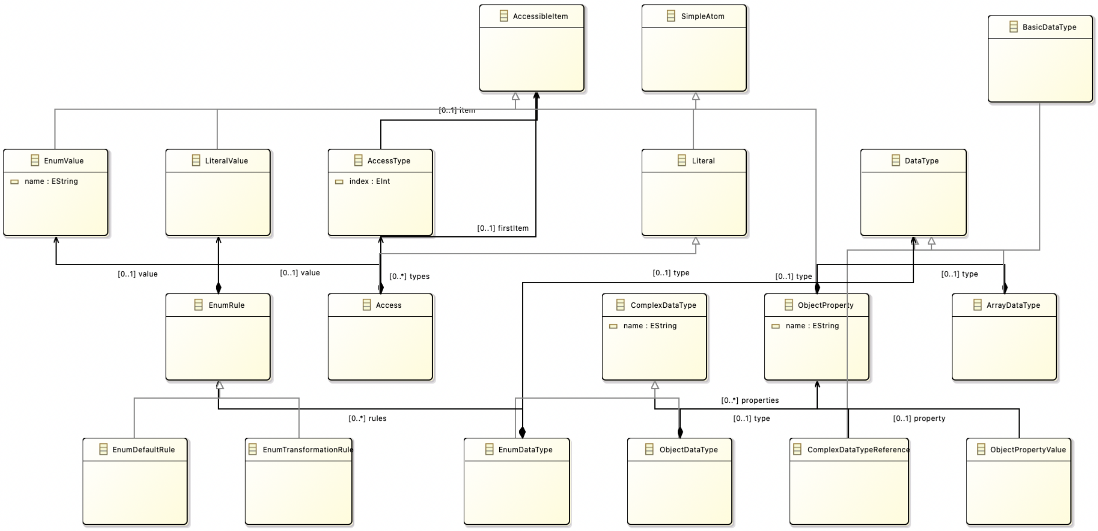


Figure 4.5: Part of the metamodel for the data types concepts of the robot.

that there are components for the light bulb, obstacle grabber and the motor. The sensors are represented by two components. One for the line detector and one for the distance sensor. This code for this model was added in listing 4.2.

```
robot ThesisLineFollower {
    component DistanceSensor {}
    component ObstacleGrabber {}
    component LightBulb {}
    component LineDetector {}
    component Motor {}
}
```

Listing 4.2: Components concept code for running example.

4.2.3 Data types

The language distinguishes three different types of data:

- Basic data types
- Complex data types
- Arrays

First, basic data types consist of *strings*, *booleans*, *integers*, *doubles* and the *none*-type. Additional constraints as to where they are used in the language may apply. The complex data types consist of objects and enums. Objects define key-value pairs, each with an associated data type. They may be used recursively. Enums are used to convert input data, based on a set of conditions, to a finite set of values. This helps to prevent exponential growth of the amount of possible states when using integers in controller synthesis, for example. An enum is defined as a set of transformation rules. Then, arrays are defined as a list of another data type. Restrictions apply as to where they are used, they can only be used to model the format of data coming from the middleware, not as a datatype for a variable. More information about this can be found in the section on validation 4.5. The concepts described are shown in figure 4.5.

For the example of the line follower robot mentioned in the introduction, four custom data types are needed. The first object datatype, *Twist*, represents the structure of the data that is sent to the motor. It stores a linear and angular velocity, which are both represented as a vector. This vector is modelled as a separate object datatype, which is referenced from the *Twist* data type. The third object datatype represents the request to the light bulb which will update its state. Next to the three object data types, there also is an enum type that converts the value that is coming from the distance sensor to a finite set using transformation rules. The value of the distance sensor is a double. If it is higher than a certain threshold, the distance is defined as *free*. If not, and by default, the value is *obstructed*. These data types will be used in the section on communication. The syntax for all these concepts is written down in listing 4.3.

```
robot ThesisLineFollower {
    // ...

    datatype object Twist {
        angular: Vector3
        linear: Vector3
    }

    datatype object Vector3 {
        x: double
        y: double
        z: double
    }

    datatype object LightBulbRequest {
        ^state: boolean
    }

    datatype enum DistanceEnum from double to {
        value >= 10 -> free
        default -> obstructed
    }

    // ...
}
```

Listing 4.3: Data concept code for running example.

4.2.4 Expressions

The language supports a basic set of expressions that are used in different parts of the language, for example in passing data to the communication items, assigning values or specifying requirements. The following types of expressions are supported:

- Conjunction
- Disjunction
- Equation
- Implication
- Addition
- Subtraction
- Multiplication
- Greater than
- Greater or equal than
- Less than
- Less or equal then
- Negation
- Assignment

To allow users to reference variables, states, enum value names or object properties in an expression, the language defines the concept of an access model. This model lets users reference the value of such an item. It is possible that the access is nested, for example if users want to reference the property of an object, which is also a property from a parent object.

4.2.5 Interfaces

The language defines the concept of an interface, which describes the software packages that contain the format specification of data coming from the middleware. Each message, service and action has a corresponding data type. Although some basic data types are supported out-of-the-box, more complex ones require external software packages. The content of these interfaces can be described via custom data objects (see section 4.2.3). Interfaces can contain one or more data types. For example, an action interface contains three data objects: the goal, feedback and response. The language user can define interfaces for a robot that contain the name of the package, and the name of the interface. Then, each communication item that uses a complex data type can reference this interface.

The example robot uses a data type *Twist* from an external package, *geoemtry_msgs*. This is declared as a single interface which references the mentioned packages, as outlined in listing 4.4. There are also two communication items that need data formats from an external package.

```
robot ThesisLineFollower {
    // ...

    interface twist use Twist from geometry_msgs
    interface light use LightBulbService from robot_common
    interface grabber use GrabberAction from robot_common

    // ...
}
```

Listing 4.4: Interface concept code for running example.

4.2.6 Communication

The inputs and outputs of each component are defined as communication types. These communication types correspond to the the types of the supported middleware [53]:

- Messages
- Services
- Actions

The corresponding parts of the metamodel have been attached in figure 4.9.

Messages

Messages are published using an anonymous publisher/subscriber model in the middleware. A message can either be a message that is going into a component (so the component subscribes to), or a message that is going out of a component (and the component publishes to). A message has a name and optionally a topic can be specified. If no explicit topic is specified, the name of the message will be used as the topic. Furthermore, a message has a corresponding data type that contains the format of the message. An illustration of the communication process is depicted in figure 4.6.

In the example that was set in the beginning, there are three components that define messages. First, there is the distance sensor component, which provides the supervisory controller with a message that contains the distance to an object. It is an outgoing message from the component to the controller, and has an enum data type which was already defined above. The topic name is defined explicitly. Then, the line detector component also defines outgoing messages. These messages represent the correction value and the event that there is no line. The correction value has a type *double*, whereas the no line message has no attached datatype and is therefore assigned the *none* type. Lastly, there is the motor component which receives incoming messages that control the movement of the robot. These messages are defined separately, but they map to the same middleware identifier. They differ in the data that will be sent along with the message. The move message will move the robot forward, whereas the stop message requests the robot to halt its movement. The data type of both

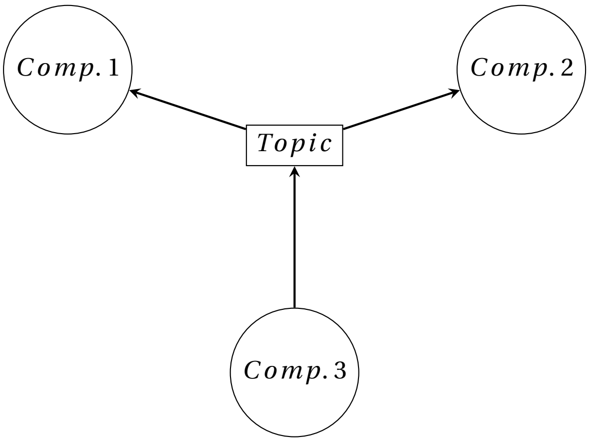


Figure 4.6: Sample message communication between multiple components.

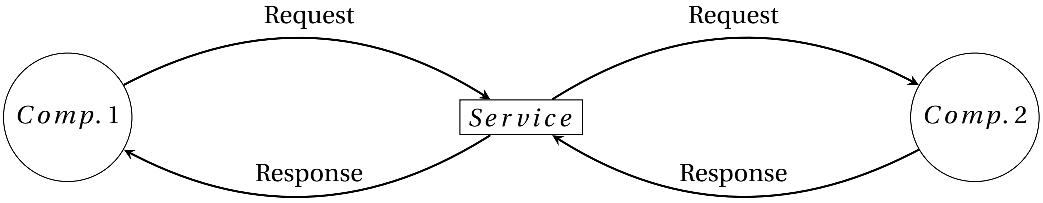


Figure 4.7: Sample service communication between multiple components.

commands refers to the complex data type *Twist* that was defined above. Furthermore, the message should be linked to the interface of that data type. The code is attached in listing 4.5.

```
robot ThesisLineFollower {
    component DistanceSensor {
        outgoing message distance with identifier: "/distance", type: DistanceEnum
    }

    // ...

    component LineDetector {
        outgoing message correction with identifier: "/correction", type: double
        outgoing message no_line with identifier: "/no_line", type: none
    }

    component Motor {
        incoming message move with identifier: "/vel", type: Twist links twist
        incoming message stop with identifier: "/vel", type: Twist links twist
    }
}
```

Listing 4.5: Messages concept code for running example.

Services

Another type of communication, are services. The software architecture of a service follows the call-and-response architecture, similar to that of the HTTP client and server model. A schematic overview has been given in figure 4.7.

Within the example, there is only a single component that offers a service, which is the light bulb. It offers a service which can be provided with a *boolean* value that represents the state of the light bulb: on or off. If *true* is sent to the light bulb, it will be turned on, and it will be turned off if *false* was sent.

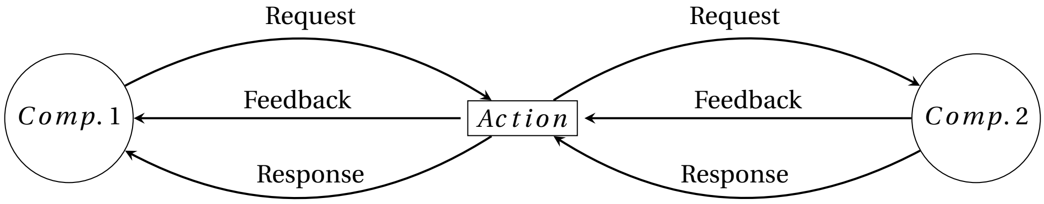


Figure 4.8: Sample action communication between multiple components.

The data type of this request was already modelled in the section on data. The corresponding service interface, that contains the data type, is also linked. The code for such a service can be found in listing 4.6.

```
robot ThesisLineFollower {
    // ...

    component LightBulb {
        service set_light_state with request: LightBulbRequest, response: none links light
    }

    // ...
}
```

Listing 4.6: Service concept code for running example.

Actions

The last type of communication that can be defined on a robot, are actions. These actions are intended for tasks that happen for a longer time [57]. Furthermore, during the period that the action is running, feedback about the process is published to a topic that can be consumed by the initiating node. A schematic overview is given in figure 4.8.

The example robot has an action which can be started to pick up the obstacle that is in front of the robot. It does not need require any data, therefore the data types of the request, response and feedback of the action are of type none. It is required that the action is linked to the interface that was already defined, which represents this empty action format. The definition of such an action on a component can be found in listing 4.7.

```
robot ThesisLineFollower {
    // ...

    component ObstacleGrabber {
        action grab with identifier: "/grab", request: none, response: none, feedback: none
            links grabber
    }

    // ...
}
```

Listing 4.7: Action concept code for running example.

4.2.7 Automata

Within the language, components can define their behaviour in form of an automaton. An automaton has one or more states, and should have a single initial state. A state can also be labelled as marked [43]. It should be possible to reach a marked state from all of the other reachable states [36]. Each state can define a set of transitions. This can either be result transitions, or tau transitions. Result transitions occur when communication with the middleware happens. Events that trigger these transitions are

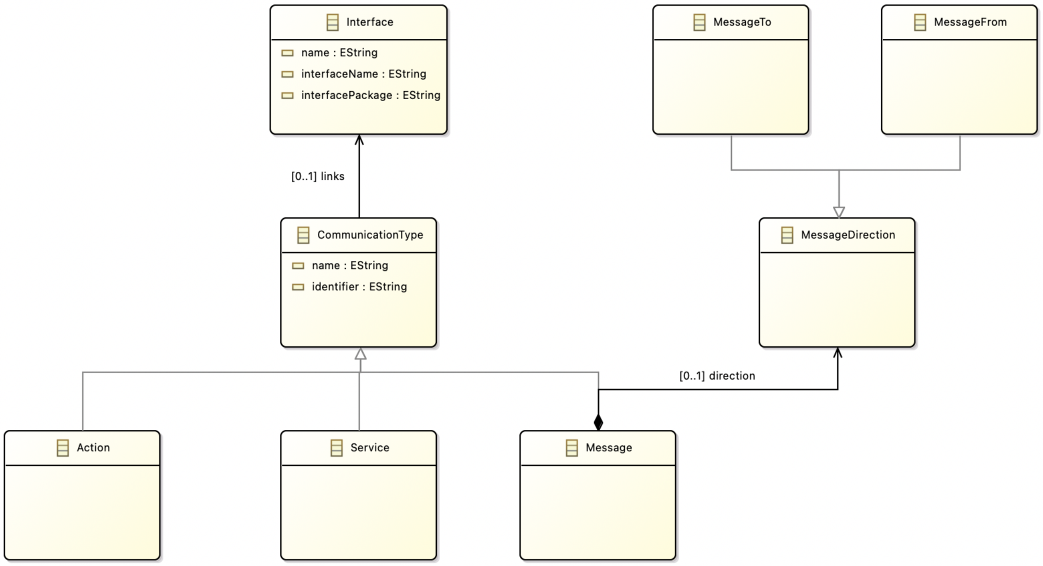


Figure 4.9: Part of the metamodel for the communication concepts of the robot.

requests, responses, feedback and errors. Tau transitions can be executed as soon as the guard is enabled. An example of such an automaton, with incoming messages and tau transitions, is depicted in figure 4.10. Note that the **value**-keyword in the assignments represents the data that is coming to the middleware.

In some cases, a user might want to have transitions that are present in every state within the automaton. In this case, the option is offered to define them on the automaton level, rather than on a single state. Then, the transition will be possible in all states that are defined within the automaton. This is especially useful in a case where a user wants to receive incoming data in all states and store it.

To store data, automatons can define variables. The variables have a data type, which should be a simple data type (see section 4.2.3). Upon a transition, these variables can be assigned, for example based on the data that is received from the robotic middleware. The data from the middleware is represented by the **value** keyword in an assignment.

Figure 4.11 shows an overview of the concepts related to automata in the DSL. It contains definitions for variables, transitions and states.

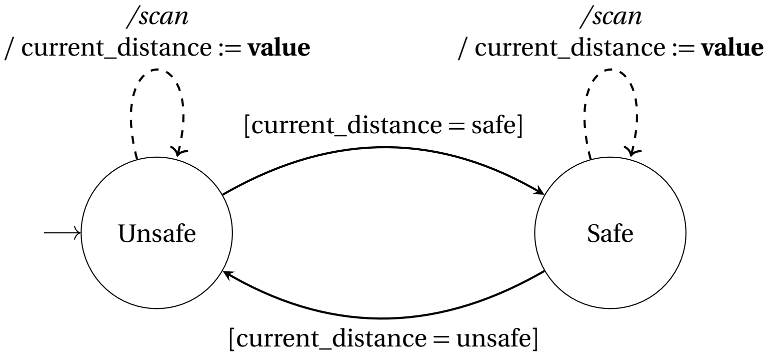

/scan
/ current_distance := **value**
/scan
/ current_distance := **value**
[current_distance = safe]
Unsafe
Safe
[current_distance = unsafe]

Figure 4.10: Example DSL automaton with result transitions and tau transitions.

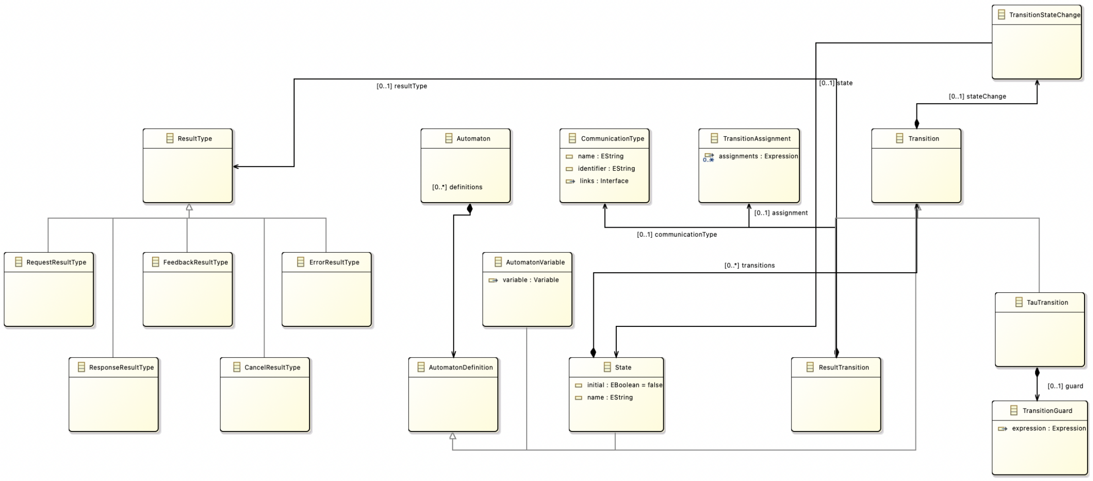


Figure 4.11: Part of the metamodel for the automata concepts of the robot.

The example in the introduction of this section requires the definition of the behaviour of three components: the distance sensor, the obstacle grabber and the line detector. The distance sensor only has a single state, in which it senses the distances and publishes this value. Upon a response from this message, the value is stored in a variable. The variable is defined in the automaton. For the obstacle grabber, there are two states. The first state where it is idle, and a state where it is grabbing an object. Note that the idle state also represents the initial state, and is marked, because the component will eventually return to that state. When started, the component transitions from its initial state to the state where it is grabbing, until it has finished grabbing the object, after which it returns to the initial state. Then, lastly, the line detector also knows two states. There is a state when there is no line, and there is a state where a line was found. From the initial state, there is a result transition which takes the automaton to the state where a line is found. In this state, any incoming correction values are stored in a variable on the automaton. The code for these components is added in listing 4.8.

```
robot ThesisLineFollower {
    // ...

    component DistanceSensor {
        // ..

        behaviour {
            variable current: DistanceEnum = obstructed

            initial marked state sensing {
                on response from distance do current := value
            }
        }
    }

    component ObstacleGrabber {
        // ..

        behaviour {
            initial marked state idle {
                on request to grab goto grabbing
            }

            state grabbing {
                on response from grab goto idle
            }
        }
    }

    component LineDetector {
        // ..

        behaviour {
            variable current_correction: double = 0.0

            initial marked state no_line {
                on response from correction goto line_found
            }

            state line_found {
                on response from no_line goto no_line
            }
        }
    }

    // ...
}
```

Listing 4.8: Automata concept code for running example.

4.2.8 Data provisioning

Controllable communication takes place when certain requirements are met. This means, that there is no direct invocation of communication, much like a method in a regular programming language like Java or C++. Therefore, it is not possible to directly pass data to it.

To solve this problem, the language introduces the concept of provide-statements which link data to the communication with the robotic middlware. These statements provide communication with data, based on *boolean* expressions. These *boolean* expressions determine whether data should be sent along with communication based on the current state of the supervisory controller. More information about the data flow can be found in section 4.3. The concepts on data provisioning can be found in figure 4.12.

The example robot from the introduction requires some provide statements. First, the messages that are sent to the motor require a linear and an angular velocity. For the move message, this means

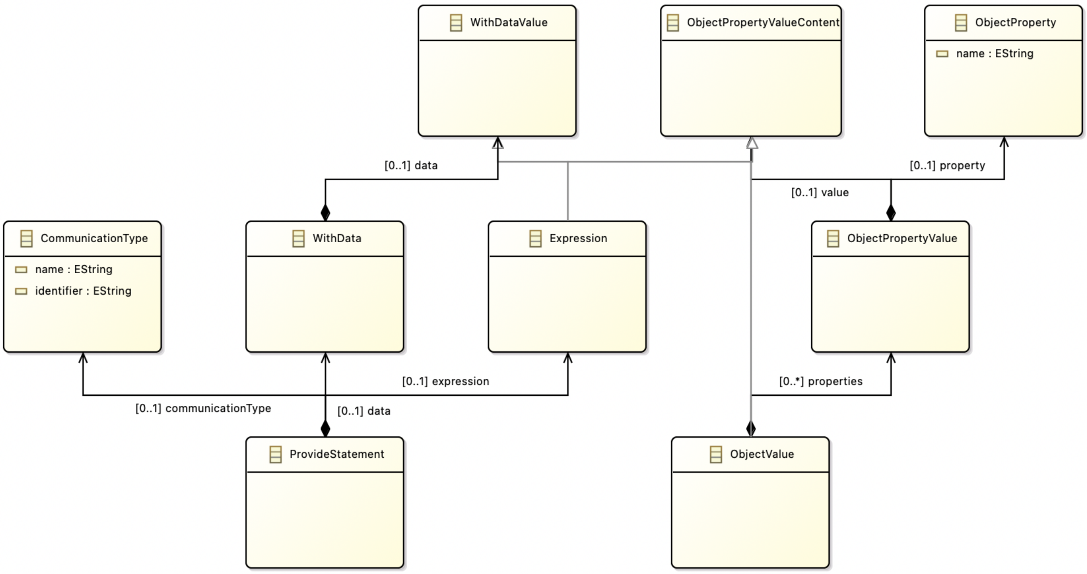


Figure 4.12: Part of the metamodel for the data provisioning concepts of the robot.

that the robot will be provided with a constant value for the linear speed, and a fraction of the line correction value for the angular speed, such that it can steer towards the line. The stop command will be provided with a value of zero for both the linear as well as the angular speed. The service that updates the light state also requires a value, depending on the state of the line detector component, which is added as a condition. The provide statements are outlined in listing 4.9.

```
robot ThesisLineFollower {
    // ...

    // Provide communication with the correct speed
    provide move with {
        linear: { x: 0.4 },
        angular: { z: LineDetector.current / 100 }
    }
    provide stop with {
        linear: { x: 0.0 },
        angular: { z: 0.0 }
    }

    // Enable light when no line found
    provide set_light_state with { ^state: false } if LineDetector.line_found
    provide set_light_state with { ^state: true } if LineDetector.no_line

    // ...
}
```

Listing 4.9: Provide statement concept code for running example.

4.2.9 Communication requirements

In order to restrict the behaviour of controllable communication, the user has to define requirements. Requirements are split into two types:

- Requirements that disallow controllable communication to take place based if a condition holds. This condition is captured using a *boolean* expression.

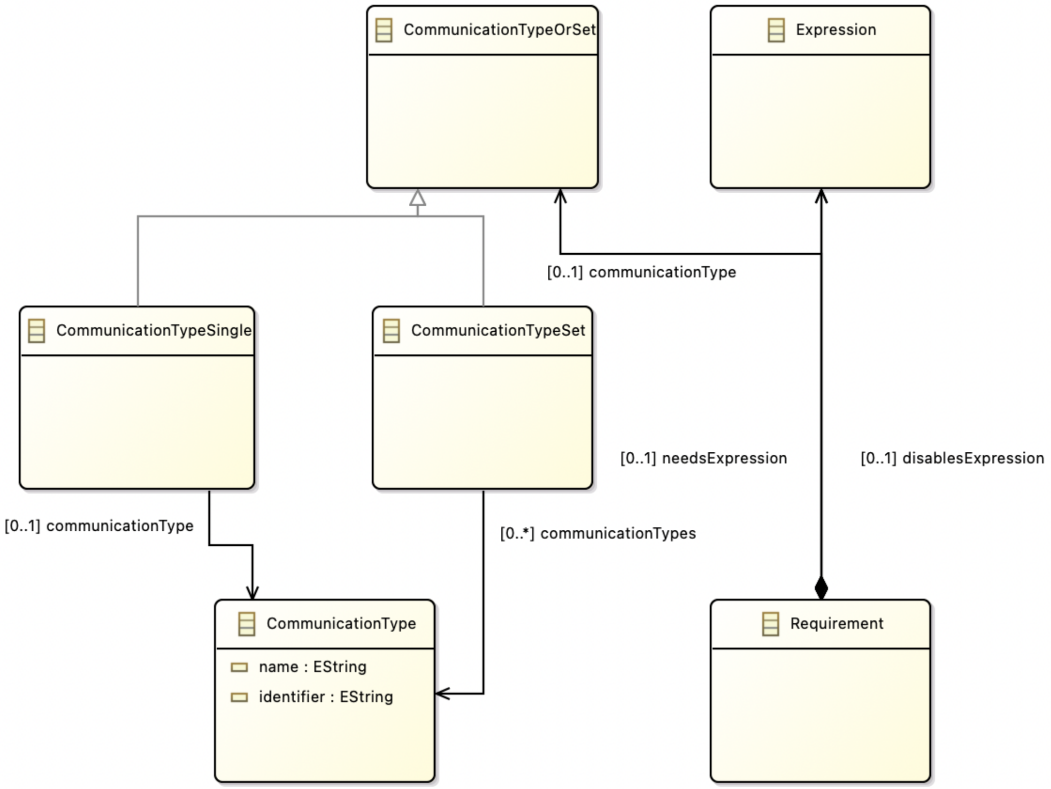


Figure 4.13: Part of the metamodel for the requirement concepts of the robot.

- Requirements that need a condition to hold in order to have the communication take place. This condition is captured using a *boolean* expression as well.

It is possible to specify requirements either for a single communication item, or for a set of communication items.

The model of the requirements is attached in figure 4.13. It contains models that allow requirements to either specify a condition for a single communication type, or for multiple types. Requirements can be of two types. Either, a requirement with a condition that disallows communication based on a condition that holds, or a requirement that needs a condition to hold to allow the communication to take place. Note that the expression concept allows referencing of states.

There are a total of four requirements for the example robot. For the move message, nothing should be in front of the robot and therefore the distance sensor value should be equal to *free*. Furthermore, the robot should not move while grabbing anything and therefore the obstacle grabber should be in state idle. The stop command should be executed if one of these conditions does not hold, so if the distance sensor has something in front or the obstacle is being picked up. The last requirement specifies that if the distance sensor says there is nothing in front of the robot, the obstacle grabber action can not start, as it has nothing to pick up. These requirements are shown in listing 4.10.

```
robot ThesisLineFollower {
    // ...

    // Movement
    requirement move needs DistanceSensor.current = free
    requirement move needs ObstacleGrabber.idle
    requirement stop needs DistanceSensor.current = obstructed or ObstacleGrabber.grabbing
    // Grabbing
    requirement DistanceSensor.current = free disables grab

    // ...
}
```

Listing 4.10: Requirement statement concept code for running example.

4.3 Data Flow

Because all communications that take place happen when certain conditions are met, it is impossible to directly invoke such a communication, and pass data to it. Therefore, the language defines provide-statements which, based on a specific condition, determine what data should be passed. If one or more conditions overlap, it is non-deterministic what data will actually be passed to that communication item.

These statements are converted to an automaton with states for each of the statements. The state of the automaton determines what should be sent. Each communication item has a single data automaton, where the states are defined as $S = \{empty\} \cup P$, with P being defined as the set of all provide statements for that communication item. The labels of these states are generated randomly. In each state, edges to all other states are added, where the transition guard of each edge is defined as the condition specified in the provide-statement.

Each time the controller executes the control loop, it attempts to execute all data-related transitions that are enabled in a random order, hence internally determining what data currently should be sent to the communication item. A sample of such a data automaton can be found in figure 4.14.

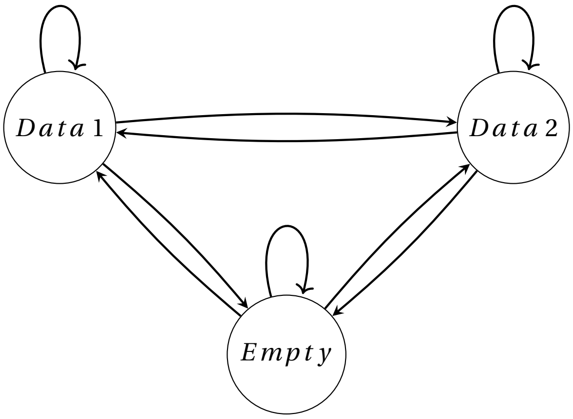


Figure 4.14: Sample data plant for communication items.

4.4 Type System

The language has a few basic data types: *booleans*, *integers*, *strings* and *doubles*. It also defines the special data type *none*, which represents communication that accepts or receives no data. The data types can be used in various places. For example, data can be provisioned to a communication item or data can be used in a multiplication. Not all of these data types are compatible with each other, as it should not be possible to multiply a *string* with a *number* for example. To solve this, the language defines its own type system. Each expression results in an inferred type, and the resulting type is checked with the expected type.

4.4.1 Inference

The inference process of the type system determines a data type for a given expression. It does not yet validate the type, this happens in a separate validator. The formal definition of the type system and the inference rules are outlined in table 4.1.

One of the special concepts of the language, is the access model. The access model provides users with the possibility to reference states, variables, enum values or object properties. The type system first checks what is actually referenced. If the referenced item is a state, the resulting type will always be a *boolean* (that represents whether the automaton is in that state). If the reference was to a variable, the type of the referenced variable will be used. For an enum value, the enum that defines the value is used as the type and lastly, if the reference was to a property of an object, the type of the property definition is used.

If the system is unable to infer a type for an expression, it will return the unknown data type. This can happen when the user has referenced a property or variable that does not exist. Although the DSL code will not compile (because of a broken reference), validation checks are still executed. And, as there is no reference to a variable, it can not infer the type.

$\Gamma \vdash \text{true} : \mathit{bool}$	$\Gamma \vdash \text{false} : \mathit{bool}$
$\Gamma \vdash \text{literal int} : \mathit{int}$	$\Gamma \vdash \text{literal double} : \mathit{double}$
$\Gamma \vdash \text{literal string} : \mathit{string}$	$\frac{\Gamma \vdash \mathrm{E1} : \mathit{bool} \qquad \Gamma \vdash \mathrm{E2} : \mathit{bool}}{\Gamma \vdash \mathrm{E1} \text{ and } \mathrm{E2} : \mathit{bool}}$
$\frac{\Gamma \vdash \mathrm{E1} : \mathit{bool} \qquad \Gamma \vdash \mathrm{E2} : \mathit{bool}}{\Gamma \vdash \mathrm{E1} \text{ or } \mathrm{E2} : \mathit{bool}}$	$\frac{\Gamma \vdash \mathrm{E1} : \mathit{bool} \qquad \Gamma \vdash \mathrm{E2} : \mathit{bool}}{\Gamma \vdash \mathrm{E1} \Rightarrow \mathrm{E2} : \mathit{bool}}$
$\Gamma \vdash \mathrm{E1} = \mathrm{E2} : \mathit{bool}$	$\Gamma \vdash \mathrm{E1} \mathrel{!=} \mathrm{E2} : \mathit{bool}$
$\frac{\mathrm{E1} : \mathrm{T} \in \{\mathit{int, double}\} \qquad \mathrm{E2} : \mathrm{T} \in \{\mathit{int, double}\}}{\Gamma \vdash \mathrm{E1} > \mathrm{E2} : \mathit{bool}}$	$\frac{\mathrm{E1} : \mathrm{T} \in \{\mathit{int, double}\} \qquad \mathrm{E2} : \mathrm{T} \in \{\mathit{int, double}\}}{\Gamma \vdash \mathrm{E1} < \mathrm{E2} : \mathit{bool}}$
$\frac{\mathrm{E1} : \mathrm{T} \in \{\mathit{int, double}\} \qquad \mathrm{E2} : \mathrm{T} \in \{\mathit{int, double}\}}{\Gamma \vdash \mathrm{E1} >= \mathrm{E2} : \mathit{bool}}$	$\frac{\mathrm{E1} : \mathrm{T} \in \{\mathit{int, double}\} \qquad \mathrm{E2} : \mathrm{T} \in \{\mathit{int, double}\}}{\Gamma \vdash \mathrm{E1} <= \mathrm{E2} : \mathit{bool}}$
$\frac{\Gamma \vdash \mathrm{E} : \mathit{bool}}{\Gamma \vdash !\mathrm{E} : \mathit{bool}}$	$\frac{\Gamma \vdash \mathrm{E} : \mathit{int}}{\Gamma \vdash \text{-}\mathrm{E} : \mathit{int}}$
$\frac{\Gamma \vdash \mathrm{E} : \mathit{double}}{\Gamma \vdash \text{-}\mathrm{E} : \mathit{double}}$	$\frac{\mathrm{E1} : \mathrm{T} \in \{\mathit{int, double}\} \qquad \mathrm{E2} : \mathrm{T} \in \{\mathit{int, double}\}}{\Gamma \vdash \mathrm{E1} \,/\, \mathrm{E2} : \mathit{double}}$
$\frac{\Gamma \vdash \mathrm{E1} : \mathit{double} \qquad \mathrm{E2} : \mathrm{T} \in \{\mathit{int, double}\}}{\Gamma \vdash \mathrm{E1} + \mathrm{E2} : \mathit{double}}$	$\frac{\mathrm{E1} : \mathrm{T} \in \{\mathit{int, double}\} \qquad \Gamma \vdash \mathrm{E2} : \mathit{double}}{\Gamma \vdash \mathrm{E1} + \mathrm{E2} : \mathit{double}}$
$\frac{\Gamma \vdash \mathrm{E1} : \mathit{double} \qquad \mathrm{E2} : \mathrm{T} \in \{\mathit{int, double}\}}{\Gamma \vdash \mathrm{E1} \text{ - } \mathrm{E2} : \mathit{double}}$	$\frac{\mathrm{E1} : \mathrm{T} \in \{\mathit{int, double}\} \qquad \Gamma \vdash \mathrm{E2} : \mathit{double}}{\Gamma \vdash \mathrm{E1} \text{ - } \mathrm{E2} : \mathit{double}}$
$\frac{\Gamma \vdash \mathrm{E1} : \mathit{double} \qquad \mathrm{E2} : \mathrm{T} \in \{\mathit{int, double}\}}{\Gamma \vdash \mathrm{E1} * \mathrm{E2} : \mathit{double}}$	$\frac{\mathrm{E1} : \mathrm{T} \in \{\mathit{int, double}\} \qquad \Gamma \vdash \mathrm{E2} : \mathit{double}}{\Gamma \vdash \mathrm{E1} * \mathrm{E2} : \mathit{double}}$
$\frac{\Gamma \vdash \mathrm{E1} : \mathit{int} \qquad \Gamma \vdash \mathrm{E2} : \mathit{int}}{\Gamma \vdash \mathrm{E1} + \mathrm{E2} : \mathit{int}}$	$\frac{\Gamma \vdash \mathrm{E1} : \mathit{int} \qquad \Gamma \vdash \mathrm{E2} : \mathit{int}}{\Gamma \vdash \mathrm{E1} \text{ - } \mathrm{E2} : \mathit{int}}$
$\frac{\Gamma \vdash \mathrm{E1} : \mathit{int} \qquad \Gamma \vdash \mathrm{E2} : \mathit{int}}{\Gamma \vdash \mathrm{E1} * \mathrm{E2} : \mathit{int}}$	$\Gamma \vdash \text{state reference} : \mathit{bool}$
$\frac{\text{variable} : \mathrm{T} \in \Gamma}{\Gamma \vdash \text{variable} : \mathrm{T}}$	$\frac{\text{property} : \mathrm{T} \in \Gamma}{\Gamma \vdash \text{property} : \mathrm{T}}$
$\Gamma \vdash \text{enum} : \mathit{enum}$	

Table 4.1: Language type system inference rules.

4.5 Validation

Next to syntax validation, there are more constraints which are checked to ensure correctness of the model. A violation of such a constraint will result in an error that is presented to the user, and it will prevent the generator from generating any code.

4.5.1 Data provisioning check

Every communication item that is defined in the model can be supplied with data by using provide statements. But, in some cases this is not allowed. This check validates that no data is passed to a message that is coming out of the controller, and therefore is an uncontrollable event.

4.5.2 Result type check

Components can update their state based on uncontrollable behaviour and events that are received from messages, services or actions. For messages, this means that user can listen for response events. In the case of services, there is also the possibility to watch for a response, and actions have the option to transition upon feedback or cancellation. This check validates whether there is a result transition that watches for a disallowed event.

4.5.3 Integer range required check

Some variables are required within the supervisory controller, and some are not (more information can be found in the section about the generators for supervisory control theory 5.2). In the supervisory controller, integer variables require users to specify a domain for the integer. In this validation rule, it is checked whether a variable is required within the supervisory controller and if it has no integer range present, it will throw an error.

4.5.4 Interface link required check

Communication items have the option to link an interface, if they are using a custom data type. In this validation rule, the data type of every communication type is checked, and if it is found to be custom, it is checked whether it has an associated interface, which is used to determine the imports, parameters and linked libraries.

4.5.5 No assignment on messages to node check

If a result transition is triggered upon sending a message to a node (controllable behaviour) it is not allowed to update any variables, because no data is coming in. This rule checks each message to see the direction and validates whether there are any assignments.

4.5.6 No assignment outside scope check

In CIF, the concept of global read and local write is used [37]. This means that all automatons can read variables from another automaton, but not write to it. They can only write to variables that are declared within the scope of their own automaton. As some variables that are declared in the model are also specified in plants with supervisory control theory, it means that the language should also adhere to this concept. This validation rule checks whether this is the case and will present the user with an error if not.

4.5.7 Single component behaviour check

The syntax of the DSL allows users to have the definition of communication and behaviour of a component in no given fixed order. This also means that the syntax allows the specification of multiple component behaviours. There can only be a single behaviour definition per component, which is checked using this validation rule.

4.5.8 Single default enum case check

An enum should have exactly a single default case, which is used as a fallback. This rule checks whether the user has specified a default transformation rule.

4.5.9 Single initial state check

Similar to the enum validation rule, there is also a check which ensures that every automaton in the language has an initial state. The language only supports automata that are in one state at a time, and therefore there should always be a single initial state.

4.5.10 Marked state check

To ensure that a supervisor can be synthesized, all of the reachable states (starting from the initial state) should be able to reach a marked state. This is checked, and if this is not the case, the user will be presented with an error.

4.5.11 Type check

To check whether users have specified the correct types for assignments, data provisioning and guards, the type system is used. The types of different expressions are inferred using the type system, and then compared to the expected types. If something is wrong, an error annotation is added on the corresponding line, together with more information about the inferred data type and the expected data type. For *boolean* operations (conjunctions, disjunctions and negations) this means that both the left-hand-side, as well as the right-hand-side, should be of type *boolean*. For numeric operations, the values on both sides should either be of type *double* or *integer*. Next to that, it is validated that equations have the same data type.

There are also some places that require expressions to be of a specific type. For example, transition guards and data provisioning conditions should both be of type *boolean*. Also, the provided data to a communication item should be of the type that was specified in the definition. This also holds when the provide statements define an object value, then the value of a field should have the data type that was defined in the custom data type declaration. Both initial values for, as well as assignments to, variables are checked to determine whether they have correct types.

Apart from the type validation, there is also another part to type checks. Users can provide communication items with object values. If they provide the object with a property that is not defined on the custom data type, the reference can not be resolved. This will also result in an error which is visible to the user.

4.5.12 Uniqueness check

Some entities require unique names, to ensure that they can be identified correctly. There is a validation rule for unique state, communication, enum value, data type and topic names. Duplicate topic names (in separate components) do not produce an error, but rather a warning which notifies the user of unexpected consequences.

4.5.13 Enum value name state overlap

Besides the fact that enum value names must be unique within the robot, the names of these enum value names also can not overlap with the name of any automata state in the robot. This is done because CIF will consider the enum value as a state (and therefore have different semantics).

4.5.14 Variable data type check

There are limitations to the types that a variable can be. A variable can never be an object or an array. Furthermore, if a variable is required in the controller synthesis process, it can not have the type of a *double* or a *string*.

4.6 Assumptions

In order for the supervisory controller to work within the robotic middleware, there are some assumptions on the communication between nodes in the middleware. If these assumptions do not hold, the controller could end up in an unexpected state.

The first assumption is that there is no communication between nodes that influences the state of a component and that the controller is unaware of. As this could internally change the state of a component, users might observe unexpected behaviour. This could be solved by either altering the developed model for the components to include the additional communication, or by altering the nodes to add additional communication to notify the supervisory controller of the state changes. This additional communication should be included in the DSL model as well. An example of this disallowed behaviour is shown in figure 4.15, where the controller is aware of communication via topic A, but not topic B.

The other assumption is that no two components publish a message to the same topic. If two components were to do so, it means that the controller is unaware from which component the message came, therefore not knowing which component's state should be updated. This case is visualized in figure 4.16, where both nodes publish to the same topic, topic A. Although the language will provide the user with a warning if two components publish to the same topic, this is not water tight, as there is no one-to-one mapping from a component to a node. There are two ways to solve this. First, if the language user has the option to alter the topic name of a message, it could change them to a unique value. If not, the nodes which the components represent could be started with remappings. Remappings ensure that all communication from and to a node for a given topic, is remapped to a different topic. Publishers will publish to that new topic name and subscribers will listen to the new topic name.

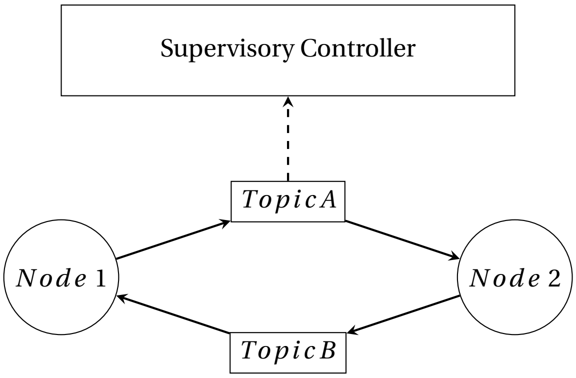


Figure 4.15: Visualization of assumption that nodes do not communicate with each other such that it could change the state of either component without the controller knowing.

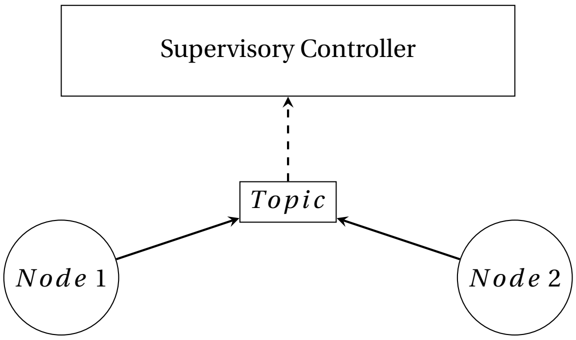


Figure 4.16: Visualization of assumption that no two nodes publish to the same topic.

4.7 Supervisory Layer

The focus of this thesis has been on the supervisory controller, where a controller is synthesized based on a set of requirements. However, it is also possible to look at it from a different perspective: a supervisory layer. By doing so, the supervisor layer can be connected to an existing controller and check whether the behaviour of the controller is allowed according to the requirements. This can be extremely helpful for cases where a controller already exists, but the existing solution has no formal guarantees on its behaviour. The supervisory layer can then block all communication from that existing controller that is not allowed to occur according to the requirements that were specified.

4.7.1 Overview

The idea is that all communication from the existing controller to the middleware is supervised by a supervisory layer, in the form of a node in the middleware. But, instead of starting the existing controller normally, it will start the controller with remapped topic and service names, which allows the supervisory layer to control the communication from and to the controller. The original names are defined in the model of the DSL, and remapped to an unique name. Then, the supervisor node subscribes to these unique names. All controllable events (publishing of messages, requests to services or sending goals to actions) are checked against the current state of the supervisor, and if they are allowed based on the set of the requirements, the message is passed to the original topic, the request to a service is forwarded to the original server or the goal is sent to the original action server.

Instead of updating the state of the supervisory controller in the control loop, and trying to execute all possible controllable communication, the supervisor waits for commands from the existing controller. For services and actions this means that the supervisor creates a proxy server, which forwards all data, if allowed in the current state. The supervisor also subscribes to all uncontrollable events. Note that uncontrollable events are also remapped (and therefore the controller is not instantly made aware of them), because it allows the supervisor to update its internal state before the existing controller can. If an uncontrollable event would reach the existing controller before the supervisor, it might lead to unexpected behaviour as it might allow or disallow controllable communication incorrectly. After communication has taken place (both incoming or outgoing), the robot attempts to execute all tau transitions.

It is possible that the controller executes behaviour that is not allowed. In that case, the supervisor informs the controller by means of a message. A message is published to the topic, containing information about the disallowed behaviour of the robot.

An overview of the structure of the communication is shown in figure 4.17. In this overview, the communication between the components is relatively simple using messages.

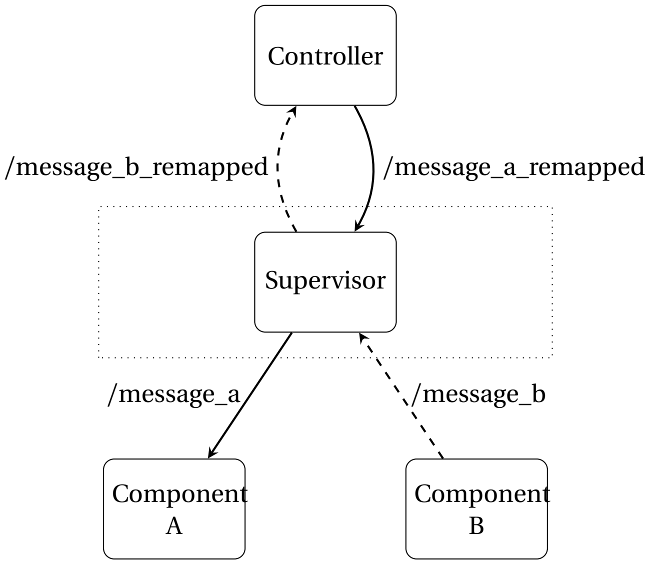


Figure 4.17: Overview of the structure of the supervisory layer. The dashed rectangle represents what is generated from the DSL.

4.7.2 Differences to supervisory controller approach

Although there is a lot of overlap between the two approaches, there are some important differences. The main one, is that the supervisor will only start communication with the middleware based on what the controller attempts. It supervises the communication. As a result, the concept of provide statements is not relevant to the supervisor, because it will only pass on data that is coming from the controller or a component in the middleware.

Another implication of this, is that there is no timer that executes transitions in the control loop. These transitions are executed when communication is started from the controller. All silent transitions that are enabled are executed after communication from the controller to the supervisor has occurred.

4.7.3 Limitations

There are some limitations to this approach. First, messages, services and actions can not have duplicate topic names. For supervisory controllers, it was possible to have multiple messages in the model that map to the same topic name, as long as they are coming from the same component. However, this is not allowed in the supervisor layer approach as the supervisor can not distinguish which message was actually invoked.

Furthermore, it was found that the remapping functionality of ROS does not work for actions, meaning that the controller will send goals to the actual action server. The supervisor still generates a proxy server with a unique name, so users have the option to connect to the supervised action server manually.

Lastly, the supervisor does not initiate communication. Some robots (like the Turtlebot) are controlled by sending a command to the robotic middleware using a message that determines the linear and angular speed. If the supervisor layer disallows the passing of this message because an emergency stop was pressed, for example, the robot will not directly stop as it has not received a message that sets the linear and angular speed to 0. Although some robots have a fallback which stops the robot when no new speed has been received for a while, there are also other options to solve this. One option is to create a component which moves the robot at a certain speed, but for a limited amount of time.

4.8 Workbench

The language workbench that was used, is Eclipse XText. XText is a framework that can be used for the development of domain-specific languages, containing different tools that streamline the language development process. It will generate a parser, syntax highlighting and auto completion [14]. All of this is generated automatically based on an XText grammar. The language itself still specifies scopes, validation rules and generators. One of the main reasons that XText was used over MPS, is the support for multiple model to text generators. From a single model, multiple files can be generated, contrary to how JetBrains MPS works.

Next to XText, the language also uses XTend [1], an Eclipse project for modern Java expressions, allowing easy templating for model to text transformations. XTend code is transformed into Java code, that can run on the Java JVM just as any regular Java program.

Often, the use of a specific language workbench comes with a deep integration into an integrated development environment (IDE) [7]. This has advantages in term of user experience, but means that the user of the language will often locked into that specific IDE. To solve this, the language server protocol has been developed by Microsoft [32]. Because XText generates support for the Language Server Protocol automatically, it means that language users can use their favorite editor, as long as it supports the protocol and an extension has been built to integrate the support.

4.8.1 Supported editors

Although the language was built with Xtext, usage of the DSL is absolutely not bound to Eclipse. There is an Eclipse-plugin, which language users can install. But, XText also comes with out-of-the-box support for the Language Server Protocol, which allows other editors to integrate with the DSL as well. In this project, a Visual Studio Code extension was built as well, which starts the language server using Java. This means that Java must be present on the users system for the user to run the tool. Both editors support syntax highlighting, cross-referencing, auto completion, syntax checking and code generation for the DSL.

[1] https://www.eclipse.org/xtend/

Chapter 5

Generators

5.1 Introduction

Several artifacts are generated from the DSL. These artifacts are generated in multiple steps. First, using a model-to-text transformation, code is generated for supervisory control theory, specifically for the CIF-tool. Then, supervisory controller synthesis is applied using that tool. After synthesis is applied, the code generator for the supervisory controller can be executed. This resulting controller code is then used by an additional model-to-text generator from the DSL, that creates a ROS1 and ROS2 node with bindings to the supervisory controller. Figure 5.1 shows the transformation chain from the DSL code to a ROS node. All rectangles represent an artifact, whereas ovals represent generators. The arrows stand for the data flow between artifacts and generators, where the source of the arrow represents the output of an artifact and the sink defines that artifact as an input to the generator. Generators with double lines stand for generators from the supervisory controller tool.

This chapter describes the generated concepts and code from the model, both for supervisory control theory, as well as the ROS-nodes and how they integrate with each other. Furthermore, an explicit mapping is defined between the concepts of the language and the concepts of ROS and supervisory control theory.

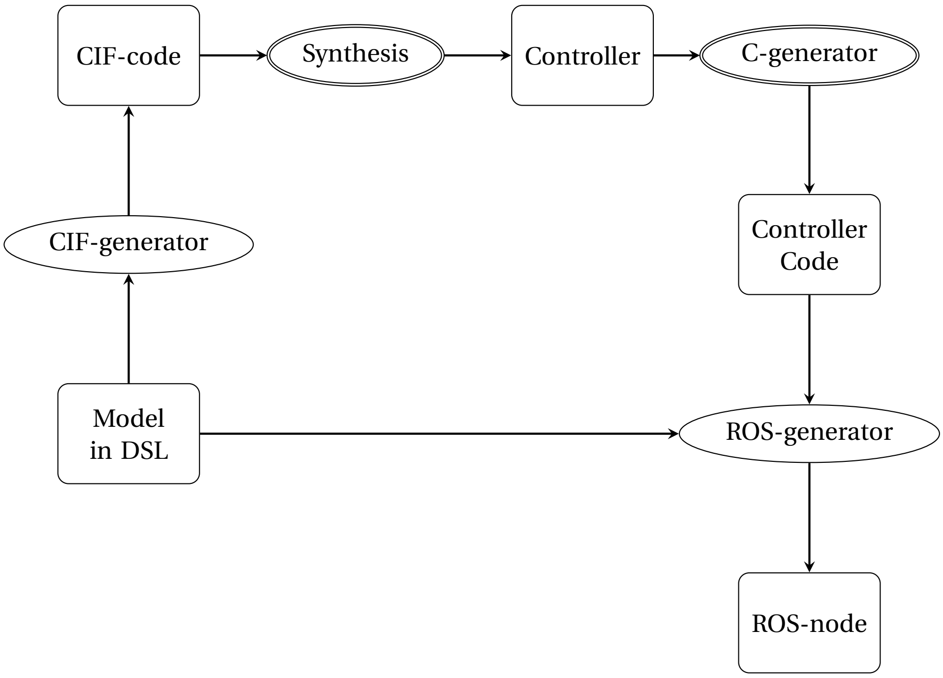


Figure 5.1: Transformation chain overview of artifact generation based on DSL. Rectangles represent artifacts, ovals represent generators. Generators with double lines stand for generators from the supervisory controller tool. Arrows transfer the output (source) to an input (sink).

5.2 Supervisory Control Theory

The heart of the controller is built using supervisory control theory (SCT). SCT is being used to apply event-based controller synthesis based on a set of plants and a set of requirements. The code is converted to C-code using an external tool, which is embedded into a ROS-node. Although within this thesis CIF was chosen, any tool that supports supervisory controller synthesis could have been used, as equal concepts apply. This chapter highlights the generated plants and requirements based on the DSL model, and shows the workings of the technical integration between CIF and the DSL. Furthermore, some modifications had to be made to the CIF code generator, which have been offered to the open source project.

5.2.1 Plants

Several plants are constructed based on the DSL model. This section describes all of the generated plants, their states, transitions and variables.

Communication

All communication items are represented by plants. Each plant defines controllable events, uncontrollable events, inputs, states and edges. The controllable events are used to send or start communication from the controller, whereas the uncontrollable events represent events that notify the controller of incoming data. If the data is required within a plant, inputs store this incoming data. Depending on the type of communication, states represent the phase that the communication currently is in.

Plants that support incoming data from a communication item, store this data in inputs. For basic data types, this results in a single input property. However, for complex objects, an input is created for each field. An algorithm (described in algorithm 1) finds all properties using a recursive search, and checks whether the property should be present in a plant, and if so, defines an input. A property should be present if its value is used in a guard or requirement. This can be directly, or indirectly, where the value is first stored in a variable of a plant. Note, if an input is defined within a plant, the field itself can only be of a simple data type (*enums*, *integers* with a range or *booleans*). So, arrays are not supported for example. For more information about this, consider the section on validation rules 4.5.

```
input  : A property $property$ (with type and name) and a prefix $prefix$ that is prepended
         to the inputs.
output: A set of inputs that are required within CIF for a given data type.
$R \leftarrow \emptyset$
if property.type is object then
    for child in property.type do
        $R \leftarrow R \cup CIFInputs(child, prefix + \_ + child.name)$
    end
    return $R$
else
    if property required in some plant then
        return {prefix}
    else
        return $\emptyset$
    end
end
```

Algorithm 1: Algorithm to define plant input properties for a given data type.

The plants that correspond to messages are relatively simple. Each message is defined within the scope of a component, and depending on the direction of the message with respect to the component, the definition changes. For incoming messages to a component, the plant has a controllable edge, which can occur in the single state of the plant. However, if the message is going out of the component, there is a single uncontrollable edge within the plant. The name of the plant is the name of the message, prefixed with *message_*.

If the data of this message is required within a guard or requirement, the plant defines inputs that store the incoming data so that it can be used in guards or assignments. An example of a plant for a message is shown in listing 5.1, represented as CIF-code.

```
plant message_sample:
    uncontrollable u_response;
    input bool i_response_object_value_one;
    input bool i_response_object_value_two;

    location:
        initial; marked;
        edge u_response;
end
```

Listing 5.1: CIF code for a message plant.

The plant of a service looks a bit more complex than that of a message. In ROS2, services are asynchronous, whereas in ROS1, they are synchronous. In our plant, they are modelled equally: asynchronous. The plant distinguishes four different states: idle, waiting for response, ready and error. A service plant is in the (initial) idle state when it is waiting to be executed. In this state, there is an edge that allows the invocation of a service, which takes the plant to the next state: waiting for response. When an error occurs, an uncontrollable event will take the automaton to the error state. If a response was received, the plant is taken to the ready state. Similar to how messages work, the result values are stored in an input field if they are required in further guards, requirements of assignments. An example of a service plant is shown in listing 5.2, again represented as CIF-code. The name of the plant is the name of the service, prefixed with *service_*.

```
plant service_sample:
    controllable c_trigger, c_reset;
    uncontrollable u_response, u_error;

    input bool i_response_object_value_one;
    input bool i_response_object_value_two;

    location idle:
        initial; marked;
        edge c_trigger goto wait_for_response;
    location wait_for_response:
        edge u_response goto ready;
        edge u_error goto error;
    location ready:
        edge c_reset goto idle;
    location error:
        edge c_reset goto idle;
end
```

Listing 5.2: CIF code for a service plant.

For ROS actions, plants also consist of four states: idle, executing, ready and error. Again, idle represents a state where the action can be started using a trigger edge, and if the action is executed, the plant transitions to an executing state. From here, both uncontrollable as well as controllable events can occur. First, the action server can provide feedback about the execution using the uncontrollable feedback event. This keeps the plant in the same state, whereas the response event will transition the automaton to the ready state. In case something goes wrong (for example, an action server is unavailable), an uncontrollable error event occurs, transitioning the plant into that corresponding state. The name of the plant is the name of the action, prefixed with *action_*.

Both responses as well as feedback sent by the action are stored in separate input fields of the plant. An example of such an action plant can be found in listing 5.3.

```
plant action_sample:
    controllable c_trigger, c_reset, c_cancel;
    uncontrollable u_feedback, u_response, u_error;

    input int[0..20] i_feedback_object_value_one;
    input int[0..20] i_feedback_object_value_two;

    input bool i_response_object_value_one;
    input bool i_response_object_value_two;

    location idle:
        initial; marked;
        edge c_trigger goto executing;
    location executing:
        edge u_feedback;
        edge u_response goto ready;
        edge u_error goto error;
        edge c_cancel goto idle;
    location ready:
        edge c_reset goto idle;
    location error:
        edge c_reset goto idle;
end
```

Listing 5.3: CIF code for an action plant.

Behaviour

If a component has associated behaviour, this behaviour is transformed into a plant as well, where the name of the plant is the name of the component prefixed with *component_*. The behaviour within the DSL is defined as an automaton with states and events. There are two possible transitions between the states: result transitions and tau transitions.

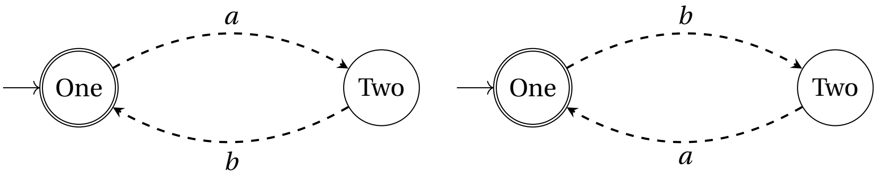


Figure 5.2: Example of a case with two plants where an uncontrollable event can not occur in regular supervisory control theory.

For result transitions, the represented edge in the plant depends on the type of result. Result types depend on the type of communication, and can have the following values: request, response, feedback, error or cancel. The edges originate from the plant of the communication item, and synchronize (meaning that the transition must be enabled in all plants). These correspond to the plant events as described in table 5.1.

Result Transition Type	**Plant event**
request	<communication plant>.c_trigger
response	<communication plant>.u_response
feedback	<communication plant>.u_feedback
error	<communication plant>.u_error
cancel	<communication plant>.c_cancel

Table 5.1: Mapping between result transition types and the corresponding plant event.

If a plant includes an edge within a specific state, it means that the plant must be in that state in order for the edge to be able to execute. If this is not the case, the transition can not take place. This can result in unwanted behaviour, as the robotic middleware can always provide the supervisory controller engine with data at any moment. If a component defines a transition that occurs upon a message from a specified topic, but this transition is only defined in a single state, and the plant is not in that state it means that none of the plants can perform the transition, meaning that the system is unaware of the incoming data. An example of such a scenario, with two uncontrollable events a and b, is shown in figure 5.2. In the left plant, event a is enabled in state one, but in the right plant it is only enabled in state two. The same holds for event b, but this event is only enabled in state two in the left plant, and in state one in the right plant. This would mean that the event can not occur.

The solution to this problem can be split into two parts. First, it is important that result transitions have no guard. If they were to have a guard, and the guard was not enabled, it would mean that none of the plants can be notified of new data.

The other part of the solution is that each state within the plant of component behaviour should be extended with additional edges. These edges have no state changes and no guard, but ensure that these events can always occur, no matter the state of the plant. For each state of a (behaviour) plant, all communication items in the robot are scanned, and all the events from table 5.1 are added if they were not already present. It should be made sure that no duplicates exist, because otherwise the robot might end up in an unexpected state, as the same edge defines different output states. An example is shown in figure 5.3.

Next to result transitions, component behaviour automata can also make use of tau transitions. The tau transitions specify a guard and a possible state change. Although CIF supports tau transitions, they can not be used in supervisory controller synthesis [36]. To solve this, the SCT generator generates random strings for controllable events. These controllable events are defined on the plant, and at every iteration the controller will try to execute them, if enabled.

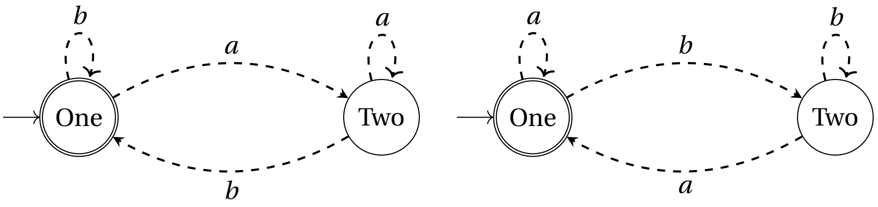


Figure 5.3: Example of a case with two plants where an uncontrollable event can occur, because of added self-loops.

Data

Some communication items require data to be passed along. To do so, the provide statement was introduced. If a communication item has one or more associated provide statements, a plant will be generated that represents the data that is passed. This plant always has a none state, that states that no data can be passed at the time. Then, for each provide statement a new location is added. The name of this location is randomly generated. Then, each of these generated locations contains edges to all other generated locations, along with the guard that was specified on the corresponding provide statement. There is also an edge that goes back to the none location, whose event is only executed when none of the guards hold anymore.

An example of a data plant is listed in listing 5.4, represented in CIF-code. Note, that in this example the names of the edges and locations are not randomly generated. This is done for readability.

```
plant data_sample:
  controllable c_data1, c_data2, c_none;

  location none:
    initial; marked;
    edge c_none when not (ExampleComponent.state1) and not (ExampleComponent.state2) goto none
        ;
    edge c_data1 when ExampleComponent.state1 goto data1;
    edge c_data2 when ExampleComponent.state2 goto data2;
  location data1:
    marked;
    edge c_none when not (ExampleComponent.state1) and not (ExampleComponent.state2) goto none
        ;
    edge c_data1 when ExampleComponent.state1 goto data1;
    edge c_data2 when ExampleComponent.state2 goto data2;
  location data2:
    marked;
    edge c_none when not (ExampleComponent.state1) and not (ExampleComponent.state2) goto none
        ;
    edge c_data1 when ExampleComponent.state1 goto data1;
    edge c_data2 when ExampleComponent.state2 goto data2;
end
```

Listing 5.4: CIF code for a data plant.

5.2.2 Requirements

The transformation from requirements in the DSL to supervisory control theory is relatively straight forward. In supervisory control theory, a requirement is defined as a (controllable) event and a condition which is captured as a *boolean* expression. The requirement can either state that the event needs this condition to hold in order for the event to be enabled, or that the condition disables the event if it holds. The DSL allows users to specify one or more communication items that either require an expression to hold in order to be enabled, or to specify an expression that disables the execution of one or more communication items. If multiple communication items are specified within a single requirement, a single supervisory control theory requirement is generated for each of them. The event that the requirement should hold for, is in this case the trigger edge of a communication item plant.

Two generated CIF requirements, that both represent the same condition, are attached in listing 5.5.

```
requirement message_sample.c_trigger needs Component.stateA;
requirement not Component.stateA disables message_sample.c_trigger;
```

Listing 5.5: CIF code for two equal requirements.

Cancel

For actions, it is a bit different. As actions are meant for longer-running tasks, it is possible that an action can be started based on the requirements, but during the execution of the action the state of the controller changes, meaning that the requirement does not hold anymore. As actions can be cancelled using a controllable event, some actions require an additional requirement. If an action has one or more associated requirements, an extra requirement is generated, but for the controllable cancel event. The expression of this requirement can be described as the case when any of the requirements to start the action does not hold anymore.

This expression is a disjunction of different expressions, and can be constructed as follows. Given an action, all requirements associated to this action are collected. For each of these requirements, a term is added to this disjunction. If the requirement needs an expression to hold, the negation of the expression is added to the disjunction. If the expression of a requirement disables an action, the expression itself is added to the disjunction. Then, if the disjunction is not empty, the requirement to cancel the action is added to the generated code. The algorithm that computes this boolean expression is described in algorithm 2.

```
input  : The requirements $R$ of an action
output: A boolean expression that represents when an action should be cancelled.
O ← True
for $r \in R$ do
    if $r$ is needs requirement then
    else
        O ← O ∨ (¬ r.condition)
    end
    O ← O ∨ r.condition
end
return $O$
```

Algorithm 2: Algorithm to compute the *boolean* expression for the cancel requirement of an action.

5.2.3 Enums

The language allows the definition of enums, which can help the controller to process complex data, without suffering from an extreme growth of the amount of possible states within the controller (which slows down the synthesis process). The conversion of a value to the corresponding enum value happens within the ROS-node. Therefore, the supervisory controller synthesis process only needs the names the possible enum values. Although CIF supports having multiple enums, the values of these enums can not have overlapping names. Currently, it means that this limitation also holds for the DSL. An example of such a definition can be found in listing 5.6.

```
enum sample_enum = one, two, three;
```

Listing 5.6: CIF code for an enum.

Language concept	Supervisory Control Theory concept	Cardinality
Message	Plant	One-to-one
Service	Plant	One-to-one
Action	Plant	One-to-one
Enum	Enum	One-to-one
Requirement	Requirement	One-to-many
Provide statement	Plant	Many-to-one
Component Behaviour	Plant	One-to-one

Table 5.2: Mapping between language concepts and concepts from supervisory control theory.

5.2.4 Data elimination

To prevent state space explosion and support more complex data, the SCT-generator eliminates any inputs and variables that are not relevant to the supervisor. The first step is that it checks whether a variable is required for the supervisory controller. It does so by checking whether a variable is used in a:

- Transition guard
- Data condition
- Requirement

If it is used in one or more of these items, then the variable will be defined within the plant. In the case that is found that a variable is not required in the controller, it will be defined as a variable within the ROS-node. In this case, the variable could be used to pass incoming data to anther node, for example, and therefore is not relevant to the controller.

Next to variables, data elimination is also used at plant inputs. Inputs are used to handle incoming data from the robotic middleware. There can be more than one input for a communication item, if the data is in the form of a complex object type (then the inputs are generated as in algorithm 1). The elimination check determines whether an input is required in the supervisory controller by looking at all assignments to variables upon incoming data, and checking whether that variable is referenced in any of them.

5.2.5 Mapping

The language uses several concepts from supervisory control theory. Most of the concepts are a one to one mapping. Communication items (messages, services and actions) are all directly converted to a SCT-plant for example. The same holds for a requirement and component behaviour. For provide statements, it is a bit different. A group of provide statements for a single communication item is mapped to a single CIF-plant. An overview can be found in table 5.2.

5.2.6 CIF

Within this thesis, CIF is used to apply controller synthesis. The concepts w.r.t. supervisory control theory in CIF are the same as the concepts from the theory itself, meaning that any other tool could be used that fulfills the following requirements:

- The tool supports supervisory control theory (plants and requirements)
- The tool supports supervisory controller synthesis
- The tool supports code generation

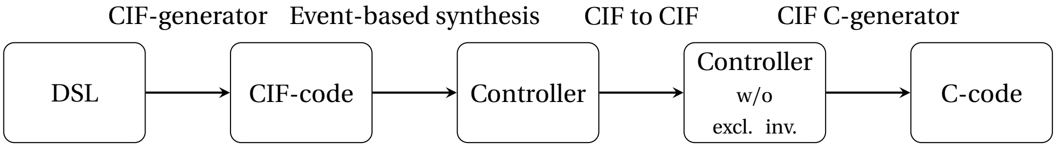


Figure 5.4: Overview of the technical integration with CIF.

Technical integration

There are multiple options that can be used to integrate and communicate with CIF from external tools. The first is option is to use the command line interface of CIF. Although this seems like the most easy option to integrate with CIF, this comes with some downsides:

- The user should have CIF installed on their computer.
- The DSL should be aware of the install location of CIF.
- Although it is a command line tool, Eclipse will still be started in the background, adding a lot of overhead.
- Different versions of CIF may break the DSL generator.

That is why the DSL in this thesis uses a different approach. CIF is developed using Java, and so is the DSL. Java allows reusing code between different applications in the form of JARs (Java Archives). During the build process of the DSL (which uses Gradle), an additional build command is executed that downloads Eclipse Escet from Git (at a specific branch or tag) and builds all the required projects from the Eclipse Escet bundle (cif, common, thirdparty and setext). Because Eclipse Escet is bundled using Maven, the build process is invoked from Gradle but executed with a simple command. The build of the Escet project results in a set of JARs, which are added as dependencies to the language. This allows the DSL to directly invoke and use Java classes from CIF. By embedding the CIF toolset like this, tests can ensure that the CIF version works as expected and will not change unexpectedly. Furthermore, the user does not need to have CIF (and Eclipse) installed.

After the generator has generated a CIF-file, the CIF classes will be invoked. CIF defines several apps, that are responsible for operations executed on an input CIF-file. The language uses these apps as if they were ran from the command line, by providing the arguments as an array of strings. The full process is visualized in figure 5.4.

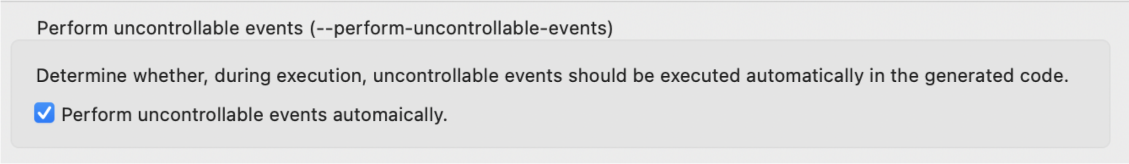
The process uses three tools that result in code for the controller. The first step is to apply event-based controller synthesis, after which the resulting file is transformed using a CIF to CIF transformation, which eliminates the state/event exclusion invariants (required for code generation). Then, the code generator is executed.

One of the problems of executing the tools like this, is that CIF writes output to the console and exits the process after it has finished. To prevent this, output is redirected and captured to a different output stream, and the exit code is saved to a variable, rather than the program being stopped.

Contributions

CIF supports code generation to multiple programming language, including C. Although the generated code will be used within C++, the C code can be compiled such that it can be used within C++. However, the interface of this code does not expose all the required methods and functionalities that the controller would need. As changing generated code comes with a risk of breaking code with newer versions of the language, the code generator interface changes have been implemented in CIF itself, based on the Eclipse Escet [1] source. These changes are made based on a fork of the source code, and were offered to the maintainers of the Eclipse Escet project.

[1] https://gitlab.eclipse.org/eclipse/escet



Perform uncontrollable events (--perform-uncontrollable-events)

Determine whether, during execution, uncontrollable events should be executed automatically in the generated code.

☑ Perform uncontrollable events automaically.

Figure 5.5: Additional option to the C-generator of CIF that determines whether uncontrollable events should be performed automatically.

The first change introduces an option in the code generation process. One of the functions that the C-generator exposes, is a step function that attempts to execute all transitions that are currently possible. If a transition can be taken, the loop is stopped and all transitions are checked from the beginning. In this code generator, both controllable as well as uncontrollable events are executed. This means that an uncontrollable event blocks the execution of any following event, as the order in which these are executed is deterministic and fixed.

To prevent this, an additional option was introduced, which determines whether uncontrollable events should be performed automatically. The option can be passed via the graphical user interface (as in figure 5.5), or via the command line. For the controller generation process, the value of this option is false.

The other problem to the C-generator of CIF, was that it is not possible to execute transitions individually. For example, when data is received from a communication item, the corresponding transition should be executed. However, the generated controller code from CIF does not expose these functions, as they are marked as being static. In C, this means that they are scoped to the current file, and are not intended for external use. Although the simple solution would be to remove the static keyword, the actual solution is a bit different. The names of the functions that execute individual transitions are not named by their transition, but by a number. This number is internal to the CIF code generator, and is not exposed to the outside. Therefore, an additional function was added to the generated code. This function takes the name of a CIF transition (which comes from a C-enum that is exposed), and (attempts to) invoke the corresponding transition function, and returns the result.

5.3 ROS

One of the artifacts of the DSL is a ROS-node for a controller. The language has generators for both versions of ROS: ROS1 and ROS2. This node integrates with the CIF-engine and is responsible for all communication with the robotic middleware. The node defines fields for all communication and attaches listeners for the relevant communication. As soon as the node is started, a timer starts that will try to perform all transitions that the supervisory controller engine allows. When a transition can be executed, the relevant ROS communication will be triggered. The node also listens to events that are coming from the robotic middleware and performs the corresponding (uncontrollable) supervisory controller transitions which will update the states of all plants.

The nodes for ROS1 and ROS2 are both using C++ code, which allows the re-use of a lot of logic between the generators for the two versions. There are some differences in the ROS-interface and build configuration. These files are generated using platform-specific code.

5.3.1 Supervisory Controller

The most important part of the ROS-node is the integration with the synthesized supervisory controller. After the supervisory controller has been synthesised with a tool that supports Supervisory Control Theory as described in section 5.2, the supervisor can be integrated into the controller node. In this case, CIF is used as the tool to perform this synthesis. Although the CIF-engine generates C-code and not C++-code, the files can still be compiled and linked to the ROS-node via CMake [2]. The

[2] https://cmake.org

following section describes the technical connection to CIF.

The integration between the supervisory controller and the ROS-node is as follows. At each execution of the control loop (which is described more detail in section 5.3.1), the supervisory controller attempts to:

- Execute all possible controllable data transitions
- Execute the first controllable communication or tau transition that is enabled

Upon incoming data from the middleware, the supervisory controller will:

- Update the plant inputs with the correct data
- Execute the corresponding uncontrollable event

Initialization

The CIF-engine needs to be initialized in two ways. First, the generated C-code from CIF exposes the *AssignInputVariables* function, which is implemented in the ROS-node. Each of the inputs is connected to a variable that should have an initial value. The inputs are used to pass data from the middleware to the engine. The actual initial values are not relevant, as inputs are only used when data is received from the platform, and therefore immediately overwritten. Nevertheless, the initial value of a *boolean* input is false, for an *integer* it is 0 and for an *enum* it is the default transformation rule as modelled in the DSL.

The other phase of the initialization sets the internal CIF-state. The generated code from CIF does not expose a function that only performs this initialization. There is a function, *EngineFirstStep*, which attempts to execute the first event that is enabled for n times, where n defaults to 100 which can be changed using a build flag. Because the controller node performs the execution itself, rather than using the *EngineFirstStep*, the parameter can be set to 0, which transforms the function into an initialization function, as no events are executed.

Control loop

A timer, which forms the control loop, is invoked at every tick. In this control loop, the controller attempts perform transitions. It does not just try all transitions, but it splits up the transition into different sets.

At every timer tick, the controller node performs the following actions in the control loop:

- **Update the state of all data plants.** It does so by executing all the data plant events that are enabled. These controllable events update the state of the data plants that represent what should be sent to the middleware. The order in which these data events are executed is shuffled at every execution of the control loop. This means that, if multiple events are enabled (which can happen when provide statements have overlapping conditions), the data that will be sent to the middleware is non-deterministic.
- **Execute first enabled communication or tau transition.** All communication is modelled as a plant, which have trigger (and possibly cancel) transitions defined on them, see also section 2.5. If these transitions are fired, communication with the middleware should take place. There are also tau transitions, which can be defined in the behaviour automaton of a component. The controllable events for communication with the middleware are merged into a set together with all tau transitions. This set is shuffled at every execution of the control loop, after which the node will attempt to execute all transitions until it has found a transition that is enabled, which is then executed. If the transition that was executed is a trigger for communication with the middleware, this communication will take place. Note, that actions also have a controllable event which will cancel the action if fired.

In the case that the event represents a message to the middleware, a message will be published using a publisher. If it represents a service, then the service client will send a request to the service server. If the event is a trigger for an action, then a goal is sent to the action server. This goal is cancelled if the event that was executed represents a cancel event for that action.

Incoming events

There is also communication coming from the robotic middleware, that the supervisory controller should be made aware of. This could be in the form of messages, responses or action feedback. Upon receiving data, the controller first assigns the data to plant inputs, if they are required in the supervisory controller. Then, as soon as the inputs are set, the controller node invokes the supervisor transition.

It is also possible that the data is not directly used in the supervisory controller, meaning that it is not used in transition guard or requirement. If that is the case, the data is stored within the controller node itself, ready to be transmitted. An example of such a callback function with assignment can be found in listing 5.7.

```
void callback_message_correction(const std_msgs::msg::Float32::SharedPtr msg) {
  code_LineDetector_current_correction = msg->data;

  // Call engine function
  controller_EnginePerformEvent(message_correction_u_response_);
}
```

Listing 5.7: C++-code for message callback with code-only assignment.

Event hook

The CIF-code generator exposes functionality that allows for notifications when transitions are executed. This is done in form of the *InfoEvent* method, which receives the name of the transition together with a *boolean* that determines whether the function is invoked before or after the engine has performed the transition. Note that notifications before transitions are taking place already indicate that the transition is enabled.

As soon as the notification is received that a transition has taken place, the controller will check what it should do based on a switch statement. The trigger transition for messages, services and actions results in the call of a method which will start the corresponding communication. For actions, there is an additional switch case, which is for the cancel event. When the cancel event is executed, all goals for the corresponding action server are cancelled. An example for action events is shown in listing 5.8. The actual implementation of the call and cancel functions is explained in the section on communication.

```
switch (event) {
    case action_navigate_c_trigger_:
        node_controller->call_action_navigate();
        break;
    case action_navigate_c_cancel_:
        node_controller->cancel_action_navigate();
        break;
    default:
      return;
   }
```

Listing 5.8: Sample switch case for actions.

5.3.2 Communication

The generated code for communication with the middleware depends on both the type of communication (message, service or action) and the ROS-version. The concepts for both versions are similar, however.

Messages

The implementation of a message depends on the direction of that message. If a message is coming out of a component, the controller should create a subscriber for the message as it is an uncontrollable event. If the component has an incoming message, the controller should create a publisher which publishes to the same topic.

The name of the topic can be explicitly specified within the DSL. If that is not the case, then the name of the message will be used to subscribe to.

Next to the field for the message, the generator also decorates the controller with additional methods. If the controller subscribes to a message, an additional method is generated which is invoked as soon as a message is received from ROS. This function takes the incoming message, takes the data and assigns the data to the corresponding variables. For data that is required in the supervisory controller, inputs will be used. In all other cases, data is assigned to variables that are defined within the scope of the controller node. More information can be found in the section on data. The actual C++-type of the message is depending on the interface that is linked to the message if it is a non-basic data type. After the data has been assigned correctly, the supervisory controller is notified of the event. An example of such a function is shown in listing 5.7 (note that this is for ROS2).

A different method is generated when a controller should publish to a topic. Then, there is a function to trigger the communication. A new object is constructed that stores the data of the message. The actual data that is sent along is depending on the state of the data plants, as described in the section on data. When the message is constructed, it is published to the topic. Again, the C++-type of the message is determined by the linked interface for messages that are defined with a custom data type. Note that this function is never called directly, but only after the trigger transition has taken place in the supervisory controller. Listing 5.9 contains an example of such a function.

```
void call_message_move() {
  auto value = geometry_msgs::msg::Twist();

  if (data_move_ == _controller_data_one) {
    value.linear.x = 0.6;
    value.angular.z = (-code_LineDetector_current_correction) / 100;
  }

  this->publisher_client_move->publish(value);
}
```

Listing 5.9: C++-code for publishing a twist message

Services

A service client is generated for all services that are defined in the model. The name that is used for the client is determined similarly to the topic of a message. It can either be explicitly defined, or a fallback will be used. The fallback is the name of service within the DSL. The definition of a service consists of three parts. First, a client is created and stored in a field, allowing the controller node to access it when needed.

The second part of the service, is the invocation method. It allows the controller to send a request to the server. Again, the actual data that is put into the request is determined by the state of the corresponding data plants. Then, the request is sent to the service. For ROS2, the next step is to wait for the response. Because services are asynchronous within ROS2, the controller waits for a response in the form of a callback function. This callback function handles the data, assigns it to the correct variables and notifies the supervisory controller of the incoming data. The C++-types of the incoming and outgoing data are determined by the linked interface in the model.

The procedure for ROS1 is a bit different. In ROS1, services are synchronous calls, meaning that if the controller sends a request using the service client, it will wait until it has received an answer. So, for ROS1, the response and request handling are done within the same method. Because services are still asynchronous in the model, the method notifies the controller of the incoming response after it has processed the data. The difference can be seen in listing 5.10 (ROS1) and 5.11 (ROS2).

```
void call_service_sample() {
  default_msgs::SumAction srv;
  auto request = std::make_shared<default_msgs::SumAction::Request>();

  if (data_sample_ == _controller_data_one) {
    srv.request.a = 2;
    srv.request.b = 7;
  }

  if (service_client_sample.call(srv)) {
    code_SampleComponent_result = srv.response->sum;

    controller_EnginePerformEvent(service_sample_u_response_);
  } else {
    controllerEnginePerformEvent(service_sample_u_error_);
  }
}
```

Listing 5.10: C++-code for invoking a service in ROS1

```
void call_service_sample() {
  auto request = std::make_shared<default_msgs::srv::Sum::Request>();

  if (data_sample_ == _controller_data_one) {
    request->a = 2;
    request->b = 7;
  }

  using ServiceResponseFuture = rclcpp::Client<default_msgs::srv::Sum>::
      SharedFutureWithRequest;
  auto result = service_client_sample->async_send_request(request, std::bind(&Controller::
      response_service_sample, this, std::placeholders::_1));
}

void response_service_sample(rclcpp::Client<default_msgs::srv::Sum>::SharedFuture future) {
  std::shared_ptr<default_msgs::srv::Sum_Response> result = future.get();

  code_SampleComponent_result = result->sum;

  // Call engine function
  controller_EnginePerformEvent(service_sample_u_response_);
}
```

Listing 5.11: C++-code for invoking a service in ROS2

Actions

Similar to services, actions have a request (also known as goal) and a response. But next to that, actions have the option to provide feedback during the execution of that action. Actions are asynchronous in both ROS1 as well as ROS2.

Each action first gets an action client, where the name of the action either is explicitly defined in the model, or the entity name is used as a fallback. Four additional methods are generated next to the client. First, a method that can start the action and provide it with a goal. The C++-datatype of the goal is depending on the interface type of the action and the value of the goal is based on the data that is passed to it using the provide statements. The method to start the action first attempts to reach the action server, as it might be unavailable. If that is the case, the error transition is executed, allowing the user to be indicated about the error that occurred. Then, there is a callback for when feedback is received. This feedback is assigned to the corresponding variables, after which the supervisory controller is notified of the data. Then, when the action is finished, the action client receives a response using the generated response method. The data is processed, and the engine is made aware of the uncontrollable event. Lastly, there is also a cancel method that can be invoked to cancel the current goal, if there was any. An example of a generated function to start an action can be found in listing 5.12. Note that the example is for ROS2, but the code for ROS1 is very similar.

```
void call_action_navigate() {
  if (!this->action_client_navigate->wait_for_action_server(1s)) {
    controller_EnginePerformEvent(action_navigate_u_error_);
    return;
  }

  auto goal_msg = nav2_msgs::action::NavigateToPose::Goal();

  if (data_navigate_ == _controller_data_one) {
    goal_msg.pose.pose.position.x = code_Nav2_current_x;
    goal_msg.pose.pose.position.y = code_Nav2_current_y;
    goal_msg.pose.pose.position.z = code_Nav2_current_z;
  }

  auto send_options = rclcpp_action::Client<nav2_msgs::action::NavigateToPose>::
      SendGoalOptions();
  send_options.result_callback = std::bind(&Controller::response_action_navigate, this, std::
      placeholders::_1);
  send_options.feedback_callback = std::bind(&Controller::feedback_action_navigate, this, std
      ::placeholders::_1, std::placeholders::_2);
  this->action_client_navigate->async_send_goal(goal_msg, send_options);
}
```

Listing 5.12: C++-code for starting an action in ROS2

5.3.3 Data

Enums

The DSL defines the concept of an enum, which can transform an arbitrary data type into a finite set of values using transformation rules. If the value matches a given expression, the data will be transformed to that value. If none of the transformation rules match, a default rule will be used. The ROS-node converts values before they reach the supervisory controller

The generator generates a function that takes an input and uses multiple if-statements to determine the return value. The default return value will be the enum default. The return type of the function is *controllerEnum*, which comes from the CIF-generator, and stores all the enum values in one single data type. As a result, the actual return values are prefixed with the *controller* keyword, as they represent an internal CIF data type. Listing 5.13 demonstrates such a function, where the input of a laser scan is converted into a value that determines whether there is a wall, or there is not.

```
controllerEnum convert_enum_Distance(const sensor_msgs::msg::LaserScan::SharedPtr input) {
  if (input->ranges[270] < 0.7 || input->ranges[240] < 0.7) {
    return _controller_wall;
  }

  return _controller_no_wall;
}
```

Listing 5.13: C++-code for enum value transformation.

Code-only assignments

Some assignments that happen upon incoming data from the middleware, are done to variables that are not required in the supervisory controller. In this case, the assignment of such a variable is done in the node only. The generator defines C++ variables for each of them. For each function that handles incoming data from a response, message or feedback, the generator checks which transitions perform assignments and checks for each variable whether that is a C++ assignment. If it is a C++ assignment, the expression is compiled to C++ and assigned to the corresponding variable.

Language concept	ROS concept	Cardinality
Message	Message	One-to-one
Service	Service	One-to-one
Action	Action	One-to-one
Interface	Custom Interface	One-to-one
Component	Node	Many-to-many

Table 5.3: Mapping between language concepts and concepts from ROS.

Provisioning

As explained in chapter 4, data is provided to communication items using provide statements. Each of these statements are modelled using data plants, whose state determines the data that is sent along.

Data can be passed to messages (going out of the controller), services and actions. Before such communication takes place, the controller first determines the state of the corresponding data plant and links it with the actual data. This is done by generating if-statements for each the different states, and then assigning the values. If the data that should be sent is a basic type (*string*, *bool*, *integer* or *double*) then the property *data* will be used to store the value, as is the name of the built-in ROS-type.

If complex objects are used, all properties from the value are extracted and assigned individually. If the given value of a property contains an expression, then it the expression will first be compiled to C++-code.

5.3.4 Metadata

Both versions of ROS require some additional configuration that stores metadata about the package, and how to build it. First, there is the *CMakeLists.txt* file, which contains information about the build process. Most of the generated file is based on the ROS-defaults, but there are some changes. First, the file links all of the custom message definitions that the model uses. These packages are defined within the model, and added as a dependency to the build file. Furthermore, the file contains code that can compile the C-code for the supervisory controller. It also contains an important flag which ensures that the CIF-engine can be initialized safely, without resulting in any unexpected behaviour.

The other file that is generated stores information about the package, which is the *package.xml* file. The packages holds information about the build tool (which is different between ROS1 and ROS2), and about the required packages for custom message definitions. These packages should be present at build time.

5.3.5 Mapping

The DSL presented in this thesis introduces concepts from the ROS domain. All communication concepts (messages, services and actions) are coming from that domain. The same holds for an interface, it has a one-to-one mapping to the concept of a custom interface in ROS. The relation between a component from the DSL and a node is a bit more complex. Although users regularly will find that a one-to-one mapping works best, it is possible to have many components map to a single node and vice-versa. Therefore the mapping between the two concepts is a many-to-many mapping. An overview can be found in table 5.3.

5.3.6 Supervisory Layer

To support the supervisor node for the supervisory layer approach, as introduced in section 4.7, an additional generator was created which is based on the ROS2 code generator. Instead of generating a controller node, it now generates a supervisor node. It still uses a lot of common generator logic that is shared between all generators, but there are some differences. Note that the code for the supervisor is only generated for ROS2, and not for ROS1. This was done to show the possibilities of this layer approach. A generator for ROS1 could be implemented by applying the same changes and concepts.

The first addition to the generator is the generation of a launch file. Based on information supplied in the config file, the launch file stores logic to start the existing controller and the supervisor. Within the launch file, ROS offers the option to apply remappings. This means that unique names are generated for all of the communication that is defined within the DSL-model.

The supervisor node now creates a subscription and a publisher for each message. If the controller provides the supervisor with a message, the supervisor creates a subscription for the remapped topic and a publisher for the original one. If the controller expects a message, the publisher publishes to the remapped topic and the subscriber listens to the original topic.

If the supervisor receives a message from the controller, the data is first stored into a variable, and the trigger event is executed in the supervisor. If the supervisor allows the behaviour, the data is retrieved from the property and published to the original topic. Note that this process takes place within a mutex, which prevents the passing of incorrect data because fields are overwritten. Mutexes ensure that a critical part of the code is only accessed by one thread at a time. Messages to the controller also result in a transition in the supervisor, similar to the controller behaviour, and are passed to the existing controller using the remapped topic.

To allow the supervisor to control the communication of services, the supervisor creates a service server and client for all defined services. The service server listens to the remapped service name, and will handle all incoming communication from the controller. When a request is coming in, the supervisor checks whether the trigger is allowed. If that is the case, the request will be forwarded to the original service. As soon as a response is coming in, it is sent back to the controller.

Similar to services, actions also get a proxy server. Goals, responses and feedback are forwarded between the controller and the original action server, if the supervisor allows the behaviour. The problem for actions is that they are not remapped by ROS. This means that the bindings to the robotic middleware should be altered such that the existing controller connects to the proxy action server.

5.4 Generator Configuration

There are some settings that can be defined which will alter the generated code. This configuration is stored in a separate file, *controller-config.json*. This file can either be in the same directory as the DSL-code file that is compiled, or in one of the child directories starting from that location. The choice to use JSON for the configuration was made because it is easy to read, write and parse which makes it perfect for simple settings like this.

Within the config, users have the option to disable publishing state information to the middleware using the *publishStateInformation* option, as it takes more processing time in the control loop. The same holds for the logging of input and output, which can be enabled using the *writeEventsToLog* key. Furthermore, they have the option to copy the code of the controller (or supervisor) to a specific location. Normally, code that is generated is placed within the *src-gen* folder, requiring users to copy the generated controller to the correct location every time. By using the *output* key within the configuration, and setting one or more of the *ros1ControllerNodeLocation*, *ros2ControllerNodeLocation* or *ros2SupervisorNodeLocation* fields, the copying will take place automatically.

Lastly, there is an option for the supervisor which determines how to start the existing controller. This is required, because it will be started with remapped topics and services. There is an option to start the controller from an existing launch file, or running the node directly. These options can be configured using the *controller* property of the *supervisor* key.

Initially, the idea was to also use the configuration file for linking custom interface definitions from ROS, as they appeared not to be relevant to the actual model, only as a setting for generation. But, although for compilation this would have been enough, the actual imports and parameter types of these custom interface definitions, depend on the message, service or action that they are connected to. Therefore, they became part of the model, known as interfaces.

5.5 Debugging

The core idea of the language is that code is generated from the DSL script. This code can be deployed to a robotic middleware and will instantly run. This means that users can make use of debugging tools provided by the generated language (like C++ for ROS2). But, this can be cumbersome to set up and understand. It is also very likely that users have little to no knowledge about these generated languages.

In order to make it easier to debug the behaviour of the supervisory controller, a configurable option has been added that specifies whether the controller node should publish information about the current state of all components to a topic. Other nodes can subscribe to this topic, and handle the information.

The data that is published to the topic, is a serialized JSON string. Although a custom interface definition could have been used, JSON was chosen as it does not require any additional dependencies on the target platform. The current state is serialized to JSON and contains the following information:

- Components
- Component states
- Component transitions
- Component variables
- Variable values
- Executed transitions

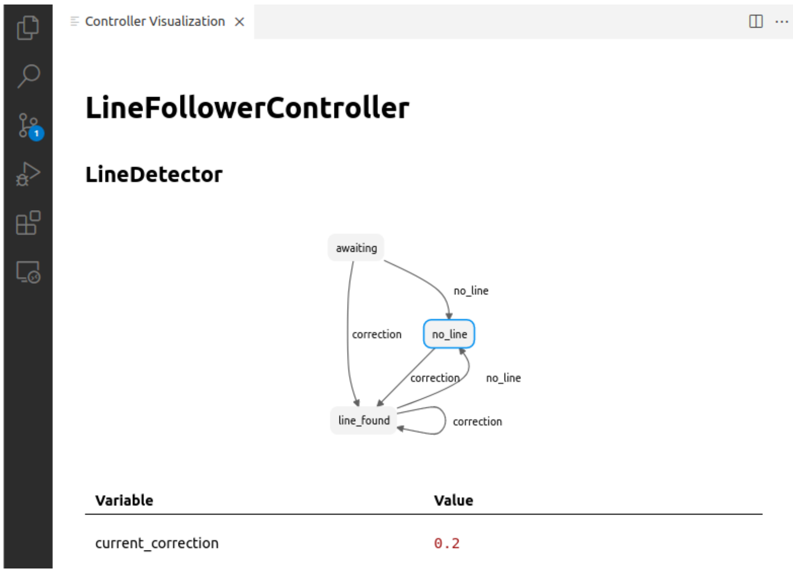
Although the serialized information already provides some information, it is still hard to read. That is why the Visual Studio Code extension has a feature that visualizes the information. The extension starts a node in the background and subscribes to the controller state topic. As soon as information comes in, it will either create a visual representation of the automata and variables, or update the existing visualization. If data from a new robot comes in and thus the definition of the robot changes, it will create a new visual representation. The extension was built using JavaScript and uses the ROS-packages from Foxglove [3] [4] to facilitate communication with the middleware, and Dagre D3 [5] for the visualization of the state machines. An example of such a visualization can be found in figure 5.6.

It is also possible to gather more information about the input and output of the controller using a text file. By default, no logging is generated. If a user configures logging (see section 5.4), the controller node writes all events to a file together with a timestamp, which can be used to reason about the behaviour that is or was executed by the controller node.

[3] https://www.npmjs.com/package/@foxglove/ros1

[4] https://www.npmjs.com/package/@foxglove/ros2

[5] https://www.npmjs.com/package/dagre-d3



LineFollowerController

LineDetector

Variable	Value
current_correction	0.2

Figure 5.6: Example of a visualization of the robot state in Visual Studio Code.

Chapter 6

Evaluation

6.1 Introduction

This chapter evaluates the language by developing several models with the DSL for a given set of scenarios, in order to evaluate expressiveness, behaviour, memory usage, compilation time, execution time and the amount of generated lines. All of the implementations of the scenarios were tested with a simulation tool, and a single scenario has been verified on multiple physical robotic hardware platforms as well. The chapter also highlights the limitations of the language, that came to light when implementing the different scenarios.

6.2 Evaluation Goals

In order to evaluate the language, several scenarios have been developed in order to test the expressiveness, flexibility, generated code size, compilation time and the execution time of the language. These scenarios have all been implemented with the DSL. In all scenarios, an additional component is added which acts as an emergency stop. The scenarios are defined as follows:

6.2.1 Scenario 1 - Line follower

The first scenario describes a robot that is following a yellow line. The yellow line is found by taking the camera of the robot as an input, whose camera feed is processed by a ROS-node that finds the center of the line. The offset of the line center to the center of the image is published as a correction value, which can be used by the controller to adjust the movement of the robot. The procedure that the robot should follow was inspired by the tutorial from Gaitech [1].

6.2.2 Scenario 2 - Simple navigation

This scenario describes a scenario where the robot should perform simple navigation in a room, with no obstacles. The robot receives prior knowledge about the room in the form of a map. This map is visualized with Rviz (a tool that can visualize robotic data [2]). Then, the robot waits for an initial pose. When the initial pose has been received, the robot waits for a point that it should navigate to. To perform the navigation, the ROS Nav2 library [31] is used. It is used in such a way that it can be used together with a supervisory controller.

[1] http://edu.gaitech.hk/turtlebot/line-follower.html

[2] https://wiki.ros.org/rviz

6.2.3 Scenario 3 - Obstacle navigation

Contrary to the simple navigation scenario, in this scenario the robot has to navigate to a point, but comes across obstacles that it needs to navigate around. It again has prior knowledge about the environment, visualized with Rviz [3]. The robot waits for an initial pose. As soon as the initial pose is determined, the robot waits until it receives a navigation goal. For navigation, the ROS Nav2 library [31] is used.

6.2.4 Scenario 4 - Object finder

The object finder scenarios describes a task where the robot is situated within a room, and will use its LiDAR-sensors and camera to find a stop sign. It will move until it reaches a wall, and based on the distance to the wall on either the left or right side choose a direction. When the image detection node finds a stop sign, it will stop. The Darknet ROS package is used for the detection of a stop sign (and other classes, which are ignored in this scenario) [5].

6.2.5 Scenario 5 - Maze solver

In this scenario, the robot attempts to escape a maze using the wall follower algorithm. The layout of the maze has been reconstructed based on the article from Yong [63]. It will use the LiDAR sensors to sense the environment, and will keep following walls on the right side of the robot until it eventually reaches the end. Similar to the object finder scenario in 6.3.5, an additional node will be used that can rotate the robot for a fixed amount of degrees.

6.2.6 Scenario 6 - Push ball into goal

In the following scenario, the goal for the robot is to find a red ball and push it into a green goal. The robot uses the camera feed to detect red and green surfaces, and move it into the correct direction. The assumption in this scenario is that the red ball is located between the robot and the goal, and that the goal is visible for the robot when it is faced towards the ball, so it can push the ball in the goal. When the ball (and the robot) have reached the goal, the robot will stop moving.

6.2.7 Scenario 7 - Person follower

This scenario represents a scene where the robot needs to find a person, and follow it, while keeping a safe distance. The detection of a person happens using the camera feed, which classifies objects within the feed and assigns a predicted class to it. For the image classification, the Yolox package is used [20]. The package publishes all found objects to a bounding box, with the positions in the image. The Lidar sensor detects whether there is a person in front.

6.2.8 Scenario 8 - Supervisor

The last scenario is somewhat different than the other ones. In this scenario, a supervisor is used. It uses the controller that was obtained from the line follower scenario in section 6.3.2, but without the emergency stop and Lidar sensor. The emergency stop and LiDAR-sensor are enforced using the supervisor. All communication from and to the controller passes the supervisor. If the emergency stop was activated, any movement is blocked by the supervisor.

[3] https://wiki.ros.org/rviz

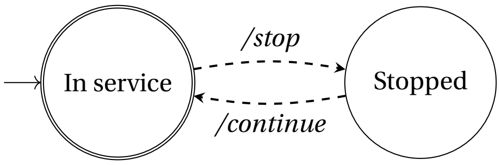


Figure 6.1: States of the emergency stop component.

6.3 Implementation

All scenarios have been developed for a Turtlebot 3 Waffle Pi [9]. For each scenario, a controller is developed using the language described in this thesis. All controller code has been added as an appendix.

6.3.1 Shared components

Most of the nodes re-use components using the concept of a library, as it allows the re-use of logic. For the scenarios, two shared components have been developed. One component models the emergency stop, and one component models the Turtlebot Platform.

The emergency stop, which is a middleware node that was purposely built for the evaluation of the language, provides the user with an interface to stop the robot. It has two states, one where the stop is in service, and one where it has been stopped. It can switch between the states using two messages, that determine whether it should stop or continue. This behaviour has been visualized in figure 6.1.

The other component, the TurtleBot Platform, only defines two messages, one message to move, and one message to halt all movement. It has defined the corresponding datatypes that allow the language user to specify angular and linear movement, but it has no associated behaviour.

6.3.2 Scenario 1 - Line follower

Line detection

For the line detection, an additional node has been developed for the purpose of this thesis. The node emits a value that provides the offset between the center of the yellow line and the center of the camera feed, in case a line has been detected. If a line has not been detected, it will emit a message to the robotic middleware that no line has been found.

The line detection makes use of the open-source OpenCV library [39]. First, the node that was developed transforms all colors that are yellow into white, and all colors that are not into black. The range of yellow is determined using HSV colors (hue, saturation, lightness) (information about HSV and its relation to RGB can be found in [2]). All colors in the image (from the camera feed) that lie within the HSV range of (10, 10, 10) and (255, 255, 250) are considered as yellow colors. Then, the OpenCV library is used to find the image's moments, whose center is used to determine the center of the yellow line (if present). Then, the center of this value is used together with the horizontal center of the image feed to determine the amount of correction that is needed for the robot to come back to the yellow line and follow it.

Model

The model is added in appendix B. In the model, the following four components can be distinguished: a line detector, a LiDAR sensor, an emergency stop and the Turtlebot Platform. Note, that the emergency stop and the Turtlebot Platform are imported from a library, as described in section 6.3.1.

The line detector component knows two states. First, the initial state is the state where no line is found. The other state is when the a line is detected, and a correction value is received. The component has two outgoing messages. First, when a line is detected, the component emits a message that contains a value that represents the offset from the line with respect to the center of the image feed, as

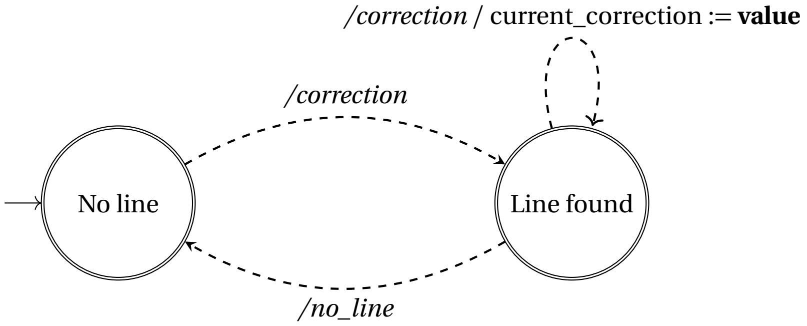


Figure 6.2: States of the line follower component.

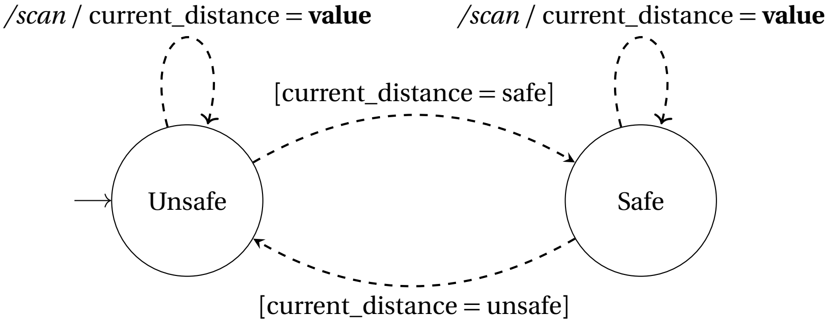

/scan / current_distance = value
/scan / current_distance = value
[current_distance = safe]
Unsafe
Safe
[current_distance = unsafe]

Figure 6.3: States of the LiDAR sensor component.

described in the section on line detection. This value is stored in a variable with the latest correction. When the controller receives a value from either the correction or no line message, it will transition to the corresponding state. The state machine of this component is depicted in figure 6.2.

The LiDAR sensor measures the distances on the front-, the left- and right-hand sides of the robot to determine whether it is safe for the robot to move forward. It therefore knows two states: one that represents a safe distance, and one that stands for an unsafe distance. It receives a message from the physical LiDAR scan (which contains a 360-degree scan) and transform it using an enum. This value is stored in a variable, and depending on the value of this variable it will transition to the corresponding state.

The requirements for the controller are relatively straight-forward. The Turtlebot Platform defines two messages: move and halt. For the controllable event move to be emitted, the following should hold:

- Component **Emergency Stop** should be in state **in service**
- **and** component **Line Detector** should be in state **line found**
- **and** component **LiDAR Sensor** should be in state **safe distance**

For the halt event, the following should hold:

- Component **Emergency Stop** should be in state **stopped**
- **or** component **Line Detector** should be in state **no line**
- **or** component **LiDAR Sensor** should be in state **unsafe distance**

The controller provides both incoming messages to the Turtlebot Platform with a linear and angular velocity. For the move message, this means that it will use a constant linear velocity, and an angular velocity determined by the correction that is received from the line detector: $\frac{-current\_correction}{100}$. The halt message should always stop all movement.

Communication

The communication, as seen from the component, is shown in table 6.1.

Direction	Type	Name	Data Type	Component
Outgoing	Message	/correction	double	Line Detector
Outgoing	Message	/no_line	none	Line Detector
Outgoing	Message	/stop	none	Emergency Stop
Outgoing	Message	/continue	none	Emergency Stop
Incoming	Message	/cmd_vel	Twist	Turtlebot Platform
Outgoing	Message	/scan	LaserScan	LiDAR scanner

Table 6.1: Communication overview of the line follower scenario.

6.3.3 Scenario 2 - Simple navigation

Navigation

As stated, the navigation happens using the Nav2 library [31] for ROS2. The library normally works without an additional controller, as one is provided by the library itself. An initial pose can be specified via RVIZ, and as soon as this has been done, RVIZ allows the user to specify a navigation goal. The problem for an additional controller is that this goal is directly passed to the Navigation node, and the action is started immediately. To solve this, the controller uses a different input. RVIZ also has a plugin that allows clicking a point, which is then published to a topic. Instead of using the navigation goal, the controller stores this point, and passes it to the navigation action itself.

To prevent confusion, the launch file for this scenario contains a custom RVIZ config. This config disables the navigation goal button, to ensure that users cannot accidentally get confused and use it to trigger the navigation, of which the controller is unaware.

The navigation action will be cancelled as soon as any requirements for it to be started do not hold anymore.

Model

The model is added in appendix C. The scenario is split into two components. First, there is a component that covers all Nav2 communication. Although it was possible to split this component into two components (for example a component for RVIZ, and a component for the navigation), it is combined into a single component because the communication of both components would be depending on each other. The RVIZ component should be aware of the fact that the navigation was cancelled, for example. Next to the Nav2 component, there is an emergency stop, which is imported from the shared library as described in section 6.3.1.

For the RVIZ component, three states can be defined. First, the component awaits an initial pose using the RVIZ interface. As soon as this pose is provided using the initial pose message, the component accepts incoming points for navigation. When a point is provided through the corresponding message, it transitions to a state where it has a point and the corresponding point is saved into a variable. As soon as the navigation is completed, or was cancelled it will transition back to the awaiting point state. A visualization of the states of this component can be found in figure 6.4.

There are only two requirements that are needed for the navigation action to start:

- Component **Emergency Stop** should be in state **in service**
- **and** component **Nav2** should be in state **has point**

Note, that if this requirement does not hold during the execution of the action, the action will be cancelled.

The navigation action is provided with a value that represents the last point that was published by the RVIZ component.

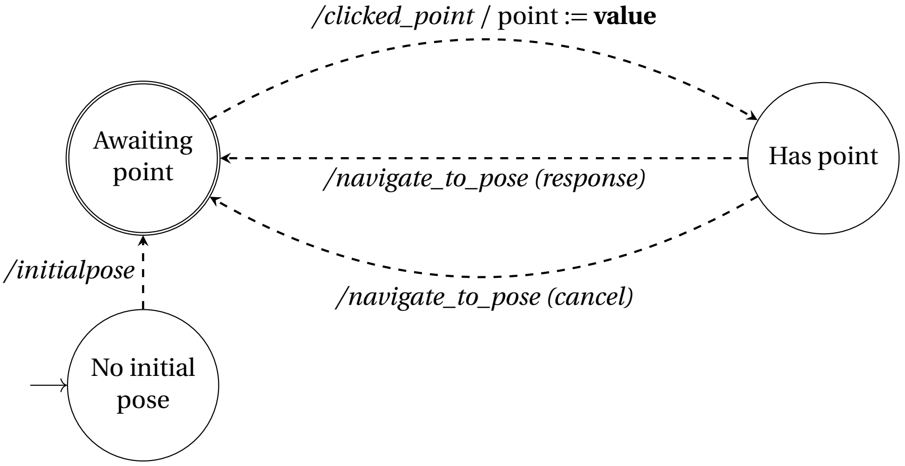

/clicked_point / point := value
Awaiting point
Has point
/navigate_to_pose (response)
/navigate_to_pose (cancel)
/initialpose
No initial pose

Figure 6.4: States of the Nav2 component.

Communication

The communication, as seen from the component, is shown in table 6.2.

Direction	Type	Name	Data Type	Component
Outgoing	Message	/clicked_point	PointStamped	RVIZ
Outgoing	Message	/initialpose	PoseWithCovarianceStamped	RVIZ
Outgoing	Message	/stop	none	Emergency Stop
Outgoing	Message	/continue	none	Emergency Stop
-	Action	/navigate_to_pose	NavigateToPose	Nav2

Table 6.2: Communication overview of the simple navigation scenario.

6.3.4 Scenario 3 - Obstacle navigation

The implementation of this scenario is exactly the same as the simple navigation scenario, because the obstacle avoidance happens using the Nav2 library. Therefore, the model, component and communication descriptions can be found in section 6.3.3. For completeness, the model for this scenario has been added in appendix D.

6.3.5 Scenario 4 - Object finder

Rotation

For this scenario, a node has been developed for the purpose of this thesis that can turn the robot a given amount of degrees. Normally, rotation of the TurtleBot platform happens using an angular velocity (in $\frac{\pi}{s}$). To allow the movement of a relative amount of degrees, this node starts a rotation and reads the current orientation of the robot. If the desired rotation has been reached, it will stop rotating. To increase accuracy, the robot is rotating at a relatively low speed, as it takes time to stop the robot. Increasing this speed is considered as out of scope for this project.

Model

The model for the scenario can be found in appendix E. In the DSL model, five components are defined. First, there is a component for the LiDAR scanner. Then, there are components for the rotator and the object detector. Lastly, there are two components that are used from the shared library,

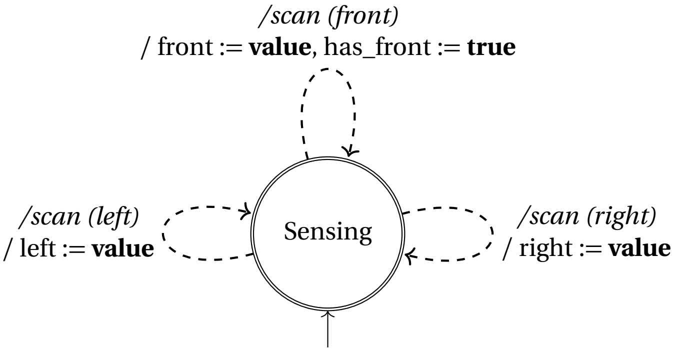


Figure 6.5: States of the LiDAR scanner component.

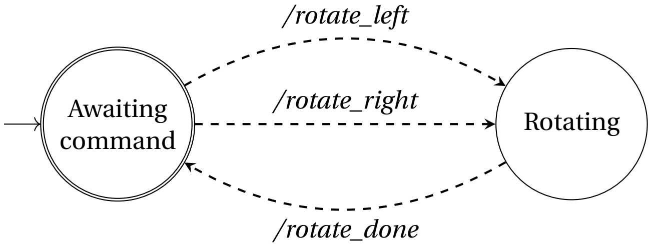


Figure 6.6: States of the rotator component.

namely the emergency stop and the Turtlebot Platform. They are described in the section on shared components (6.3.1).

The LiDAR scanner component knows a single state, the state where it is sensing. It has 3 outgoing messages, but they all have the same identifier and therefore the controller will subscribe to the same topic. However, their data type differs. Each of these three messages has an enum data type that transforms the 360-degree LiDAR scan to a safe or unsafe value, depending on the orientation (left, front and right). When one of these values is received, it will be transformed by the enum and stored into a variable on the component. Next to that, the component has a variable that determines whether a value for the front sensor has been received. This value will immediately be set to true as soon as this message reaches the controller. The states are visualized in figure 6.5.

For the rotator component, the behaviour is modelled with two states. There is a state where it is awaiting a command, and one where it is processing a command, i.e. rotating. The component has two incoming messages, one to rotate 90 degrees left and one to rotate 90 degrees to the right. Then, when this process has finished, it will send an outgoing message to notify the controller that it finished rotating. This behaviour is visualized in figure 6.6.

Then, the object detector component. It represents the Darknet ROS nodes [5] and has two outgoing messages: one message with the amount of objects it has detected in the camera feed, and one with the bounding boxes of the detected objects. The values of these messages (more specifically, the object count and the fond object) are stored in a variable in the component when the message is received. It will transition between a state where no object is found and a state where an object is found, depending on these values. The state machine is shown in figure 6.7.

The requirements for the controllable communication that should hold, are split per communication item. For the move message, this means that:

- Component **Emergency Stop** should be in state **in service**
- **and** component **Rotator** should be in state **awaiting command**
- **and** variable **front** from **LiDAR Scanner** should have value **safe**

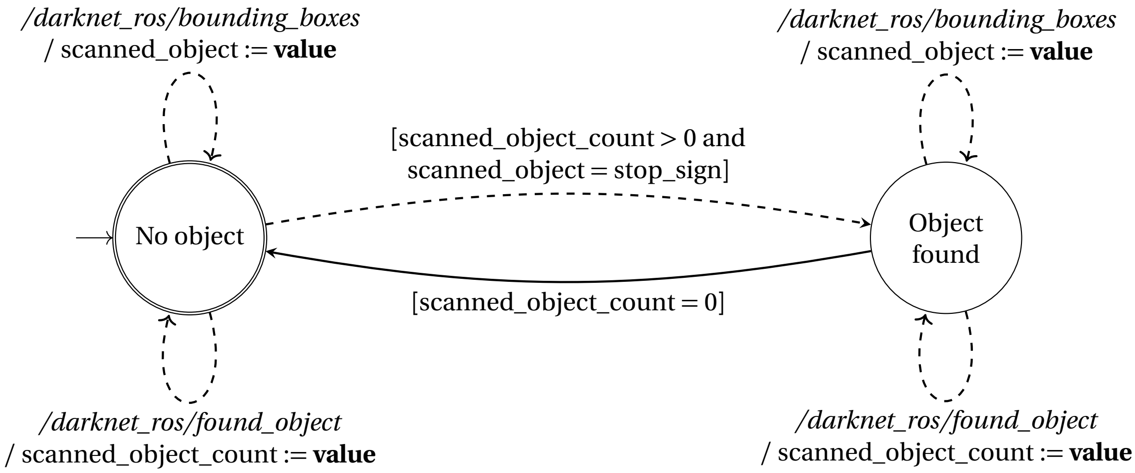

/darknet_ros/bounding_boxes
/ scanned_object := value
/darknet_ros/bounding_boxes
/ scanned_object := value
[scanned_object_count > 0 and
scanned_object = stop_sign]
No object
Object
found
[scanned_object_count = 0]
/darknet_ros/found_object
/ scanned_object_count := value
/darknet_ros/found_object
/ scanned_object_count := value

Figure 6.7: States of the object detector component.

- **and** component **Object Detector** should **not** be in state **object found**

Then, for the halt message:

- (variable **front** from **LiDAR Scanner** should have value **unsafe and** variable **left** from **LiDAR Scanner** should have value **unsafe and** variable **right** from **LiDAR Scanner** should have value **unsafe**)
- **or** component **Emergency Stop** should be in state **stopped**
- **or** component **Object Detector** should be in state **object found**

The requirements for the rotate left message are defined as below. Note, that for the rotate right message they are similar, except where the left variable is used, it should hold for the right variable of the LiDAR scanner.

- Component **Emergency Stop** should be in state **in service**
- **and** variable **left** from **LiDAR Scanner** should have value **safe**
- **and** component **Object Detector** should **not** be in state **object found**
- **and** component **Rotator** should be in state **awaiting command**
- **and** variable **front** from **LiDAR Scanner** should have value **unsafe**
- **and** variable **has front** from **LiDAR Scanner** should have value **true**

Communication

The communication, as seen from the component, is shown in table 6.3.

6.3.6 Scenario 5 - Maze solver

Wall-follower algorithm

For the algorithm to solve the maze, we use a relatively simple algorithm [8]. It is known as the wall-follower algorithm. In the basis, it means that the robot will move along the wall. The robot can either follow the wall on its left, or the wall on the right, depending on the implementation of the algorithm. In this scenario, the robot follows the wall on the right side.

Direction	Type	Name	Data Type	Component
Outgoing	Message	/scan	LaserScan	LiDARScanner
Incoming	Message	/rotate_left	none	Rotator
Incoming	Message	/rotate_right	none	Rotator
Outgoing	Message	/rotate_done	none	Rotator
Outgoing	Message	/darknet_ros/found_object	ObjectCount	Darknet
Outgoing	Message	/darknet_ros/bounding_boxes	BoundingBoxes	Darknet
Outgoing	Message	/stop	none	Emergency Stop
Outgoing	Message	/continue	none	Emergency Stop
Incoming	Message	/cmd_vel	Twist	Turtlebot Platform

Table 6.3: Communication overview of the object finder scenario.

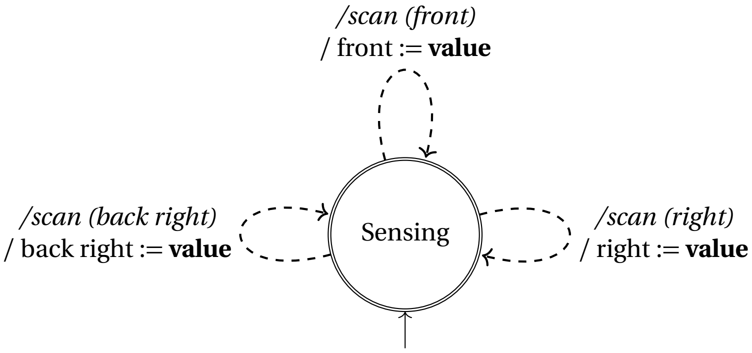


Figure 6.8: States of the LiDAR scanner component.

In the basis, the robot will move as long as it has found a wall on the right side. If it detects a wall in front, it will rotate right. If it detects that there is no more wall on the right side at −90°, but it does still find one at −105°, it will rotate right in order for it to keep following the wall. This process will repeat, until the robot has found the exit of maze. In this algorithm, the robot has no prior knowledge of the layout of the maze. Furthermore, all walls should be connected to the outer room.

Model

The model is added in appendix F. In this scenario, the problem has been modelled with 3 components. First, there is a component for the LiDAR scanner, which stores all distance values for the front and right side of the robot. Then, there is a component that represents the platform. The platform can move the robot forward, but also rotate it. Lastly, the emergency stop component is imported from the shared components library 6.3.1.

The LiDAR scanner component only knows a single state, the state where it is sensing values. The incoming scan data is modelled with three messages, as the data is transformed using three enums. One for the distance to an object for the front of the robot (0°), one for the right (−90°) and one for the back right at −105°. When a value is published, the component transforms it and stores it as a variable. Each of the enums transforms the distance value to a value that determines whether there is a wall, or there is not based on a threshold. The behaviour is represented in figure 6.8.

The other component is the platform component. It defines two messages that represent the movement of the robot (move and halt). Next to that, there are two messages, rotate left and right, that allow the robot to rotate in either direction. When this has finished, the component emits a message, rotate done, that indicates that the rotation has taken place. This behaviour results in two states: a state where the robot can move freely, and one where the the robot is turning and therefore can not accept any new rotation commands. These states and their transitions are shown in figure 6.6.

Requirements have been specified for the controllable communication in order for it to adhere to the line follower algorithm. The move command has the following requirements that should hold:

- Component **Emergency Stop** should be in state **in service**
- **and** component **Platform** should be in state **ready**
- **and** (variable **right** from **LiDAR Scanner** should have value **wall and** variable **front** from **LiDAR Scanner** should have value **no wall**)
  or (variable **front** from **LiDAR Scanner** should have value **no wall and** variable **right** from **LiDAR Scanner** should have value **no wall and** variable **back right** from **LiDAR Scanner** should have value **no wall**)

For the rotate right command, the following set of requirements should be satisfied for it to be triggered by the controller:

- Component **Emergency Stop** should be in state **in service**
- **and** component **Platform** should be in state **ready**
- variable **right** from **LiDAR Scanner** should have value **no wall**
- variable **back right** from **LiDAR Scanner** should have value **wall**

Then, for the robot to turn to the left, the following conditions should be satisfied:

- Component **Emergency Stop** should be in state **in service**
- **and** component **Platform** should be in state **ready**
- variable **front** from **LiDAR Scanner** should have value **wall**

The move and halt messages are provided with a fixed value that determines their speed, which for the halt command results in a speed of zero. Furthermore, the rotate messages are provided with a value of 90° that represents the amount of degrees the robot should rotate when invoked.

Communication

The communication, as seen from the component, is shown in table 6.4.

Direction	Type	Name	Data Type	Component
Outgoing	Message	/scan	LaserScan	LiDAR scanner
Incoming	Message	/rotate_left	integer	Platform
Incoming	Message	/rotate_right	integer	Platform
Outgoing	Message	/rotate_done	none	Platform
Incoming	Message	/cmd_vel	Twist	Platform
Outgoing	Message	/stop	none	Emergency Stop
Outgoing	Message	/continue	none	Emergency Stop

Table 6.4: Communication overview of the maze solver scenario.

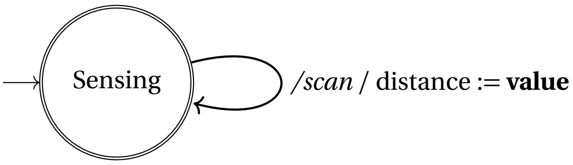


Figure 6.9: States of the LiDAR scanner component.

6.3.7 Scenario 6 - Push ball into goal

Goal and ball detection

For the goal and the ball detection, a similar process was developed for the purpose of this thesis as with the (yellow) line follower scenario from section 6.3.2. In this case, two separate nodes have been developed that use OpenCV for the processing of the image feed. These nodes publish a correction when the goal and/or ball have been detected, and publish to a different topic when no item was found.

The node for the detection of the ball, however, has some additional functionalities. When the ball is detected, but the center of it is too far to the left or right, it will publish that the ball needs an adjustment, in order for it to be in front of the robot again, and keep the robot from losing control over the ball. When this is not the case, it will publish that to a topic as well.

Lastly, the node for the ball detection also publishes a message when the ball is in front of the robot. Because LiDAR sensors do not work when the ball is blocking the lasers, it will use the camera feed to detect the presence of the ball by determining the ratio between the size of the ball and the image feed width. When this exceeds a predetermined threshold, it will let the controller know using a topic.

Model

The model for the scenario can be found in appendix G. The controller knows five different components. First, there is a component that represents the LiDAR scanner, so the robot knows when it has reached the goal. Then, there are two components that respectively model the ball and goal detectors. Lastly, two components are used that originate from the shared library, namely the emergency stop and Turtlebot Platform component. More information on them, can be found in section 6.3.1.

The LiDAR scanner component is relatively simple. It publishes a message that contains the result of a 360°-scan with all the corresponding distances. Using an enum, this value is transformed into a value that determines whether the path of the robot is obstructed, or free. At each scan, the result of the scan is saved into a variable in the automaton. The visualization of this component can be found in figure 6.9.

The behaviour of the goal detector component can be split into three states. The component publishes two messages, when a green goal was detected it will publish the offset of the center of the goal to the center of the image feed. When no goal was found at all, this will be published to a separate topic. The initial state of the component is where it is awaiting a result. Then, there are states where a goal was found, or when no goal was found. When a goal was found, the last correction will be stored in a variable. All states are shown in figure 6.10.

The ball detector is a bit more complex. As stated in the section about the ball and goal detection, the component defines multiple messages. First, it publishes whether there is a ball, and if so, the correction value. Furthermore, there are messages that define whether the ball is correctly in front of the robot, or whether the ball is too much to the left or right of the robot, causing the robot to lose control. Initially, the component waits for a result. Then, there are two states that represent whether the ball is found or not. Furthermore, there is a state when the ball is in front of the robot, and lastly, there is a state where the robot needs to adjust is movement to center the ball in front. All the states and transitions are shown in figure 6.11.

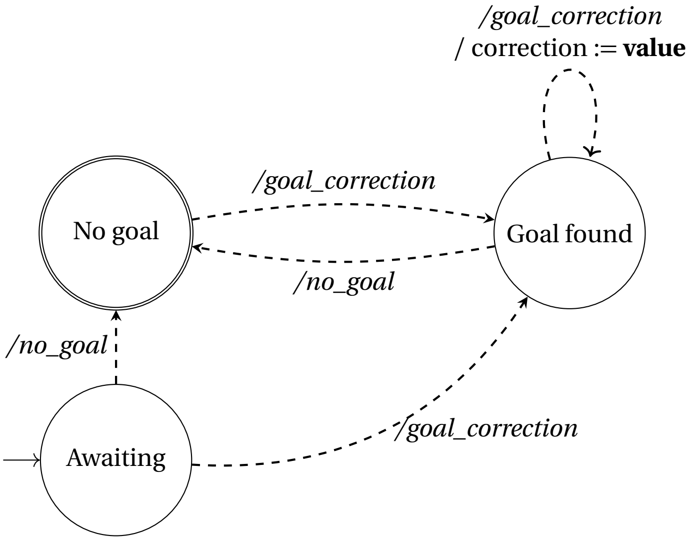


Figure 6.10: States of the goal detector component.

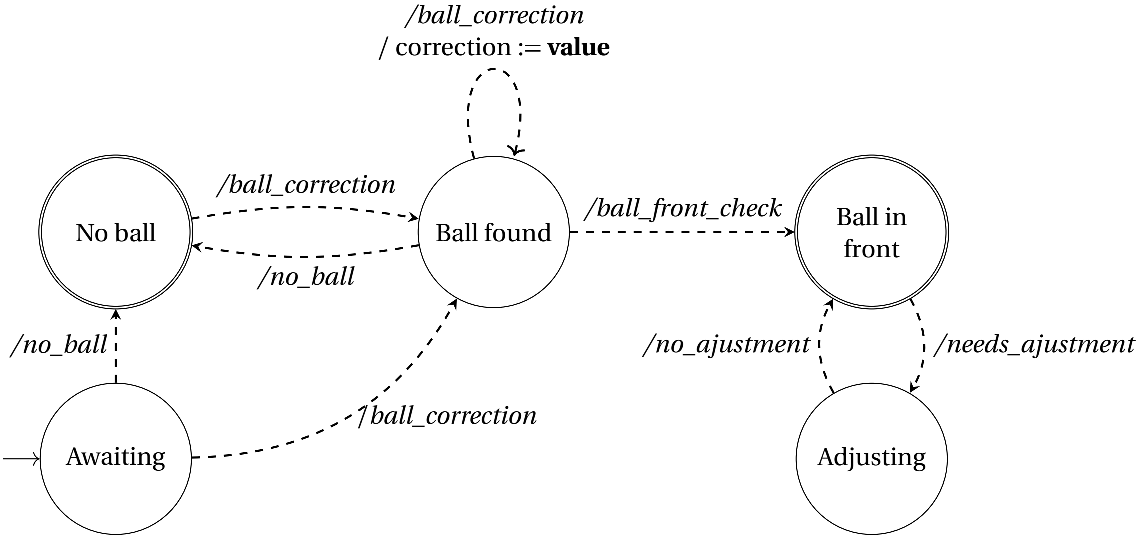


Figure 6.11: States of the ball detector component.

There are only two controllable communication items, move and halt. For move, the following should hold:

- Component **Emergency Stop** should be in state **in service**
- **and** variable **distance** from **LiDAR Scanner** should have value **free**

Then, for the halt message, the following requirements should be satisfied in order for it to be emitted:

- Component **Emergency Stop** should be in state **stopped**
- **or** variable **distance** from **LiDAR Scanner** should have value **obstructed**

Lastly, the model defines what values the controller should send along with the messages, depending on the current state. When no ball is found, the robot will rotate. When a ball is found, it will move towards that ball, until at some point it has the ball in front. It will then move towards the goal, until it needs to adjust the ball to keep it in front.

Communication

The communication, as seen from the component, is shown in table 6.5.

Direction	**Type**	**Name**	**Data Type**	**Component**
Outgoing	Message	/scan	LaserScan	LiDAR Scanner
Outgoing	Message	/ball_correction	double	Ball Detector
Outgoing	Message	/no_ball	none	Ball Detector
Outgoing	Message	/needs_adjustment	none	Ball Detector
Outgoing	Message	/no_adjustment	none	Ball Detector
Outgoing	Message	/ball_front_check	none	Ball Detector
Outgoing	Message	/goal_correction	double	Goal Detector
Outgoing	Message	/no_goal	none	Goal Detector
Outgoing	Message	/stop	none	Emergency Stop
Outgoing	Message	/continue	none	Emergency Stop
Incoming	Message	/cmd_vel	Twist	Turtlebot Platform

Table 6.5: Communication overview of the push a ball into a goal scenario.

6.3.8 Scenario 7 - Person follower

Model

The model for the scenario can be found in appendix H. The controller model uses four components. There is a component for the LiDAR scanner, one for the Yolox detector and two components from the shared library: an emergency stop and the TurtleBot movement platform. These components are described in the shared components section, section 6.3.1.

The LiDAR scanner represents the node that publishes the 360°-laser scan. It only has a single state, which processes the scan message. This message is transformed into a value that determines whether the robot can move freely, or that something is in front. The states are shown in figure 6.9.

The other component, the Yolox detection component, knows two states. First, there is a state where an object is detected. If this is the case, the position of the object is stored in the variables, next to the size of the image. This is done, to rotate the robot such that the object is in the center of the robot. The behaviour of the component is shown in figure 6.12.

Requirements are specified for two messages, namely for halt and for the move command. For the move command, the following requirements should hold:

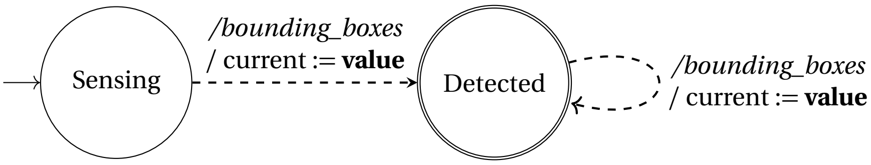


Figure 6.12: States of the Yolox detection component.

- Component **Emergency Stop** should be in state **in service**
- **and Yolox** should be in state **detected**

Similarly, for the halt message:

- Component **Emergency Stop** should be in state **stopped**
- **or** variable **distance** from **LiDAR Scanner** should have value **obstructed**

The provide statements ensure that the robot is moving at the correct speed. For the halt command, it is simple, as it should stop all linear and angular movement. For the move command, it depends on the distance to a person. When a person is in front, it should only rotate towards the person, but not move any further. If the robot is too far away, it will come closer to the person that it has found.

Communication

The communication, as seen from the component, is shown in table 6.6.

Direction	Type	Name	Data Type	Component
Outgoing	Message	/scan	LaserScan	LiDAR Scanner
Outgoing	Message	/bounding_boxes	BoundingBoxes	Yolox
Outgoing	Message	/stop	none	Emergency Stop
Outgoing	Message	/continue	none	Emergency Stop
Incoming	Message	/cmd_vel	Twist	Turtlebot Platform

Table 6.6: Communication overview of the person follower scenario.

6.3.9 Scenario 8 - Supervisor

Movement

The movement in this scenario is a little bit different to the line follower scenario. The line follower published an angular or linear velocity to a topic, which would make the robot move this way until a new velocity was received. To allow for the use of a supervisor, a node has been developed for the purpose of this thesis that can accept steps of movement. It supports the same inputs as the regular TurtleBot command, but it will only execute it for 100ms, after the robot will stop again.

By doing so, the supervisor can intercept move commands and check whether it allows it. If not, it can drop the command, which will happen in case the emergency stop was pressed or something is in front of the robot.

Model

The model for the supervisor is very similar to the line follower controller, and is added in appendix I.

Communication

The communication, as seen from the component, is shown in table 6.7.

Direction	Type	Name	Data Type	Component
Outgoing	Message	/correction	double	Line Detector
Outgoing	Message	/correction	double	Line Detector
Outgoing	Message	/no_line	none	Line Detector
Outgoing	Message	/stop	none	Emergency Stop
Outgoing	Message	/continue	none	Emergency Stop
Incoming	Message	/simple_movement	Twist	Simple Movement

Table 6.7: Communication overview of supervisor follower scenario.

6.4 Results

All of the scenarios have been evaluated using different metrics. For each of the scenarios, the amount of lines have been counted for the model and the generated CIF, ROS1 and ROS2 code. Furthermore, the compilation time for the ROS-nodes has been measured, next to the (peak) memory usage and the execution time. All experiments have been conducted on a Dell XPS 13" laptop running Ubuntu 20.04.4 (LTS). The computer has 16GB of RAM memory available and runs on an Intel® Core™ i7-10510U CPU @1.80GHz processor. For ROS1, the Noetic distribution was used, whereas for ROS2 the Foxy distribution was installed on the operating system.

6.4.1 Memory usage

First, the memory usage of each of the nodes was captured by using *psrecord* [4], a Python-tool that can record the memory of a process (and its children) for a period of time. The controller node was started directly on the laptop together with the simulation tool. During the evaluation, the nodes are all started separately from the controller (or supervisor). This was done to make sure that their memory usage does not influence that of the controller measurements. The controller is started using the *psrecord* tool, and measures the memory during a period of 5 minutes (300 seconds). Both the real memory usage, as well as the virtual memory usage have been captured in figures 6.13 and 6.14 respectively. The real memory represents the amount of memory that is currently within the physical RAM of the device, whereas the virtual memory is used by an operating system to create the illusion of a larger physical memory, when that is not the case, using an abstraction layer as explained in [4].

The first observation to make for both the virtual memory as well as the real memory, is that the line stabilizes after about 2 seconds, no matter what scenario. This suggests that, at least for the duration of the measurement and the code that was covered during the execution, no memory leaks appear to be present within the code. Furthermore, the simple navigation as well as the obstacle navigation scenario use a bit more memory than the other scenarios. Both scenarios make use of an action client within the ROS-node, whereas the others do not. The memory usage of the other scenarios is about the same in both figures.

6.4.2 Execution time

The execution times of a single execution of the control loop (in which it updates the state) have also been measured. The results are shown in table 6.8. During a period of 5 minutes, the duration of a control loop execution was measured (at each timer tick). Note, that the supervisor scenario was not measured, because the supervisor engine is only triggered based on emitted controllable events by an existing controller. Measuring this time would have skewed the measurements.

The results in the figure show that the line follower, person follower and the push ball into goal take the most time to execute. These scenarios all emit the move message that provides the robot with a velocity at (almost) every tick, which takes some time until it is published to the ROS message bus. However, the simple navigation and obstacle navigation take almost no time to execute during a

[4] https://github.com/astrofrog/psrecord

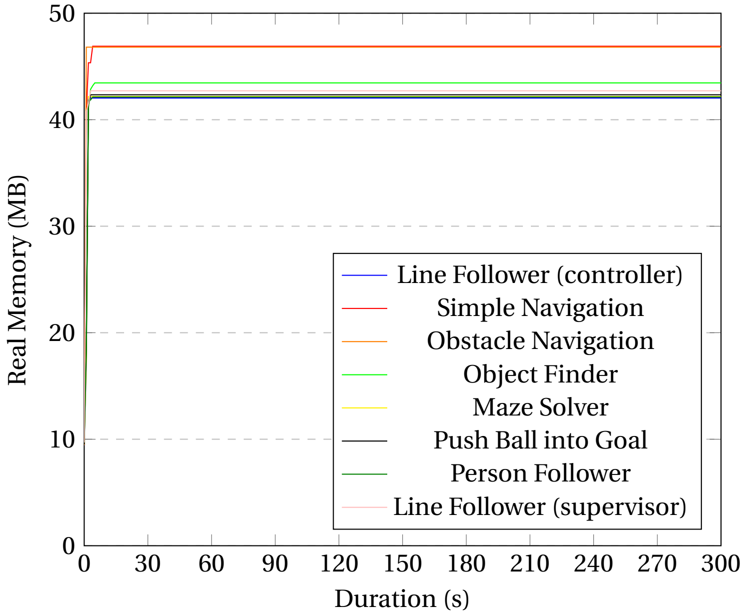


Figure 6.13: Real memory usage in megabytes for each of the scenarios.

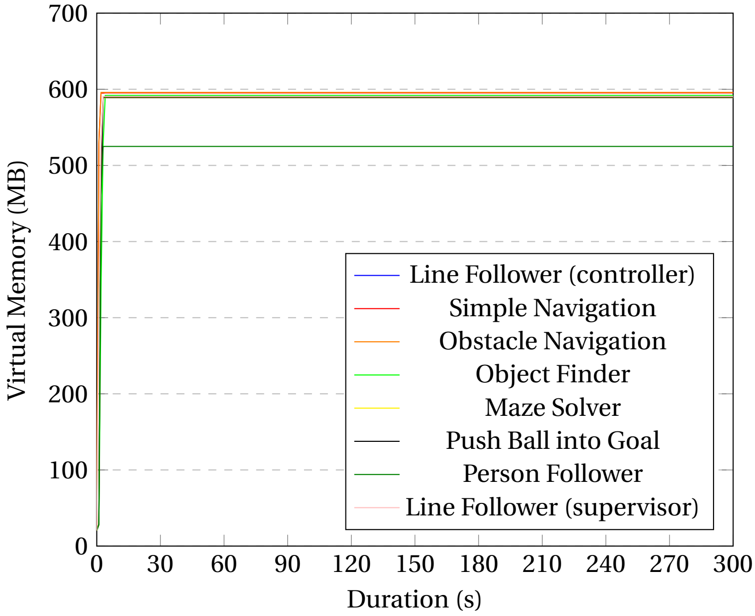


Figure 6.14: Virtual memory usage in megabytes for each of the scenarios.

	Avg. (μs)	**Min. (μs)**	**Max. (μs)**	**S. D. (μs)**
Line Follower	71	16	868	46
Simple Navigation	9	4	248	6
Obstacle Navigation	11	5	906	19
Object Finder	33	8	340	37
Maze Solver	36	8	451	42
Push Ball into Goal	42	15	528	26
Person Follower	52	6	1606	52

Table 6.8: Overview of the execution time in micro seconds of a single control loop for each of the scenarios.

controller tick, as the action is initiated by the controller, after which all movement is handled by the Nav2-package. The maze scenario and object finder also remain idle when the robot is rotating, and therefore take less time on average.

6.4.3 Compilation time

The compilation times of the controllers (and supervisor) for all scenarios have been plotted in figure 6.15. They have all been build in a clean (without cache) environment using the default build tool for the robotic middleware. For ROS1, this means that the nodes were built using *Catkin* [5], whereas *Colcon* [6] was used for ROS2. Only the time that the controller took to build has been measured, any other dependencies were ignored. Note that the two navigation scenarios make use of Nav2, which is a package that is available for ROS2 only. To ensure that the controller of these scenarios could compile, the action and message definitions of the Nav2-package were manually ported and compiled for ROS1, such that they could be used in the compilation process of the controller. Obviously, it was not possible to run these controllers on ROS1. Note that the supervised version of the line follower only contains information for ROS2, as the supervisor only has a ROS2 version.

From the results, it can be noted that the difference between the compilation time for ROS1 and the compilation time for ROS2 is relatively big. In some scenarios, compilation for ROS2 takes almost two times as long. For ROS1, the compilation of the navigation scenarios takes a bit longer than the other ones. These are the only scenarios that make use of the action library (that provides tools for action clients and servers). For the other scenarios, there are no notable differences in compilation times.

6.4.4 Source Lines of Code

Lastly, the source lines of code for all of the scenarios have been counted using *sloc*[7]. Rather than just counting the amount of lines within a directory, the tool strips white lines and comments, leaving only lines with actual code. This has been measured for the source DSL code and the generated CIF, ROS1 and ROS2 files. Note, that the code of the ROS1 and ROS2 node includes code that was generated based on the CIF file. The results are shown in the bar chart in figure 6.16.

The ratio between the amount of lines of the DSL and the amount of generated CIF lines is about 2-3, depending on the scenario. For ROS nodes, the ratio between the DSL and the ROS-node is way bigger (22 on average). Again, the exact values depend on the scenario. It can be observed that the difference in source code lines between ROS1 and ROS2 nodes is very small. Note that the supervisor does not have a ROS1 implementation, which is therefore missing from the results.

[5] https://wiki.ros.org/catkin

[6] https://docs.ros.org/en/foxy/Tutorials/Colcon-Tutorial.html

[7] https://www.npmjs.com/package/sloc

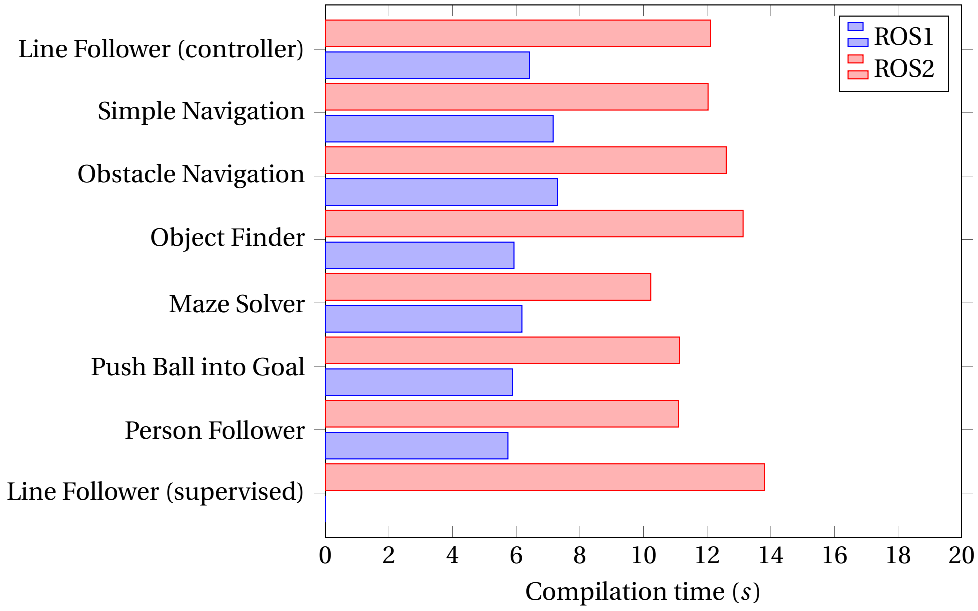


Figure 6.15: Average compilation time for ROS1 and ROS2 nodes.

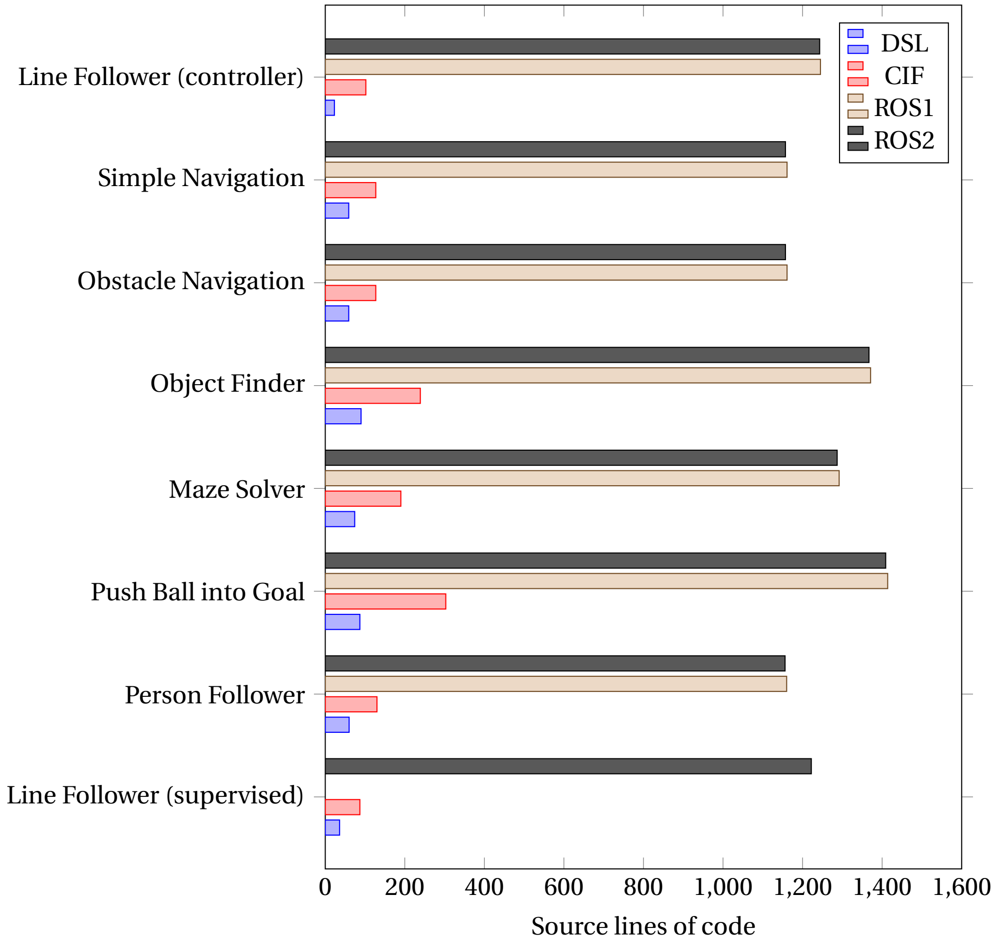


Figure 6.16: Source lines of code for DSL and generated code for all of the scenarios.

6.5 Simulation

Each of the scenarios has been simulated using the Gazebo simulation tool [8]. Gazebo is an open-source 3D simulator to verify the behaviour of a robot using a simulator. Although the default Gazebo simulator can be used without ROS, packages exist to allow integration between ROS and Gazebo [41].

All simulations were executed with the Turtlebot 3 Waffle Pi robot [9], a compact, modular and customizable robot that can be used to develop robotic applications in many different fields. The robot contains two individually controllable wheels, a Lidar sensor and a camera. It supports both ROS1 as well as ROS2 by running it on a Raspberry Pi, a small single-board computer. The simulations were done by using ROS2. Gazebo integration is available using publicly available ROS-packages[10].

Each of the scenarios requires multiple nodes to be started. To support this, each of the nodes has been equipped with a package *scenario* that contains a launch file, which will start the Gazebo simulation, all the scenario nodes, an emergency stop and any additional required tools like Rviz.

All scenarios were ran individually and each of them executed their goal correctly in the end. Some of them required minor adjustments or bug fixes to the language to achieve their goal. Screenshots of the simulations of all eight scenarios can be found in figure 6.17.

In the first scenario (figure 6.17a), the line follower, one can observe that the line follower follows the yellow line until it no longer finds a line. If the emergency stop is pressed in between, the robot stops moving. The same holds for when an object is placed in front of the robot. It will temporarily stop the line detection process. It does slightly rotate away from the line when it stops, as it needs stop all angular velocity.

In the second scenario (figure 6.17b), the simple navigation scenario, Rviz is launched next to the simulator and emergency stop. It is preloaded with a map of the room that the robot is in (within Gazebo). After determining the initial pose within Rviz, the robot is ready to accept navigation commands. After publishing a point, it correctly navigates to the point. Pressing the emergency stop results in the robot cancelling the navigation. After the emergency stop is back in service, it is ready to accept new commands.

The third scenario (figure 6.17c) is very similar to the simple navigation scenario. In this case however, both Rviz as well as Gazebo have been loaded with a default Turtlebot 3 simulation map which contains obstacles. Executing the same steps as for the simple navigation scenario results in the robot correctly navigating around the obstacle and reaching the provided destination point.

Starting the fourth scenario (figure 6.17d), which is the obstacle detection scenario, starts a visualization of the camera, next to the simulation environment and emergency stop. This camera view is annotated with any objects that the Darknet ROS package classifies. As soon as the scenario starts, the robot is moving along the wall and arbitrarily picks a direction. At some point, it will reach the stop sign and stop the robot. One observation to make is that rotating the robot for exactly 90 degrees takes relatively long.

The fifth scenario (figure 6.17e) demonstrates a maze solver. The maze has been implemented by adding walls to the scenario. The robot has no prior knowledge about the environment. Once started, the robot starts to move along the wall and follow it on the right-hand side. Because the algorithm does not find the shortest path to the exit (because the robot has no knowledge about the maze itself), and rotating takes some time, it takes about a while before the robot has reached the end, but eventually it finds the exit. It will then move around the field and start over.

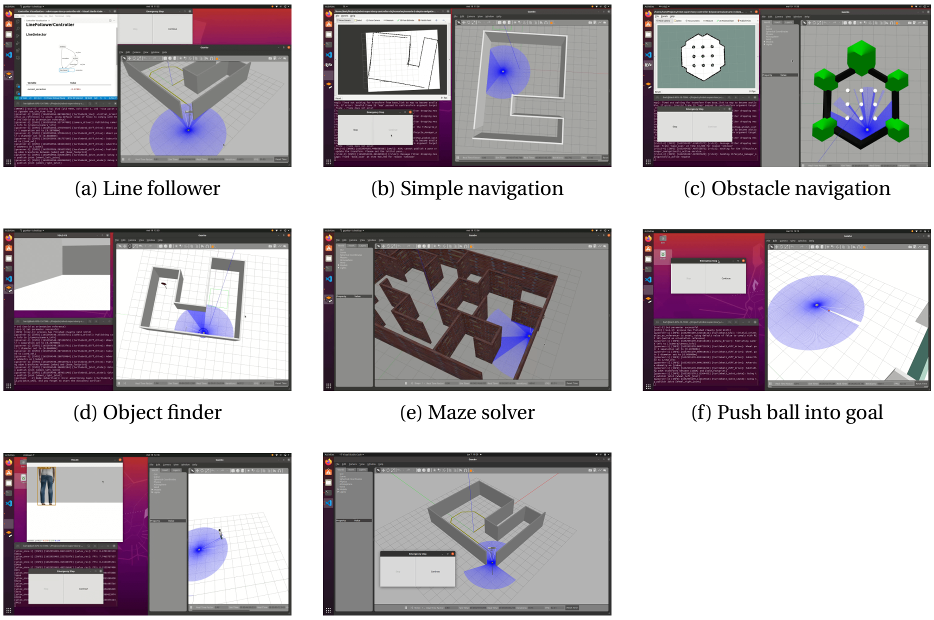
The sixth scenario (figure 6.17g) lets the robot push a ball into a goal. By starting the launch file, the goal and ball detectors are launched as well. Once started, the robot correctly finds the ball and attempts to move it to the goal. If it is too far off the center of the robot, it corrects it. Once the ball is in the goal, one can observe that the robot stops moving. It does appear that the position of the ball and the robot makes the behaviour somewhat fragile, and that the robot can not be too far off the goal.

In the seventh scenario (figure 6.17g), the robot uses the camera feed to detect and follow persons. The person is placed into the virtual world. When the scenario is started, it shows a visualization of the persons that the Yolox ROS-package detects. It can be observed that the robot correctly moves towards

[8] https://gazebosim.org

[9] https://www.robotis.us/turtlebot-3-waffle-pi/

[10] https://wiki.ros.org/turtlebot_gazebo



(a) Line follower

(b) Simple navigation

(c) Obstacle navigation

(d) Object finder

(e) Maze solver

(f) Push ball into goal

(g) Person follower

(h) Supervisor

Figure 6.17: Screenshots of the simulation of each of the scenarios.

the person. If the person is moved to a different location, the robot finds the person and drives towards the person. The emergency stop ensures that no movement takes place if it was pressed.

The last scenario (figure 6.17h) demonstrates similar behaviour to the first scenario, as they both implement a line following robot. In this case, this is implemented using a supervisor which required an additional node that allowed for movement of the robot in time steps. The most important part of the supervisor is that it stops the robot when the emergency stop was pressed. Observe that when the scenario is launched, both the controller as well as the supervisor are started and that pressing the emergency stop correctly halts the robot.

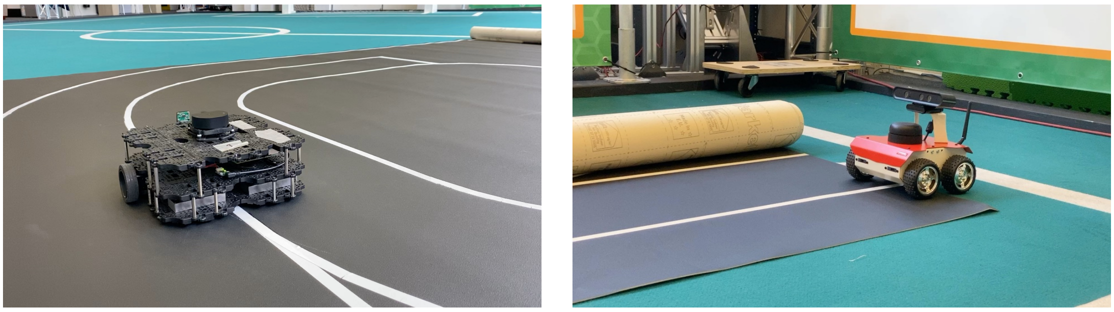
6.6 Physical hardware

To highlight that the solution not only works in a simulation, but also in the real world, the line follower (controller) scenario was also tested on two real robots, the TurtleBot 3 Waffle Pi and the ROSbot 2.0[11]. Both bots have been tested by running the ROS1 versions of the controller. The DSL already generated a ROS1-version of the controller. The line detector (which was not built using the DSL), however, was still targeting ROS2. The line detector was altered in order for it to run on both robots.

The first test was executed on the TurtleBot 3 Waffle Pi. Before the scenario could be run on the robot, some small adjustments had to be made. First, the line detection needed to be changed from yellow lines to white lines, as the environment that the robot would be deployed in, only consists of white lines. Furthermore, the field of view of the line detection needed to be altered. The camera feed of the real robot consists of more noise and could potentially detect a line while there was no line. This could be solved by limiting the portion of the camera feed that is considered when detecting lines. Furthermore, the topic names of the robot were slightly different than the ones that were used in the simulation. These needed to be altered for the movement and the camera. The data types, however, were still the same.

When running the program on the TurtleBot, it was found that the line detection was not updating fast enough. This meant that the robot could not rotate quickly enough for it to keep the line in the

[11] https://husarion.com/manuals/rosbot/



(a) TurtleBot (b) ROSbot

Figure 6.18: Pictures of the line follower on the TurtleBot 3.0 Waffle Pi and the ROSbot 2.0.

center of the camera. At some points, it could take up to 2 seconds until it had processed a new image. After some debugging, it was found that this was because the raw camera images were used, and they had to be sent over the network to the line detector node, which uses up a lot of bandwidth. Luckily, the robot also publishes the compressed images. This required a small modification in the line detector node, as it now first has to uncompress the image before it could be used. The DSL model itself was correct. After this was fixed, the line detection worked at a rate of around 15 Hz.

After the robot was running it required a bit of tweaking to find the correct linear and angular velocity, to ensure that the robot could rotate quickly enough in order for it to keep the line in center. This tweaking meant updating the data values in the model that were supplied to the robot. Eventually, a linear speed of 0.20 m/s and an angular speed of $\frac{correction}{450}\ \pi/s$ were used, which allowed the robot to move correctly. Although the line follower could still be improved (as it sometimes detected a line that was next to the one the robot was following), the robot supervisory controller works perfectly. The improvement of the line detector is outside of the scope of the thesis, but steps could be made such that it uses a light sensor or places a camera directly under the robot. Both were not possible in this project.

The second test was performed using the ROSbot. Apart from changing the name of the topics that control the movement of the robot, and the camera, the only required change was to alter the speed of the robot. Again, this meant that the data values in the model had to be updated. In the case of the ROSbot, a linear speed of 0.04 m/s was used and an angular speed of $\frac{correction}{500}\ \pi/s$, after which the robot correctly followed the line.

A picture of both robots following the line is shown in figure 6.18.

6.7 Limitations

The development of the eight scenarios which were created to evaluate the language with a diverse set of use cases led to some interesting insights. Although these scenarios highlight the expressiveness of the language, they also help to identify the limitations of the language.

One of the first limitations is that variables can only store relatively simple data, depending on their use. If a variable is used the supervisory controller itself, the type of the variable is limited to *booleans*, *enums* and *integers*. If it is a variable that is defined in C++ in the ROS node, it can also store *doubles* and *strings*. But, in neither of these cases it allows users to store complex data, like a ROS data object. In some cases, users might want to store a complex object, just to pass it to a different communication item. Due to a lack of time, this has not been implemented.

Next to that, the language does not support casting of doubles to integers or vice versa (or at least not explicitly, as a division by 1 could be used to convert the type of an integer to a double). These types are incompatible, and mixing them could lead to type errors. An option would be to have a function that can cast to either type. But, the language does not have any concept of a function at this point, which would take quite some time, and has therefore been skipped.

In some cases, the controller receives data in the form of an array that has no predefined size. An example of this could be an array which stores all the detected objects from the camera feed. The language has no option to count the amount of items in the array and has to rely on other communication to ensure that this data is passed along. Again, there is no concept of a function that could be re-used to support this.

Furthermore, the data that is passed within a provide statement is fixed. Although the provide statement itself can have a condition that needs to be matched in order for it to be sent along with any communication item, the values within the statement can not use conditional expressions. This could be solved by using multiple provide statements, but that leads to a lot of duplicate code and the amount of provide statements can grow very quickly. This can be fixed, but to keep the code clean it has been chosen not to implement this at the moment.

Another limitation, or more an improvement in this case, is that there is no option to define global constants which can be reused in expressions. For example, this constant can hold the threshold of a safe distance. Instead, users have to redefine a magic number in multiple places, with the risk of getting a mismatch between the number in different places. This can quite easily be implemented, but had no priority due to limited time.

Users have the option to define multiple enums. However, the names of these enum values have to be unique globally (within the scope of the robot). The reason for this, is that CIF does not allow duplicate enum values either. Specifying multiple enums that have overlapping enum value names results in errors when applying controller synthesis, and therefore causes an error in the controller generation process. This is a limitation of CIF, but could be repaired by the language, but because of time, it has been chosen not to do so for now.

There is also a different limitation with respect to enums. In some cases, the logic and transformation rules of an enum are very similar, but have different constants for example. There is no way to parameterize enums such that they can be reused, causing less duplicate code within the DSL file. This would be possible, but it needs re-thinking of the way enums are used in general in the language. An option could be to consider it as methods, but this concept has not been introduced because of time.

Some scenarios make use of libraries which define the data types and behaviour of components and the communication that takes place with that component. However, libraries within the DSL have no option to provide data to this communication. In some cases, this could be desirable. For example, libraries could provide a default stop message, which provides the robot with a velocity of 0. Implementing this has some side-effects, as the user might be unaware of the data that is already provided to the robot. An idea could be to make use of inheritance and use override keywords when supplying data to a communication item from a library. This has not been implemented because of limited time during the project.

Lastly, there is no validation on topics. Although it is hard to validate the actual names of a topic and check whether they are present in the middleware (as there is no binding in the language directly to the middleware), it could come in handy to check the syntax of a topic name. It could be added, but it would need knowledge of the specific middleware that is used within a project.

Chapter 7

Discussion

7.1 Answers to Research Questions

In this thesis, the *Robot Supervisory Controller* DSL was presented, which supports the development of supervisory controllers using model driven engineering. To guide the design of this language, some research questions were defined upfront. This section provides answers to these research questions.

RQ1. What main approaches are used for modelling robots with DSLs?

Chapter 3, section 3.2.1 contains research about the existing approaches that domain-specific languages within robotics take. One of the approaches, the vTSL language [23], lets users specify tasks and define them formally. It defines bindings for the ROS middleware and allows model-to-model transformations to an external model checker, converting the existing DSL model to a state machine that can be verified. Note that the behaviour of the middleware is modelled using components stubs, because the internal state of the middleware or its components is unknown.

A different approach, the MontiArcAutomaton approach, in [46] uses a component and connector architecture. The tool is text-based, but also supports integration with the Eclipse Modelling Framework to provide visual modelling tools. It models the behaviour of a robot into multiple components, which communicate with each other using ports. These ports have their own dedicated name and type and represent communication with a robotic middleware, ROS. Note, that only ROS messages are supported. The language is build with the MontiCore language workbench, which supports language composition in terms of embedding, aggregation and extension. Although the paper defines multiple languages, MontiCoreAutomaton and MonticoreAutomatonADL, only the first one is publicly available.

Another take is the somewhat older RobotML approach [40], but it still has some interesting perspectives to consider. It is based on the Eclipse Modeling Framework (EMF) together with the Papyrus extension, so it can facilitate code generation for different sets of tools. In the DSL, users can model the sensors, actuators and control system of a robot. Again, communications are defined using ports. These ports now both support the publish and subscriber architecture, as well as the request and response mechanism. The behaviour of the control system is modelled using state machines which represent the state of each system and base this on the incoming and outgoing connectors. The language also integrates tools for the deployment plan. The language still requires users to manually define bindings to the robotic middleware.

Lastly, Salty [16] lets users focus on writing specifications rather than implementations. These are written using the GR(1) specification syntax and allow for specification of initialization, transition and liveness properties for both the system as well as the environment. Specifications have the form where an assumed environment state, which results in enforced guarantees about the system. Salty synthesizes the controllers, which result in a mathematical representation, and converts them to generic programming language: Python, Java and C++. Salty defines inputs and outputs on the level of the controller, which support a basic set of data types. The language also adds some basic support for sanity checking, debugging and optimization.

To conclude, although there are many languages for robotic DSLs, a few different ones were highlighted. Some main approaches consist of both textual as well as graphical languages. There are languages that use component based approaches, either for verification, or to define different parts of the robots. Communication between components happens via ports or inputs and outputs on the controller. Furthermore, both task-based as well specification-based approaches exist.

RQ2. What concepts can be identified in the supervisory control domain for ROS?

All of the concepts that the language defines and requires from the supervisory control domain can be found in chapter 4, section 4.2. The concepts are based on ROS, supervisory control theory and related robotic domain-specific languages.

The languages introduces the communication concepts from ROS, namely the messages, services and actions, together with the interface that links the correct data types in the generated code. It also uses the concepts from supervisory control theory, which are used to perform the controller synthesis process. It uses the requirements and automata from supervisory control theory to allow state-based specifications of requirements. An explicit mapping can be found in the chapter on generators, chapter 5.

CIF was used for the supervisory controller synthesis process. But, because the concepts that are used are related to supervisory control theory, this tool could be replaced by any tool for supervisory control theory that meets the following requirements:

- Support supervisory controller synthesis
- Support code generation to C++/C

This is a bit different for the robotic middleware with respect to generalizability. The language introduces communication concepts that are specific to ROS. There are other robotic middlewares [15] with communication options between different software processes, but they do not have the definition of messages, services and actions. If this were to be generalized, for example in future work, these communication types should be decomposed into the publisher and subscriber architecture, and the client and response model. In this thesis, this has not been done, as the project has been scoped to ROS to ensure familiarity with the concepts.

RQ3. What kind of requirements should a supervisory controller be able to satisfy?

The thesis focuses on the synthesis as described in Supervisory Control Theory (SCT). Within supervisory control theory, requirements are specified on the controllable events that occur. This means that the supervisory controller adheres to these requirements by construction, as it is synthesized. To do so, the DSL uses concepts from SCT to define requirements (which are the properties that the supervisory controller satisfies), as described in section 4.2 of chapter 4. As described in section 2.4, supervisory controllers determines which controllable events are enabled given the state of the plants at some point based on a set of requirements. The controllable events that are present within the scope of this thesis, is the communication from the supervisory controller to the middleware. All of the controllable events that are defined for this communication are described in section 5.2. For messages, this means that the supervisor determines whether messages can be published to a topic or not. For services, it determines whether the controller can send a request to service server, and for action the supervisor supervises the sending of a goal, and cancelling an action.

In supervisory control theory, requirements can have of two types. There are requirements that disable controllable events based on a condition, and requirements that a controllable event needs to have satisfied in order for it to allow the controllable event to occur. If multiple disable requirements are specified, they are combined using a disjunction, meaning that if one of the conditions holds, the controllable event is not allowed in the current state. For requirements that specify a condition that needs to be true, they are combined using a conjunction, meaning that if one of them evaluates to false in the current state of the plants, then the controllable event is not allowed to occur. In the basis of supervisory control theory, these conditions should allow specifications on the current state of the plant. CIF supports to have variables defined on a plant, which requirement conditions can use in expressions like (in)equalities.

To conclude, for supervisory controller synthesis it is important that a supervisory controller can ensure that one of the plants/components is in a given state. But, CIF also supports the definition of variables on a plant. Conditions for the requirements can use these variables in expressions to ensure that they have specific values.

RQ4. Does the proposed DSL-solution work in one or more case studies?

To show that the domain-specific language that was proposed in the thesis works, chapter 6 contains an evaluation in the form of several distinct scenarios. Some of these scenarios use external nodes from GitHub for ROS, some of them use custom-built nodes for their dedicated task. A supervisory controller was created for each of them using the DSL by specifying the components, communication and the set of requirements. The following scenarios were tested:

- Line follower (controller)
- Simple navigation
- Obstacle navigation
- Object finder
- Maze solver
- Push ball into goal
- Person follower
- Line follower (supervisor)

All of the scenarios were running on the simulator and correctly achieved their goal. Although they worked correctly, some of the scenarios lead to insights that could improve or alter the concepts of the language. One of the main limitations is that the controller can only store simple data, even if the data is not required for controller synthesis. In some cases, it might be necessary to temporarily store incoming data, just to sent it to another node in the middleware. Furthermore, in some cases it might be necessary to parameterize enum transformations, as in some cases they are very similar, but have some minor differences. This can cause duplicate code.

The line follower controller was also tested on real hardware running ROS1. It ran on both the TurtleBot 3 as well as the ROSbot 2.0, whilst only requiring minor adjustments to make the scenario suitable to the platform. The only modifications to the controller that needed to be made, were speed changes and platform binding names. The line detector (which is outside the scope of this thesis) did need some alterations for it to work in the real world as well.

In conclusion, the language works very well for different scenarios. Although there is still room for improvement, all scenarios could be executed correctly, and one scenario was even tested on two real, physical, robots.

7.2 Comparison to Related Work

Throughout this thesis, several papers and other literature have been used and referenced. Chapter 3 describes existing work that was already analyzed in related fields, and has been used in this thesis.

The ideas in this thesis have been based on work by Kok et al. in [29]. Here, they use supervisory control theory with CIF to apply model-driven engineering to autonomous navigation in robotics with ROS. They model plants, synthesize them and use them in a manually constructed ROS-node. In this thesis, the CIF-code is automatically generated, minimizing replication for language users. Furthermore, the ROS-node is generated automatically as well, with the bindings to the robotic middleware in it, contrary to the work in [29], where the bindings have to be kept up-to-date manually, which can be very error-prone. Because the ROS bindings are generated automatically, the DSL presented in this thesis supports both ROS1 and ROS2 out-of-the-box. This thesis generalized the ideas of Kok et al. into a domain model, supporting more tasks than just autonomous navigation.

The section on related work contains some different approaches for robotic DSLs. Although not all of the DSL approaches model a robotic controller, some of the concepts used in the languages still apply. For example, vTSL [23] uses component stubs in verification. These components represent part of the robotic middleware, whose state is unknown. The definition of a component often is close to that of a ROS-node, but can consist of multiple nodes. This also applies to the concept of a component within this thesis. However, the approach is task-based, and the tasks are converted to an input for a model checker, whereas this thesis constructs a supervisory controller node with synthesis. Furthermore, there is also no code generation for the ROS-node in vTSL. It is listed in their section on future work.

Again, the MontiCoreAutomaton [46] also makes use of components that represent virtual parts of the robot. Communication is modelled using ports that have incoming and outgoing data. Similarly, RobotML also uses ports [13]. Both support the publisher/subscriber architecture, and RobotML supports the request/response architecture as well. Although this is a very good generalization of the communication, it does not work for actions, which is why this generalization was not applied in this thesis.

Next to that, RobotML also introduces parts of the ontology of the robot into their model. They define the concept of a sensor and an actuator, whereas in this thesis, the concepts are kept generic, such that it can be used for all sorts of ROS nodes. The RobotML language also defines control systems as state machines, whereas the DSL in this thesis synthesizes a supervisory controller based on requirements.

There is also the language Salty [16], that, similarly to this thesis, also applies synthesis to extract a controller. Salty makes us of GR(1) specifications to synthesize a controller, and does therefore not use supervisory control theory. Furthermore, Salty generates code for generic programming languages, rather than code for ROS. Also, the specification of properties can be quite hard for someone that has little to no technical knowledge of the specification format.

A survey that was performed by Nordmann et al. that discusses the state of domain-specific languages in robotics [33]. The importance of evaluation of robotic DSLs is highlighted, and they make the distinction between qualitative and quantitative evaluations. Chapter 6 first shows an evaluation based on a given scenario, and then that scenario is executed on different physical platforms, similar to [51] [61] [27] [25] [49] [44]. The thesis also contains quantitative evaluation in terms of memory usage, build time, execution time and size of the models in lines of code.

The same survey also shows that code generation is the most common scenario for using the model in model-driven engineering for robotics. Similarly to Salty [16], the language in this thesis produces a controller. Contrary to the Salty DSL, the DSL in this thesis generates a ROS-node that is ready to be deployed into a middleware.

A good ecosystem can help the adoption of a language. To highlight the expressiveness of a language to potential users, this thesis contains different example scenarios that show the workings, similar to [23] [16] [26]. This language has no external documentation apart from this thesis, contrary to [16] [61]. This is written down in section 7.4 as future work.

The ideas in this thesis are all based on a discrete event system, whereas Roszkowska and Jakubiak propose a hybrid solution for robot control [48], but in this case for multiple robots. The control of an individual robot happens using continuous systems, whereas a supervisory controller is only used for the control of the robots as a whole.

There is also some work done on the verification of a supervisor [22], next to the synthesis, by using complete testing and combining different parts of the whole workflow of the supervisor development chain. The approach in this thesis still lacks functionality for complete testing.

7.3 Correctness of Generated Code

Since the thesis uses supervisory controller synthesis, one can argue about the correctness of this synthesis process. Synthesis ensures the development of a correct-by-design supervisory controller. For the actual synthesis process, CIF is used. So, the generated supervisory controller will always supervise that the requirements hold. But, the CIF specification is not directly what the user models, but rather something that is generated.

It is hard to prove that the generated requirements are correctly generated. To do so, formal semantics of our DSL should be defined and it should be proven that the translation to supervisory control theory is semantics preserving, which is outside the scope of this thesis. However, one can argue over why it should be correct. The DSL in this thesis uses concepts that very closely relate to the concepts and semantics of supervisory control theory, and therefore CIF. The concept of a component is directly converted into a plant, together with the corresponding locations and transitions. The current way of checking whether the generator is correct, is by testing it.

The controllable events for the supervisory controller are mapped to communication within the DSL. The DSL defines the concept of communication items (messages, services and actions) which are modelled as one or more controllable events for the supervisory controller. Each requirement defined in the DSL can be accompanied by one or more communication items. In the generator process for supervisory controller synthesis, this is directly converted to the corresponding controllable event. By keeping close to the semantics of supervisory control theory, the language should have similar correctness properties.

An actual proof of correctness of the CIF-generator (and the resulting node) is outside of the scope of this thesis. One of the approaches to verify the supervisor is by creating an abstract test reference, as in [21] and checking its equivalence with the generated concrete controller which is running on the control system platform.

7.4 Future work

Although a lot of research and work was already done within the scope of this thesis, there is still room for improvements. To start, there are still some limitations that were found during the development of the different scenarios. These limitations were highlighted in section 6.7 and range from small improvements to the introduction of new concepts. For example, there is still work to do on the side of variables. Variables can only have relatively simple data types, which blocks users from storing a complex message and sending it to a different node in the middleware. Also, enums can cause a lot of duplicate code if only a few constants or variables need changing. This requires users to create a new separate enum that is almost the same as the original one. This could be solved by parameterizing these enums, for example. There are also some more small improvements that can be made, like the introduction of constants or the ability to reuse enum value names.

Currently, the language supports Eclipse and Visual Studio Code. But, because the language implements the Language Server Protocol, support can relatively easily be extended to other editors as well, as long as they support the same protocol. Supporting multiple editors allows users to use the tools they already know and love.

Furthermore, the scope of this thesis was limited to the use of ROS and CIF. A next step could be to support multiple middlewares (like MIRO or The Player Project [38] [15]) or synthesis tools. The support of multiple robotic middlewares would require generalizing some of the concepts that are currently defined in the language.

The thesis also presented novel work on a supervisor layer instead of a supervisory controller. Although there are still some limitations, it is interesting to see some implementations of re-inforcement learning with a supervisory layer.

Besides that, as already described in section 7.3, a formal proof of the correctness of the generated supervisor still misses. The supervisor based on the CIF requirements is correct, but one can only argue as to why the generated CIF-requirements based on the DSL are correct.

Lastly, there is a lot of information in this thesis, but there is no website or documentation that allows users to quickly get started and install the language. This still has to be written and could be stored on GitHub, for example.

Chapter 8

Conclusion

This thesis presents the RoboSC DSL: a language that allows users to model a robot supervisory controller for ROS using supervisory controller synthesis. Users model all the components that can be distinguished on the robot, and model their behaviour using different state-based specifications. The states of these components is updated based on communication from and to ROS. The language user defines a set of requirements that restrict the communication from the controller to the middleware. Both the components together with the requirements are then used to synthesize a supervisory controller.

The model defined by the user is used for supervisory control theory (SCT) and ROS generators, supporting both versions of ROS. CIF is used to apply controller synthesis, after which the controller is directly integrated into the ROS nodes, ready to be deployed on either a ROS1 or ROS2 platform. To support the integration with CIF, contributions to the open-source project of CIF have been made that extend the possibilities of interfacing with the code generator of CIF.

The language defines concepts to send data to the middleware and adds checks to ensure that only data that is actually required for controller synthesis ends up in the supervisory controller, limiting the possible states of the controller. It also has a type system that is used for type safety. Besides that, there is also a set of validation rules that apply semantic validation to the model specified by the user. Debugging tools have been added as a functionality to the language as well, which visualize the current state of each of the components together with their variables, making the debugging process more visual and easier to understand.

The thesis also presents initial work on a different approach, the supervisory layer, rather than a supervisory controller. The language adds support to connect an existing controller and apply remapping to it, such that all behaviour of the controller is first validated by the supervisor node before it reaches other destinations in the robotic middleware.

RoboSC was validated using eight scenarios, in each the robot had a different task (apart from the line follower, one model was intended for a supervisory controller, the other for a supervisory layer). All of these scenarios have been successfully simulated in a virtual environment, and the line follower scenario was even successfully tested on two physical robots.

To summarize the contributions of this thesis, RoboSC is presented as a domain-specific language that stimulates the development of safe ROS-based robotic applications using a supervisory controller. Using the DSL allows users to develop applications faster and safer for both ROS1 as well as ROS2. The language provides live visualization tools that allow for better understanding and debugging of the developed model. The DSL can also be used for a supervisory layer approach, which connects to existing controllers.

The source code and artifacts of the language have been published to GitHub [1]. The latest version of both the plugin for Visual Studio Code, as well for Eclipse, can be downloaded from the GitHub Releases page.

[1] https://github.com/bartwesselink/robot-supervisory-controller-dsl

Bibliography

[1] Marian Sorin Adam. Generative programming for functional safety in mobile robots. 2017. 14

[2] Max K.. Agoston. *Computer Graphics and Geometric Modeling: Implementation and Algorithms*. Springer, 2005. 60

[3] Maurice H ter Beek, Michel A Reniers, and Erik P de Vink. Supervisory controller synthesis for product lines using cif 3. In *International Symposium on Leveraging Applications of Formal Methods*, pages 856–873. Springer, 2016. 15

[4] Abhishek Bhattacharjee and Daniel Lustig. Architectural and operating system support for virtual memory. *Synthesis Lectures on Computer Architecture*, 12(5):1–175, 2017. 72

[5] Marko Bjelonic. YOLO ROS: Real-time object detection for ROS. `https://github.com/leggedrobotics/darknet_ros`, 2016–2018. 59, 64

[6] Grady Booch. Uml in action. *Communications of the ACM*, 42(10):26–28, 1999. 4

[7] Hendrik Bünder. Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages. In *MODELSWARD*, pages 129–140, 2019. 38

[8] Jianping Cai, Xuting Wan, Meimei Huo, and Jianzhong Wu. An algorithm of micromouse maze solving. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1995–2000. IEEE, 2010. 65

[9] Sukting Chong, Joe Dinius, and dps53. Turtlebot: Turtlebot 3 waffle pi. `https://www.robotis.us/turtlebot-3-waffle-pi/`. Accessed: 2022-05-12. 60

[10] Anne-Lise Courbis, Kahune Luu, Benjamin Grondin, and Kelly Roussel. A model driven architecture framework for robot design and automatic code generation. In *15th China-Europe International Symposium on Software Engineering Education*, 2019. 13

[11] Alberto Rodrigues Da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015. 4

[12] Ankush Desai, Tommaso Dreossi, and Sanjit A Seshia. Combining model checking and runtime verification for safe robotics. In *International Conference on Runtime Verification*, pages 172–189. Springer, 2017. 14

[13] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. Robotml, a domain-specific language to design, simulate and deploy robotic applications. In *International conference on simulation, modeling, and programming for autonomous robots*, pages 149–160. Springer, 2012. 11, 83

[14] Sven Efftinge and Miro Spoenemann. Eclipse XText: Why xtext? `https://www.eclipse.org/Xtext/`. Accessed: 2022-05-31. 38

[15] Ayssam Elkady and Tarek Sobh. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, 2012, 2012. 2, 5, 81, 84

[16] Trevor Elliott, Mohammed Alshiekh, Laura R Humphrey, Lee Pike, and Ufuk Topcu. Salty-a domain specific language for gr (1) specifications and designs. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 4545–4551. IEEE, 2019. 12, 13, 14, 80, 83

[17] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013. 5

[18] Klaus Fischer, Julian Krumeich, Dima Panfilenko, Marc Born, and Philippe Desfray. Based modeling: A stakeholder-centered approach for model-driven engineering. In *Advances and applications in model-driven engineering*, pages 317–341. IGI Global, 2014. 4

[19] Nadia Hammoudeh García, Harshavardhan Deshpande, André Santos, Björn Kahl, and Mirko Bordignon. Bootstrapping mde development from ros manual code: Part 2—model generation and leveraging models at runtime. *Software and Systems Modeling*, pages 1–24, 2021. 13

[20] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. Yolox: Exceeding yolo series in 2021. *arXiv preprint arXiv:2107.08430*, 2021. 59

[21] Mario Gleirscher and Jan Peleska. Complete test of synthesised safety supervisors for robots and autonomous systems. *arXiv preprint arXiv:2110.12589*, 2021. 15, 84

[22] Mario Gleirscher, Lukas Plecher, and Jan Peleska. Sound development of safety supervisors. *arXiv preprint arXiv:2203.08917*, 2022. 15, 83

[23] Christian Heinzemann and Ralph Lange. vtsl-a formally verifiable dsl for specifying robot tasks. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8308–8314. IEEE, 2018. 9, 14, 80, 83

[24] Nico Hochgeschwender and Sebastian Wrede. Dsls in robotics: A case study in programming self-reconfigurable robots. *Grand Timely Topics in Software Engineering: International Summer School GTTSE 2015, Braga, Portugal, August 23-29, 2015, Tutorial Lectures*, 10223:98, 2017. 14

[25] Nicole Hutchins. A dsml for a robotics environment to support synergistic learning of ct and geometry. kong, sc, sheldon, j., & li, ky.(eds.). conference. In *Proceedings of International Conference on Computational Thinking Education 2018.*, 2018. 13, 83

[26] Yu-qian Jiang, Shi-qi Zhang, Piyush Khandelwal, and Peter Stone. Task planning in robotics: an empirical comparison of pddl-and asp-based systems. *Frontiers of Information Technology & Electronic Engineering*, 20(3):363–373, 2019. 14, 83

[27] Andrés C Jiménez, John P Anzola, Vicente García-Díaz, Rubén González Crespo, and Liping Zhao. Pydslrep: A domain-specific language for robotic simulation in v-rep. *Plos one*, 15(7):e0235271, 2020. 13, 14, 83

[28] J. W. Kok. Synthesis-based engineering of supervisory controller for autonomous robotic navigation. 2020. 15

[29] JW Kok, Elena Torta, Michel A Reniers, JM van de Mortel-Fronczak, and MJG van de Molengraft. Synthesis-based engineering of supervisory controllers for autonomous robotic navigation. *IFAC-PapersOnLine*, 54(2):259–264, 2021. 1, 14, 16, 82, 83

[30] Hadas Kress-Gazit, Morteza Lahijanian, and Vasumathi Raman. Synthesis for robots: Guarantees and feedback for robot behavior. *Annual Review of Control, Robotics, and Autonomous Systems*, 1:211–236, 2018. 15

[31] Steven Macenski, Francisco Martin, Ruffin White, and Jonatan Ginés Clavero. The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020. 58, 59, 62

[32] Microsoft. Language Server Protocol: Specification. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/. Accessed: 2022-05-09. 38

[33] Arne Nordmann, Nico Hochgeschwender, Dennis Wigand, and Sebastian Wrede. A survey on domain-specific modeling and languages in robotics. 2016. 9, 13, 14, 83

[34] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. A survey on domain-specific languages in robotics. In *International conference on simulation, modeling, and programming for autonomous robots*, pages 195–206. Springer, 2014. 4

[35] Eindhoven University of Technology. CIF3: Compositional interchange format. https://cstweb.wtb.tue.nl/cif/trunk-r9682/. Accessed: 2022-05-02. 7

[36] Eindhoven University of Technology. CIF3: Data-based supervisory controller synthesis. https://cstweb.wtb.tue.nl/cif/trunk-r9682/tools/datasynth.html. Accessed: 2022-05-05. 7, 24, 43

[37] Eindhoven University of Technology. CIF3: Global read, local write. https://cstweb.wtb.tue.nl/cif/trunk-r9682/lang/tut/data/read-write.html. Accessed: 2022-05-05. 33

[38] University of Ulm. Miro: Middleware for robots. http://users.isr.ist.utl.pt/~jseq/ResearchAtelier/misc/Miro%20-%20Middleware%20for%20Robots. Accessed: 2022-06-30. 5, 84

[39] OpenCV. OpenCV: Homepage. https://www.robotis.us/turtlebot-3-waffle-pi/. Accessed: 2022-05-31. 60

[40] RobotML organization. GitHub: Organization page for robotml. https://github.com/orgs/RobotML/repositories. Accessed: 2022-02-10. 12, 80

[41] OSRF. Gazebo: Compositional interchange format. https://classic.gazebosim.org/tutorials?tut=ros2_overview. Accessed: 2022-05-02. 76

[42] The Player Project. The player project. https://playerproject.github.io. Accessed: 2022-06-30. 5

[43] Peter J Ramadge and W Murray Wonham. Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization*, 25(1):206–230, 1987. 1, 2, 6, 24

[44] Arunkumar Ramaswamy, Bruno Monsuez, and Adriana Tapus. Formal specification of robotic architectures for experimental robotics. In *Metrics of Sensory Motor Coordination and Integration in Robots and Animals*, pages 15–37. Springer, 2020. 13, 83

[45] Barry Ridge, Timotej Gaspar, and Aleš Ude. Rapid state machine assembly for modular robot control using meta-scripting, templating and code generation. In *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pages 661–668. IEEE, 2017. 13

[46] Jan Oliver Ringert, Roth Alexander, Rumpe Bernhard, and Wortmann Andreas. Language and code generator composition for model-driven engineering of robotics component & connector systems. 2015. 10, 80, 83

[47] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Code generator composition for model-driven engineering of robotics component & connector systems. *arXiv preprint arXiv:1505.00904*, 2015. 13

[48] Elzbieta Roszkowska and Janusz Jakubiak. Control synthesis for multiple mobile robot systems. *Transactions of the Institute of Measurement and Control*, page 01423312211047061, 2021. 15, 83

[49] Jacob Sacks, Divya Mahajan, Richard C Lawson, and Hadi Esmaeilzadeh. Robox: an end-to-end solution to accelerate autonomous control in robotics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 479–490. IEEE, 2018. 13, 14, 83

[50] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE software*, 20(5):42–45, 2003. 4

[51] Craig J Sutherland and Bruce MacDonald. Robolang: A simple domain specific language to script robot interactions. In *2019 16th International Conference on Ubiquitous Robots (UR)*, pages 265–270. IEEE, 2019. 13, 14, 83

[52] Robot Operating System. ROS: Nodes. `https://wiki.ros.org/Nodes`. Accessed: 2022-06-24. 5

[53] Robot Operating System. ROS2: About ros interfaces. `https://docs.ros.org/en/galactic/Concepts/About-ROS-Interfaces.html`. Accessed: 2022-03-17. 22

[54] Robot Operating System. ROS2: Changes between ros 1 and ros. `http://design.ros2.org/articles/changes.html`. Accessed: 2022-03-08. 5, 6

[55] Robot Operating System. ROS2: Creating a launch file. `https://docs.ros.org/en/foxy/Tutorials/Launch/Creating-Launch-Files.html`. Accessed: 2022-05-07. 6

[56] Robot Operating System. ROS2: Rolling documentation. `https://docs.ros.org/en/rolling/`. Accessed: 2022-03-08. 5

[57] Robot Operating System. ROS2: Understanding ros2 actions. `https://docs.ros.org/en/galactic/Tutorials/Understanding-ROS2-Actions.html`. Accessed: 2022-03-16. 24

[58] Robot Operating System. ROS2: Understanding ros2 nodes. `https://docs.ros.org/en/galactic/Tutorials/Understanding-ROS2-Nodes.html`. Accessed: 2022-03-10. 18

[59] Eelco Visser. Webdsl: A case study in domain-specific language engineering. In *International summer school on generative and transformational techniques in software engineering*, pages 291–373. Springer, 2007. 4

[60] Johannes Wienke, Dennis Wigand, Norman Koster, and Sebastian Wrede. Model-based performance testing for robotics software components. In *2018 Second IEEE International Conference on Robotic Computing (IRC)*, pages 25–32. IEEE, 2018. 13, 14

[61] Dennis Leroy Wigand and Sebastian Wrede. Model-driven scheduling of real-time tasks for robotics systems. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 46–53. IEEE, 2019. 13, 14, 83

[62] W Murray Wonham, Kai Cai, et al. Supervisory control of discrete-event systems. 2019. 6, 16

[63] Andrew Yong. Medium: Maze escape with wall-following algorithm. `https://andrewyong7338.medium.com/maze-escape-with-wall-following-algorithm-170c35b88e00`. Accessed: 2022-04-28. 59

Appendix A

Language Concepts Sample

```
robot ThesisLineFollower {
    interface twist use Twist from geometry_msgs
    interface light use LightBulbService from robot_common
    interface grabber use GrabberAction from robot_common

    datatype object Vector3 {
        x: double
        y: double
        z: double
    }

    datatype object Twist {
        angular: Vector3
        linear: Vector3
    }

    datatype enum DistanceEnum from double to {
        value >= 10 -> free
        default -> obstructed
    }

    datatype object LightBulbRequest {
        ^state: boolean
    }

    component DistanceSensor {
        outgoing message distance with identifier: "/distance", type: DistanceEnum

        behaviour {
            variable current: DistanceEnum = obstructed

            initial state sensing {
                on response from distance do current := value
            }
        }
    }

    component ObstacleGrabber {
        action grab with identifier: "/grab", request: none, response: none, feedback: none
            links grabber

        behaviour {
            initial state idle {
                on request to grab goto grabbing
            }

            state grabbing {
                on response from grab goto idle
            }
        }
    }

    component LightBulb {
        service set_light_state with request: LightBulbRequest, response: none links light
    }

    component LineDetector {
        outgoing message correction with identifier: "/correction", type: double
        outgoing message no_line with identifier: "/no_line", type: none

        behaviour {
            variable current: double = 0.0

            initial state no_line {
                on response from correction goto line_found
            }

            state line_found {
                on response from no_line goto no_line
                on response from correction do current := value
            }
        }
    }

    component Motor {
        incoming message move with identifier: "/vel", type: Twist links twist
        incoming message stop with identifier: "/vel", type: Twist links twist
    }

    // Provide communication with the correct speed
    provide move with {
        linear: { x: 0.4 },
        angular: { z: LineDetector.current / 100 }
    }
    provide stop with {
        linear: { x: 0.0 },
        angular: { z: 0.0 }
    }

    // Enable light when no line found
    provide set_light_state with { ^state: false } if LineDetector.line_found
    provide set_light_state with { ^state: true } if LineDetector.no_line

    // Requirements
    // Movement
    requirement move needs DistanceSensor.current = free
    requirement move needs ObstacleGrabber.idle
    requirement stop needs DistanceSensor.current = obstructed or ObstacleGrabber.grabbing

    // Grabbing
    requirement DistanceSensor.current = free disables grab
}
```

Listing A.1: DSL code for the running example in the language concepts section.

Appendix B

Line Follower Scenario

```
robot LineFollowerController {
    interface laser use LaserScan from sensor_msgs

    component LineDetector {
        outgoing message correction with identifier: "/correction", type: double
        outgoing message no_line with identifier: "/no_line", type: none

        behaviour {
            variable current_correction: double = 0.0

            initial marked state no_line {
                on response from correction goto line_found
            }

            state line_found {
                on response from no_line goto no_line
                on response from correction do current_correction := value
            }
        }
    }

    datatype object LaserScan {
        ranges: array(double)
    }

    datatype enum DistanceSafety from LaserScan to {
        value.ranges[0] > 0.25 or value.ranges[0] <= 0.0 -> safe
        default -> unsafe
    }

    component LidarSensor {
        outgoing message scan with identifier: "/scan", type: DistanceSafety links laser

        behaviour {
            variable current_distance: DistanceSafety = unsafe

            on response from scan do current_distance := value

            initial marked state unsafe_distance {
                transition if current_distance = safe goto safe_distance
            }

            state safe_distance {
                transition if current_distance = unsafe goto unsafe_distance
            }
        }
    }
    component EmergencyStop from EmergencyStopLibrary import EmergencyStop
    component TurtlebotPlatform from TurtlebotLibrary import TurtlebotPlatform

    requirement halt needs EmergencyStop.stopped or LineDetector.no_line or LidarSensor.
        unsafe_distance
    requirement move needs EmergencyStop.in_service and LineDetector.line_found and
        LidarSensor.safe_distance

    provide move with {
        linear: { x: 0.6 },
        angular: { z: (—LineDetector.current_correction) / 100 }
    }

    provide halt with { linear: { x: 0.0 }, angular: { z: 0.0 }}
}
```

Listing B.1: DSL code for the line follower scenario.

Appendix C

Simple Navigation Scenario

```
robot SimpleNavigation {
    interface point use PointStamped from geometry_msgs
    interface pose use PoseWithCovarianceStamped from geometry_msgs
    interface navigate use NavigateToPose from nav2_msgs

    datatype object Quaternion {
        x: double
        y: double
        z: double
        w: double
    }

    datatype object Point {
        x: double
        y: double
        z: double
    }

    datatype object Pose {
        position: Point
        orientation: Quaternion
    }

    datatype object PoseStamped {
        pose: Pose
    }

    datatype object PointStamped {
      point: Point
    }

    datatype object NavigateToPoseRequest {
        pose: PoseStamped
    }

    component Nav2 {
      outgoing message point with identifier: "/clicked_point", type: PointStamped links point
        outgoing message initial_pose with identifier: "/initialpose", type: Pose links pose
        action navigate with identifier: "/navigate_to_pose", request: NavigateToPoseRequest,
             response: none, feedback: none links navigate

      behaviour {
        variable current_x: double
        variable current_y: double
        variable current_z: double

        initial state no_initial_pose {
                on response from initial_pose goto awaiting_point
```

```
            }

            marked state awaiting_point {
          on response from point do current_x := value.point.x, current_y := value.point.y,
              current_z := value.point.z goto has_point
            }

        state has_point {
            on response from navigate goto awaiting_point
            on cancel from navigate goto awaiting_point
        }
      }
    }

    component EmergencyStop from EmergencyStopLibrary import EmergencyStop

    requirement navigate needs EmergencyStop.in_service
    requirement navigate needs Nav2.has_point

    provide navigate with {
      pose: {
        pose: {
          position: { x: Nav2.current_x, y: Nav2.current_y, z: Nav2.current_z }
        }
      }
    }
}
```

Listing C.1: DSL code for the simple navigation scenario.

Appendix D

Obstacle Navigation Scenario

```
robot ObstacleNavigation {
    interface point use PointStamped from geometry_msgs
    interface pose use PoseWithCovarianceStamped from geometry_msgs
    interface navigate use NavigateToPose from nav2_msgs

    datatype object Quaternion {
        x: double
        y: double
        z: double
        w: double
    }

    datatype object Point {
        x: double
        y: double
        z: double
    }

    datatype object Pose {
        position: Point
        orientation: Quaternion
    }

    datatype object PoseStamped {
        pose: Pose
    }

    datatype object PointStamped {
      point: Point
    }

    datatype object NavigateToPoseRequest {
        pose: PoseStamped
    }

    component Nav2 {
      outgoing message point with identifier: "/clicked_point", type: PointStamped links point
        outgoing message initial_pose with identifier: "/initialpose", type: Pose links pose
        action navigate with identifier: "/navigate_to_pose", request: NavigateToPoseRequest,
             response: none, feedback: none links navigate

      behaviour {
        variable current_x: double
        variable current_y: double
        variable current_z: double

        initial state no_initial_pose {
                on response from initial_pose goto awaiting_point
```

```
            }

            marked state awaiting_point {
          on response from point do current_x := value.point.x, current_y := value.point.y,
              current_z := value.point.z goto has_point
            }

        state has_point {
            on response from navigate goto awaiting_point
            on cancel from navigate goto awaiting_point
        }
      }
    }

    component EmergencyStop from EmergencyStopLibrary import EmergencyStop

    requirement navigate needs EmergencyStop.in_service
    requirement navigate needs Nav2.has_point

    provide navigate with {
      pose: {
        pose: {
          position: { x: Nav2.current_x, y: Nav2.current_y, z: Nav2.current_z }
        }
      }
    }
}
```

Listing D.1: DSL code for the obstacle navigation scenario.

Appendix E

Object Finder Scenario

```
robot ObjectFinder {
    interface laser use LaserScan from sensor_msgs
    interface count use ObjectCount from darknet_ros_msgs
    interface boxes use BoundingBoxes from darknet_ros_msgs

    datatype object LaserScan {
        ranges: array(double)
    }

    datatype enum DistanceFront from LaserScan to {
        value.ranges[0] >= 0.6 -> safe_front
        default -> unsafe_front
    }

    datatype enum DistanceLeft from LaserScan to {
        value.ranges[90] >= 0.6 -> safe_left
        default -> unsafe_left
    }

    datatype enum DistanceRight from LaserScan to {
        value.ranges[270] >= 0.6 -> safe_right
        default -> unsafe_right
    }

    datatype object BoundingBoxes {
        bounding_boxes: array(BoundingBox)
    }

    datatype object BoundingBox {
        class_id: string
    }

    datatype object ObjectCount {
        count: integer(0..1)
    }

    datatype enum ScannedObject from BoundingBoxes to {
        value.bounding_boxes[0].class_id = "stop sign" -> stop_sign
        default -> no_object_found
    }

    component LidarScanner {
        outgoing message scan_front with identifier: "/scan", type: DistanceFront links laser
        outgoing message scan_left with identifier: "/scan", type: DistanceLeft links laser
        outgoing message scan_right with identifier: "/scan", type: DistanceRight links laser

        behaviour {
            variable front: DistanceFront
            variable left: DistanceLeft
            variable right: DistanceRight
            variable has_front: boolean = false

            initial marked state sensing {
                on response from scan_front do front := value, has_front := true
                on response from scan_left do left := value
                on response from scan_right do right := value
            }
        }
    }

    component Rotator {
        incoming message rotate_left with identifier: "/rotate_left", type: none
        incoming message rotate_right with identifier: "/rotate_right", type: none
        outgoing message rotate_done with identifier: "/rotate_done", type: none

        behaviour {
            initial marked state awaiting_command {
                on request to rotate_left goto executing
                on request to rotate_right goto executing
            }

            state executing {
                on response from rotate_done goto awaiting_command
            }
        }
    }

    component ObjectDetector {
        outgoing message object_count with identifier: "/darknet_ros/found_object", type:
            ObjectCount links count
        outgoing message object_scan with identifier: "/darknet_ros/bounding_boxes", type:
            ScannedObject links boxes

        behaviour {
            variable scanned_object_count: integer(0..1)
            variable scanned_object: ScannedObject

            on response from object_count do scanned_object_count := value.count
            on response from object_scan do scanned_object := value

            initial marked state no_object {
                transition if scanned_object_count > 0 and scanned_object = stop_sign goto
                    object_found
            }

            state object_found {
                transition if scanned_object_count = 0 goto no_object
            }
        }
    }

    component TurtlebotPlatfrom from TurtlebotLibrary import TurtlebotPlatform
    component EmergencyStop from EmergencyStopLibrary import EmergencyStop

    requirement move needs LidarScanner.front = safe_front and Rotator.awaiting_command and
        EmergencyStop.in_service
    requirement halt needs (LidarScanner.left = unsafe_left and LidarScanner.right =
        unsafe_right and LidarScanner.front = unsafe_front) or ObjectDetector.object_found or
         EmergencyStop.stopped

    requirement rotate_left needs LidarScanner.left = safe_left and !ObjectDetector.
        object_found and EmergencyStop.in_service
    requirement rotate_right needs LidarScanner.right = safe_right and !ObjectDetector.
        object_found and EmergencyStop.in_service
    requirement { rotate_left, rotate_right } needs Rotator.awaiting_command
    requirement { rotate_left, rotate_right } needs LidarScanner.front = unsafe_front and
        LidarScanner.has_front

    requirement ObjectDetector.object_found disables move

    provide move with { linear: { x: 0.5 }, angular: { z: 0.0 } }
    provide halt with { linear: { x: 0.0 }, angular: { z: 0.0 }}
}
```

Listing E.1: DSL code for the object finder scenario.

Appendix F

Maze Solver Scenario

```
robot MazeSolver {
    interface laser use LaserScan from sensor_msgs
    interface movement use Twist from geometry_msgs

    datatype object LaserScan {
        ranges: array(double)
    }

    datatype object Vector3 {
        x: double
        y: double
        z: double
    }

    datatype object Twist {
        linear: Vector3
        angular: Vector3
    }

    datatype enum DistanceRight from LaserScan to {
        value.ranges[270] < 0.7 or value.ranges[240] < 0.7 -> wall_right
        default -> no_wall_right
    }

    datatype enum DistanceFront from LaserScan to {
        value.ranges[0] < 0.6 -> wall_front
        default -> no_wall_front
    }

    datatype enum DistanceDiagRight from LaserScan to {
        value.ranges[225] < 0.9 -> wall_diag_right
        default -> no_wall_diag_right
    }

    component Distance {
        outgoing message scan_right with identifier: "/scan", type: DistanceRight links laser
        outgoing message scan_front with identifier: "/scan", type: DistanceFront links laser
        outgoing message scan_diag_right with identifier: "/scan", type: DistanceDiagRight
            links laser

        behaviour {
            variable right: DistanceRight = no_wall_right
            variable front: DistanceFront = no_wall_front
            variable diag_right: DistanceDiagRight = no_wall_diag_right

            on response from scan_right do right := value
            on response from scan_front do front := value
            on response from scan_diag_right do diag_right := value
            initial marked state sensing {}
        }
    }

    component Platform {
        incoming message movement with identifier: "/cmd_vel", type: Twist links movement
        incoming message halt with identifier: "/cmd_vel", type: Twist links movement

        incoming message turn_left with identifier: "/rotate_left", type: integer
        incoming message turn_right with identifier: "/rotate_right", type: integer

        outgoing message rotate_done with identifier: "/rotate_done", type: none

        behaviour {
            initial marked state ready {
                on request to turn_left goto turning
                on request to turn_right goto turning
            }

            state turning {
                on response from rotate_done goto ready
                on response from stop goto ready
            }
        }
    }

    component EmergencyStop from EmergencyStopLibrary import EmergencyStop

    // Rules when following left-walls
    requirement movement needs (Distance.right = wall_right and Distance.front = no_wall_front
        )
            or (Distance.front = no_wall_front and Distance.right = no_wall_right and Distance
                .diag_right = no_wall_diag_right)

    requirement turn_right needs Distance.right = no_wall_right and Distance.diag_right =
        wall_diag_right
    requirement turn_left needs Distance.front = wall_front

    requirement {turn_left, turn_right, movement} needs Platform.ready

    requirement EmergencyStop.stopped disables {
        movement,
        turn_left,turn_right
    }

    requirement halt needs EmergencyStop.stopped

    // Provide communication with correct data
    provide turn_left with 90 // degrees
    provide turn_right with 90 // degrees

    provide movement with { linear: { x: 0.3 } }
    provide halt with { linear: { x: 0.0 } }
}
```

Listing F.1: DSL code for the maze solver scenario.

Appendix G

Push Ball Into Goal Scenario

```
robot MazeSolver {
    interface laser use LaserScan from sensor_msgs
    interface movement use Twist from geometry_msgs

    datatype object LaserScan {
        ranges: array(double)
    }

    datatype object Vector3 {
        x: double
        y: double
        z: double
    }

    datatype object Twist {
        linear: Vector3
        angular: Vector3
    }

    datatype enum DistanceRight from LaserScan to {
        value.ranges[270] < 0.7 or value.ranges[240] < 0.7 -> wall_right
        default -> no_wall_right
    }

    datatype enum DistanceFront from LaserScan to {
        value.ranges[0] < 0.6 -> wall_front
        default -> no_wall_front
    }

    datatype enum DistanceDiagRight from LaserScan to {
        value.ranges[225] < 0.9 -> wall_diag_right
        default -> no_wall_diag_right
    }

    component Distance {
        outgoing message scan_right with identifier: "/scan", type: DistanceRight links laser
        outgoing message scan_front with identifier: "/scan", type: DistanceFront links laser
        outgoing message scan_diag_right with identifier: "/scan", type: DistanceDiagRight
            links laser

        behaviour {
            variable right: DistanceRight = no_wall_right
            variable front: DistanceFront = no_wall_front
            variable diag_right: DistanceDiagRight = no_wall_diag_right

            on response from scan_right do right := value
            on response from scan_front do front := value
            on response from scan_diag_right do diag_right := value
            initial marked state sensing {}
        }
    }

    component Platform {
        incoming message movement with identifier: "/cmd_vel", type: Twist links movement
        incoming message halt with identifier: "/cmd_vel", type: Twist links movement

        incoming message turn_left with identifier: "/rotate_left", type: integer
        incoming message turn_right with identifier: "/rotate_right", type: integer

        outgoing message rotate_done with identifier: "/rotate_done", type: none

        behaviour {
            initial marked state ready {
                on request to turn_left goto turning
                on request to turn_right goto turning
            }

            state turning {
                on response from rotate_done goto ready
                on response from stop goto ready
            }
        }
    }

    component EmergencyStop from EmergencyStopLibrary import EmergencyStop

    // Rules when following left-walls
    requirement movement needs (Distance.right = wall_right and Distance.front = no_wall_front
         )
            or (Distance.front = no_wall_front and Distance.right = no_wall_right and Distance
                .diag_right = no_wall_diag_right)

    requirement turn_right needs Distance.right = no_wall_right and Distance.diag_right =
         wall_diag_right
    requirement turn_left needs Distance.front = wall_front

    requirement {turn_left, turn_right, movement} needs Platform.ready

    requirement EmergencyStop.stopped disables {
        movement,
        turn_left,turn_right
    }

    requirement halt needs EmergencyStop.stopped

    // Provide communication with correct data
    provide turn_left with 90 // degrees
    provide turn_right with 90 // degrees

    provide movement with { linear: { x: 0.3 } }
    provide halt with { linear: { x: 0.0 } }
}
```

Listing G.1: DSL code for the push ball into goal scenario.

Appendix H

Person Follower Scenario

```
robot PersonFollowing {
    interface laser use LaserScan from sensor_msgs
    interface boxes use BoundingBoxes from bboxes_ex_msgs

    datatype object BoundingBoxes {
        bounding_boxes: array(BoundingBox)
    }

    datatype object BoundingBox {
        xmax: double
        xmin: double
        img_width: double
    }

    datatype object LaserScan {
        ranges: array(double)
    }

    datatype enum Distance from LaserScan to {
        value.ranges[0] > 5.0 and value.ranges[350] > 5.0 and value.ranges[10] > 5.0 -> free
        default -> person
    }

    component Scanner {
        outgoing message scan with identifier: "/scan", type: Distance links laser

        behaviour {
            variable distance: Distance

            initial marked state sensing {
                on response from scan do distance := value
            }
        }
    }

    component YoloxDetection {
        outgoing message bounding_boxes with identifier: "/bounding_boxes", type:
            BoundingBoxes links boxes

        behaviour {
            variable current_image_size: double = 0.0
            variable current_xmax: double = 0.0
            variable current_xmin: double = 0.0

            on response from bounding_boxes do current_image_size := value.bounding_boxes[0].
                img_width,
                                                current_xmax := value.bounding_boxes[0].xmax,
                                                current_xmin := value.bounding_boxes[0].xmin
```

```
                                            goto detected

            initial state initializing {}
            marked state detected {}
        }
    }

    component EmergencyStop from EmergencyStopLibrary import EmergencyStop
    component TurtlebotPlatform from TurtlebotLibrary import TurtlebotPlatform

    // Base requirements
    requirement halt needs EmergencyStop.stopped or Scanner.distance = person
    requirement move needs EmergencyStop.in_service

    requirement move needs YoloxDetection.detected

    // Data for movement
    provide move with {
        linear: { x: 0.0 },
        angular: { z: ((YoloxDetection.current_image_size / 2) - ((YoloxDetection.current_xmin
             + YoloxDetection.current_xmax) / 2)) / 1000 }
    } if YoloxDetection.detected and Scanner.distance = person

    provide move with {
        linear: { x: 0.2 },
        angular: { z: ((YoloxDetection.current_image_size / 2) - ((YoloxDetection.current_xmin
             + YoloxDetection.current_xmax) / 2)) / 1000 }
    } if YoloxDetection.detected and Scanner.distance = free

    provide move with {
        linear: { x: 0.0 },
        angular: { z: 0.3 }
    } if YoloxDetection.initializing

    provide halt with { linear: { x: 0.0 }, angular: { z: 0.0 }}
}
```

Listing H.1: DSL code for the person follower scenario.

Appendix I

Supervisor Scenario

```
robot LineFollowerSupervised {
    interface laser use LaserScan from sensor_msgs
    interface movement use Twist from geometry_msgs

    datatype object Vector3 {
        x: double
        y: double
        z: double
    }

    datatype object Twist {
        linear: Vector3
        angular: Vector3
    }

    component LineDetector {
        outgoing message correction with identifier: "/correction", type: double
        outgoing message no_line with identifier: "/no_line", type: none

        behaviour {
            variable current_correction: double = 0.0

            initial marked state no_line {
                on response from correction goto line_found
            }

            state line_found {
                on response from no_line goto no_line
                on response from correction do current_correction := value
            }
        }
    }

    datatype object LaserScan {
        ranges: array(double)
    }

    datatype enum DistanceSafety from LaserScan to {
        value.ranges[0] > 1.0 and value.ranges[270] > 0.5 and value.ranges[90] > 0.5 and value
            .ranges[45] > 0.7 and value.ranges[305] > 0.7-> safe
        default -> unsafe
    }

    component LidarSensor {
        outgoing message scan with identifier: "/scan", type: DistanceSafety links laser

        behaviour {
            variable current_distance: DistanceSafety = unsafe
            on response from scan do current_distance := value

            initial marked state unsafe_distance {
                transition if current_distance = safe goto safe_distance
            }

            state safe_distance {
                transition if current_distance = unsafe goto unsafe_distance
            }
        }
    }

    component SimpleMovement {
        incoming message move with identifier: "/simple_movement", type: Twist links movement
    }

    component EmergencyStop from EmergencyStopLibrary import EmergencyStop

    requirement move needs LineDetector.line_found
    requirement move needs EmergencyStop.in_service
    requirement move needs LidarSensor.safe_distance

    provide move with {
        linear: { x: 0.6 },
        angular: { z: (-LineDetector.current_correction) / 100 }
    }
}
```

Listing I.1: DSL code for the supervisor scenario.