Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Petter Bjørseth

New visual programming approaches for robots

Master's thesis in Computer science
Supervisor: Michael Engel
July 2022

NTNU
Norwegian University of
Science and Technology

Petter Bjørseth

New visual programming approaches for robots

Master's thesis in Computer science
Supervisor: Michael Engel
July 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Robot programming is a complex endeavour that requires advanced programming skills. Visual programming aims to reduce the syntactic complexity of programming languages and ease development of complex applications.

This thesis explores visual programming approaches for robots that aim to reduce the complexity of robot programming. This is done by creating a simple visual programming language that compiles to Java code. This Java code can then be run in the Webots robot simulator.

This visual programming language is evaluated as less complex and simpler to learn for non-programmers. It does, however, still require some programming knowledge to use efficiently.

Sammendrag

Programmering av roboter er et komplekst problemområde som krever avanserte programmeringsferdigheter. Visuelle programmeringsspråk ønsker å senke den syntaktiske kompleksiteten til programmeringsspråk og senke terskelen for å utvikle komplekse applikasjoner.

Denne masteroppgaven utforsker muligheten for å anvende visuelle programmeringsspråk for å redusere kompleksiteten ved programmering av roboter. Dette utføres ved å utvikle et enkelt visuelt programmeringsspråk som kompilerer til Java kode. Denne Java koden kan deretter kjøres i robot simulatoren Webots.

Dette visuelle programmeringsspråket evalueres til å være mindre komplekst, og enklere å lære for ikke-programmere. Det krever dog fortsatt noe kunnskap om programmering for å kunne brukes effektivt.

Preface

This thesis continues the work from the specialization project TDT4501. For this reason, and since it might not be available to the reader, some parts of chapter 2 are adapted or directly taken from the report of that project [1].

Contents

Abstract i

Sammendrag ii

Preface iii

List of Figures vii

1 Introduction 1

2 Previous work 3

2.1 Visual programming . . . 3

2.2 Statecharts . . . 3

2.2.1 Clustering . . . 4

2.2.2 Orthogonality . . . 5

2.2.3 Default states . . . 6

2.2.4 History . . . 6

2.2.5 Economizing of arrows . . . 7

2.2.6 Condition and selection entrances . . . 7

2.2.7 Delays and timeouts . . . 8

2.2.8 Unclustering . . . 8

2.2.9 Actions and activities . . . 9

2.2.10 Broadcast . . . 9

2.2.11 Problems with statecharts . . . 10

2.3 Coordinating robotic tasks and systems with rFSM statecharts . . . 10

2.4 The ArmarX statechart concept: graphical programming of robot behavior . . . 12

2.5 RAFCON . . . 13

2.6 Scratch 14

3 Technologies **16**

3.1 Eclipse Modeling Framework 16

3.2 Xtext 16

3.3 Xtend 16

3.4 Yakindu Statechart Tools 17

3.5 Webots 17

4 Implementation **18**

4.1 General approach 18

4.2 Yakindu Statechart Tools 18

4.2.1 Language 19

4.2.2 Grammar 20

4.2.3 Code generation 21

4.3 Webots 21

4.4 Extending the language 22

5 Results **27**

5.1 Line follower example 30

6 Discussion **33**

6.1 Webots 33

6.2 Complexity 33

6.3 Yakindu Statechart Tools 34

6.4 Visual and textual components 34

6.5 Event-based vs cycle-based execution 35

7 Conclusion and future work **36**

7.1 Future work 36

References **37**

Appendix **39**

A Complete Yakindu Statecharts grammar 39

List of Figures

1 A simple state machine [4] . . . 3

2 A simple example of clustering [4] . . . 4

3 Orthogonal state in statechart. [4] . . . 5

4 The orthogonal state from 3 modelled without orthogonality. [4] . . . 6

5 Economical representation of arrows [4] . . . 7

6 Different ways to represent a condition entrance [4] . . . 7

7 Timeout of a state [4] . . . 8

8 A simple state machine [4] . . . 9

9 Structural model of rFSM [7] . . . 11

10 Basic state machine classes of RAFCON [9] . . . 14

11 A sample scratch script [10] . . . 15

12 Overview of Yakindu Statecharts language [15] . . . 19

13 Grammar of StatechartScope in Xtext . . . 20

14 Grammar of ScopeDeclaration in Xtext . . . 20

15 Ecore model of Yakindu Statechart Tools type system . . . 21

16 Nodes of a robot in Webots . . . 22

17 Some methods of the class DistanceSensor . . . 22

18 Overview of the compilation process . . . 23

19 Robot definition section . . . 23

21 Robot element class . . . 24

20 Robot scope parsing . . . 24

22 Generating functions to identify each element . . . 25

23 Generating robot functions . . . 25

24 Main method . . . 26

25 Definition sections . . . 27

26 Main statechart region 28
27 Function that is executed upon entering the state 'Drive' 29
28 Function that is executed upon entering the state 'Turn' 29
30 Green lines showing the infrared sensors on the robot 30
29 Condition checked on each cycle while in the state 'Drive' 30
31 Picture of the robot following the line in Webots 31
32 Definition section for line follower example 31
33 Main statechart region for line follower example 32

1 Introduction

Robot programming is a very complex task. It requires advanced programming skills, and intricate knowledge of the robot itself. This makes it very difficult to get into as a novice. It is a considerable effort to learn both programming and robotics, which means it is accessible only to those who are willing to dedicate a lot of time to learn this skill.

The world is becoming increasingly technological, and the demand for software engineers is increasing [2]. The world is becoming increasingly dependent on software, and there is constantly software that needs to be created or updated. An increasing number of software applications are written by people without formal software development training [3].

Visual programming languages are programming languages where the program is expressed graphically rather than textually. They are commonly used educationally, with languages such as scratch, as well as to spark interest for programming. They simplify the process of learning programming by making the behaviour of programs easier to visualize, and reducing the need for deep syntactical knowledge. Though they are mostly used for education, they are also used for real-world applications.

Statecharts are a visual formalism for complex systems. It was invented by Harel to address the weaknesses of traditional state machines [4]. Over the years, statecharts has been included and adapted in many projects and variants. One variant has become part of UML, yielding a rise in popularity.

Multiple researchers have succeeded in using statecharts for robot programming. In these cases statecharts are used as a visual programming language, usually combined with a conventional programming language. Statecharts are then handling the structure of the program, while the conventional programming language handles the underlying behaviour.

This project looks at new visual programming approaches aimed at programming robots. For this purpose, statecharts are utilized, as it is well-suited to model complex, reactive systems. It is natural to use in such a situation since it uses boxes and arrows that make it easy to understand the behaviour and flow of a program. The aim of this is to reduce the complexity of programming robots, and to make it more available to those with little programming knowledge.

The overarching goal of this project is to explore the usage of visual programming for robot applications. This will include creating a simple visual programming language that can compile to a robot controller that can be run in the Webots robot simulator.

This thesis will first look at previous work done on visual programming as well as robot programming. It will then introduce the technologies used in this project. Then the implementation of the visual programming language is explained. The results of this are then presented. Then these results are discussed, before finishing with a conclusion and a discussion of future work.

2 Previous work

2.1 Visual programming

Visual programming languages allow the programmer to create applications by manipulating visual objects. This is usually done with a combination of boxes and arrows, where the boxes represent an action, and the arrows represent a transition between these actions. Typically, this is done to make programming more accessible and easier to understand for people who do not have a comprehensive understanding of the field.

Repenning [5] identifies three main cognitive challenges when it comes to programming. The first one is syntactic challenges. They deal with arranging the components of a programming language into well-formed programs. Visual languages aim to reduce the syntactic challenges by removing the need for deep knowledge of the syntactic structure of the language. Since less text is used there is less room for syntactic errors. Secondly, semantic challenges are about the user comprehending the meaning of the program. And finally, pragmatic challenges deal with the practical concerns of programming languages. This can include understanding what programs do in specific situations.

2.2 Statecharts

Finite-state machines are widely used as a mathematical model of computation. A finite-state machine can be in one state at a given time, and changes state based on the input it receives. Even though they are commonly used in many different fields, they are severely underpowered to model complex, reactive systems. This is mainly due to the number of states growing exponentially as the system grows in complexity. Figure 8 shows a very simple state machine. It consists of three states, and transitions between them.

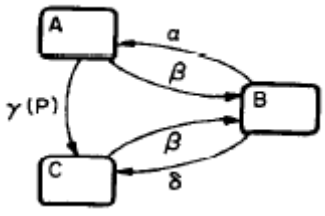


Figure 1: A simple state machine [4]

In 1987, Harel presented a broad extension to state machines, to enable it to model

complex systems [4]. This extension is called statecharts and adds multiple new features that give additional capabilities not possible in the original concept. Possibly the most important additions are orthogonality, clustering and broadcast communication. The sections below will highlight some of the most important features of statecharts, as it lays the foundation for the rest of the work in this project.

2.2.1 Clustering

Clustering creates a hierarchy of states, where two or more states are contained within the same superstate. If state A is a superstate containing state B and C, if the system is in state A it must be in either state B or C, but not both. In other words, this superstate is the XOR of the two states. This superstate can be seen as an abstraction of the substates it contains.

In statecharts, a transition leaving a superstate represents a transtition leaving all substates. This is a very important aspect of statecharts, as it is one of the main ways it economizes arrows. Arrows represent transitions in both finite-state machines and statecharts. One of the reasons that regular state machines are underpowered to model complex systems, is that the number of states and arrows simply grow too large. With clustering and clever ways to bundle transitions together, statecharts are able to reduce the number of arrows significantly. This enables it to model larger systems without becoming too cluttered for human comprehension.

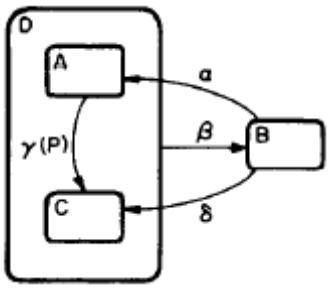


Figure 2: A simple example of clustering [4]

Figure 2 shows a simple example of clustering. State A and C are contained within state D. The arrow labeled β show a transition leaving state D, which then also represents leaving state A and C.

2.2.2 Orthogonality

Orthogonality is another feature introduced by statecharts to simplify the modeling of complex systems. An orthogonal state also contains multiple substates, just like superstates in clustering. The difference is that to be in an orthogonal state the system must be in all of its substates. This means that the system can be in multiple states at the same time, enabling concurrency. Orthogonal states severely reduce the amount of states needed to model complex systems. This is clearly illustrated by comparing figure 3 and figure 4. Figure 3 shows a simple orthogonal state with 2 components containing 2 and 3 substates respectively. Figure 4 shows this same state modelled without ortogonality. This system contains 6 states, since it contains all combinations of the substates. This is not a huge deal in this simple system, but if each component in the orthogonal state contained a thousand states, the number of states in the system without orthogonality would reach one million.

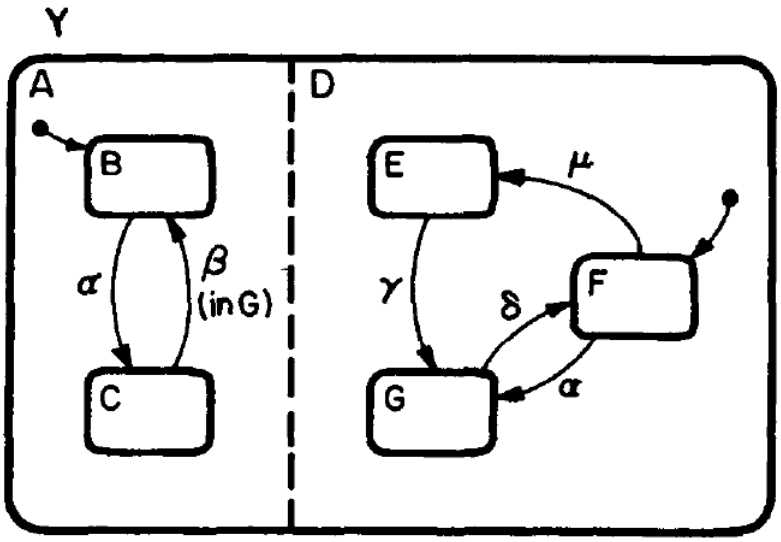


Figure 3: Orthogonal state in statechart. [4]

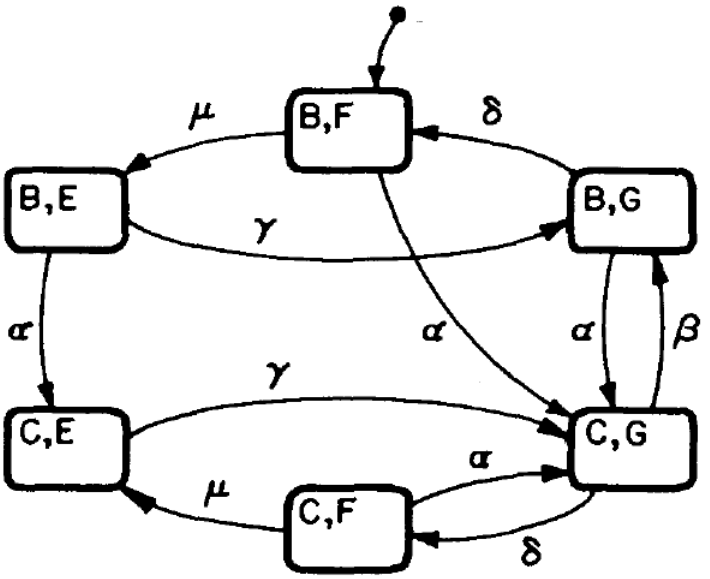


Figure 4: The orthogonal state from 3 modelled without orthogonality. [4]

2.2.3 Default states

Default state refers to one state being entered unless otherwise specified. If we have a state A containing substates B and C, and B has a default arrow pointing at it, it will be entered when entering A, unless it is specified that C should be entered. This can simplify the model, as some transitions can be replaced by default states. It can also make it easier for humans to understand the flow and logic of the system.

2.2.4 History

History is another way to determine which state to enter without the use of explicit arrows. It is represented with an "H", and specifies that the most recently visited state should be entered. It can also be coupled with a default arrow, which means that if the system has not been in this state before, and therefore has no history, it should enter the default state.

Specifying "H" applies the history feature just to the current level. It is also possible to apply this feature to this level as well as all levels below. This is specified by an asterisk: "H*". To have the history applied to some levels below, but not all, one would have to specify it at the levels it should apply to.

Statecharts also provide two actions clear-history(state) and clear-history(state*). The first action is used to clear the history of a state, while the second one is used to clear the history of a state as well as all levels below the state. When these are

used the history is forgotten, and default states will be used.

2.2.5 Economizing of arrows

Statecharts allow arrows to represent transitions with common sources, targets or events to merge. This results in a less cluttered, easier to understand graphical model. However, this functionality should be used with some caution. If one is not careful, subtle contradictions can occur. One example of this is shown in the rightmost example of figure 5: one arrow splits into two arrows. In this case it is unclear if arrow α or arrow β should be chosen.

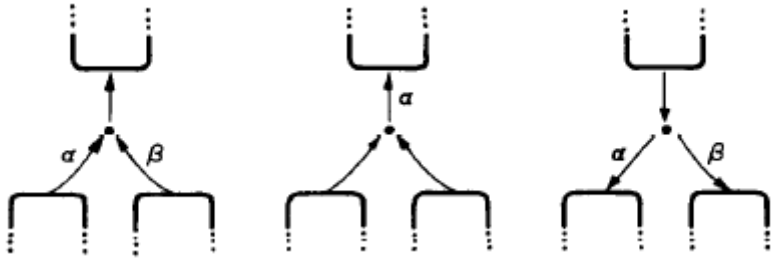


Figure 5: Economical representation of arrows [4]

2.2.6 Condition and selection entrances

Condition and selection are two ways to enter substates that provide more possibilities than a single arrow. They are similar to history in that enable entering a given state based on some criteria.

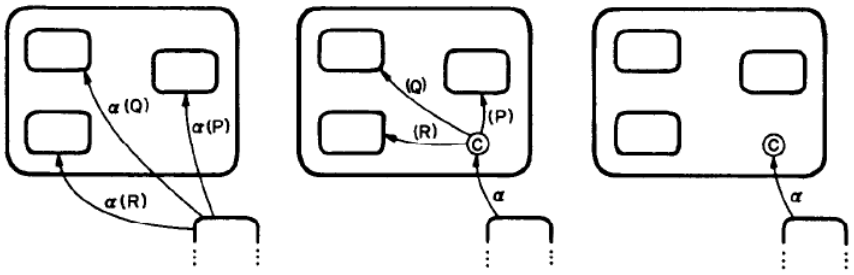


Figure 6: Different ways to represent a condition entrance [4]

The condition entrance enables the programmer to specify a condition which determines which state to enter. The programmer can choose how to represent this condition, as illustrated in figure 6. The decision is typically based on how complex

the condition in question is. If the condition is to complex to be effectively visualized in the chart, the full details should be provided separately. The rational is to keep the graphical visualization clear and concise, while enabling the opportunity for advanced functionality.

The selection entrance determines which state to enter based on the value of a generic event. Given a number of states, the event would contain a value which chooses between the states. This makes the event a selection that chooses between some number of clearly defined options.

2.2.7 Delays and timeouts

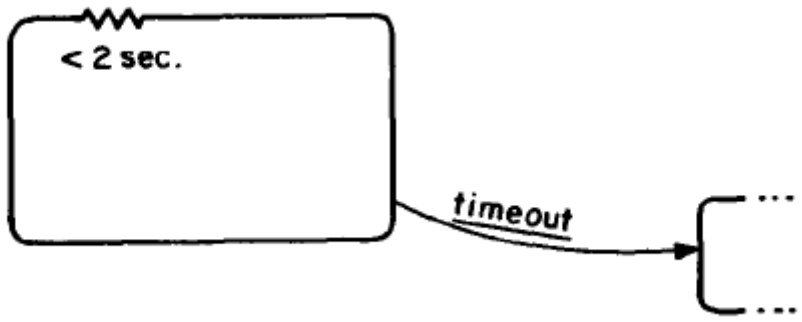


Figure 7: Timeout of a state [4]

Timeouts trigger an event a specified time after the occurrence of an event. This is specified using the expression 'timeout(event, number)'. Statecharts also provides a simple way to express that a state will be left after a given time period. Figure 7 shows that this is represented with a squiggle and a time condition. This means that the system cannot linger in this state for longer than 2 seconds. It is also possible to use lower bounds, specifying that the system has to remain in a state for a given amount of time.

2.2.8 Unclustering

Statecharts are created with the use of computerized graphics in mind, which was quite a new idea at the time of the original paper [4]. The paper puts great emphasis on using zooming and similar functions to enable the programmer to express the general layout of the system, as well as the details of each part. In other words, one can zoom out to see an overview without seeing the details, but zoom in to see the details of each part.

2.2.9 Actions and activities

Pure statecharts represent the control part of the system. To make this system fully functional we need some way to generate events, in addition to reacting to them. This is where actions come in. Actions are expressed with the following notation:

$$".../S" \tag{1}$$

This notation is then attached to the label of a transition, where S is the action. The dots in the notation are replaced by the event triggering the action. So if event α triggers action S, the notation would be α/S.

In statecharts, actions are instantaneous, and should be considered as occurences which take zero time. They are not long-running, and only use the time it takes to actually send the signal. In addition, actions are not solely used for triggering other events in the system, but could be the output of the system as a whole.

In addition to actions, activities are introduced. Where actions are instantaneous and ideally take zero time, activities always take at least some time. This allows computations to be done, and other functionality can be implemented. To start and stop activities we require two special actions: $start(X)$ and $stop(X)$, where X is the given activity. To start a given activity upon entering a state, $start(X)$ could be attached to the entering transition. In addition to this the condition $active(X)$ can be used to check if a given activity is running.

2.2.10 Broadcast

Statecharts implement broadcast communication. This means that if a state generates an event, all other states in the chart sense this event. This event can then trigger another event in another part of the system.

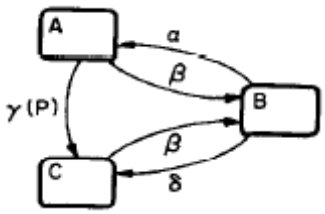


Figure 8: A simple state machine [4]

2.2.11 Problems with statecharts

All the projects discussed in the sections below, while basing themself on statecharts, make modifications to create their own version. They change how some features work, and remove other features entirely.

One common criticism of statecharts is the broadcast feature. This feature triggers events in different parts of the system than from which it originated. This eliminates the concept of modularity, which is commonly employed in software development. Modularity is commonly used both for reusability of components and ease of understanding of the system. Many states that broadcast events make the relationships between different components difficult to understand, as well as making systems more error prone.

Breen [6] argues that statecharts are too subtle. One of his arguments is that it can be difficult to understand all interactions between states, as well as the implications of an event, which makes it difficult to notice errors and inaccuracies. He also argues that the clustering and abstractions that can be done make it difficult to understand the entire system, as well as the details.

2.3 Coordinating robotic tasks and systems with rFSM statecharts

rFSM [7] stands for restricted finite state machine, and is a minimal subset of UML2 and Statecharts. It is used to simplify the programming of robot programming, and is based on the separation of four concerns:

1. Communication: how entities communicate
2. Computation: defines functionality
3. Configuration: how computations are configured, and how computations communicate
4. Coordination: managing individual entities so the system as a whole behaves as specified

The paper argues that robotic software frameworks focuses on communication, computation and configuration, but forgets to focus on coordination, which is often entangled in computation and communication. This can lead to problems for both

reusability and robustness, as the code governing the coordination between components gets mixed in with the functionality of the components. This could make it harder to reuse this functionality in similar components. rFSM therefore introduces a separate coordination entity to handle the coordination between components.

rFSM bases itself on the widely used semantics of UML state machines. The focus is on being composable, to support reuse of coordination component, and of existing models within each other. It consists of only three elements: states, transitions and connectors.

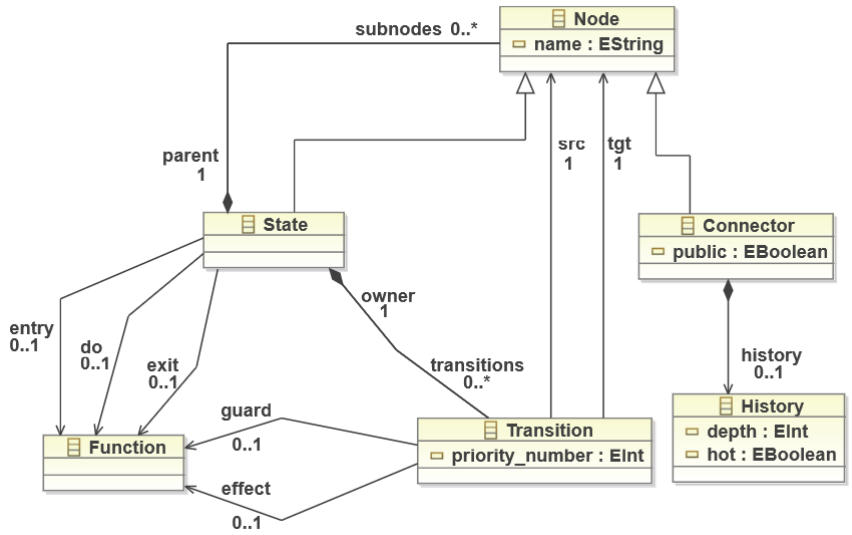


Figure 9: Structural model of rFSM [7]

States can be either a composite state or a leaf state. A composite state is a state that contains child nodes, while a leaf state does not. In rFSM only a single leaf state can be active at once. This is an important point, because it means that rFSM does not utilize the parallel states that statecharts introduced, and therefore does not have concurrency in and of itself. The reason for this decision is that parallelism requires making a number of platform-specific assumptions about the underlying concurrency mechanism. This does not work well for reusability.

Instead of orthogonal states defined in the statechart formalism, rFSM takes a loosely coupled approach where multiple state machines can be used on different machines over a distributed network. Using multiple hosts restricts the communication of these parallel components to a clearly defined interface. This improves many of the issues with the parallelism included in the statecharts formalism.

rFSM has *entry* and *exit* behaviours. These behaviours are executed when entering or and leaving a state. In addition to this, the *do* activity is executed while the

system is in a given state. The *do* activity is restricted to leaf states, and is therefore not permitted in composite states.

The rFSM state machine works by calling the *step* function. This function advances the machine one step. In doing this, it retrieves all events that happened since the last time *step* was called. The system then searches for a transition triggered by an event, starting from the top of the hierarchy. The first transition enabled by one of the events found in this search is executed, and the system searches no further. Transitions further up the hierarchy therefore have a higher priority, and the execution of the system becomes deterministic.

2.4 The ArmarX statechart concept: graphical programming of robot behavior

ArmarX statecharts [8] are similar to rFSM. The main difference is that while the separation of coordination and computation is enforced in rFSM, it is only encouraged in ArmarX.

ArmarX enables runtime-reconfigurability, as well as no need for recompilation on layout changes. This simplifies the process when making changes to a system, as the time needed to test the changes is greatly reduced. This is done by enabling the programmer to define the statecharts in configuration files. Since these files do not have to be compiled, they can be changed at runtime. It also enables runtime introspection, which makes the debugging process significantly easier.

The aim of ArmarX is to reduce the complexity of programming robotics software. It attempts to achieve this by separating the structure of the application from the behaviour. This is done by using statecharts as the structural component, while enabling the programmer to use conventional programming languages to specify the behaviour. The paper presenting ArmarX lists 5 key principles:

1. Modularity
2. Reusability
3. Runtime-reconfigurability
4. Decentralization
5. State disclosure

States in ArmarX has explicitly specified input and output, and no direct interaction is allowed between sub-states of different parent states. This is to ensure modularity,

which in turn improves reusability. Since each state has explicitly specified inputs and outputs, it creates a clearly defined interface which other states can interact with. This means that states are easy to reuse in multiple parts of the system.

ArmarX includes the same functionality as rFSM when it comes to running the system on multiple hosts. In contrast to rFSM, though, ArmarX allows some parallelism on one host, but still not in the same way has the original statecharts. In ArmarX, each active state can contain an asynchronous user code function. This enables some form of orthogonality, but still without allowing parallel states as in the original specification.

2.5 RAFCON

RAFCON stands for RMC advanced flow control and is presented in the paper "RAFCON: A Graphical Tool for Engineering Complex, Robotic Tasks" [9]. It is also made for graphical robot programming, but is more inspired by flowcharts than by statecharts. This was done because statecharts are inherently event-driven. The creators of RAFCON argues that event-driven systems are error-prone. They argue this because event-driven systems have to deal with issues such as event caching, event prioritization, event expiration and parallel event handling.

The paper also argues that statecharts have too many features. These features, while one one side making some work easier for the programmer, on the other hand increases complexity and error-proneness of the system. Since robotic programming is already a very complex field, one can argue that the focus should be on making it as easy as possible.

Figure [9] shows the basic state machine classes of RAFCON. RAFCON has three types of states: *ExecutionStates*, *HierarchyStates* and *ConcurrencyStates*. Execution states are leaf states which contain the functionality of the system, while hierarchy states are superstates containing child states.

Concurrency states are superstates where all child states are executed in parallel. This is essentially equvialent to the orthogonal states in statecharts. In contrast to rFSM and ArmarX, RAFCON chooses to implement this feature, and argues that it is crucial to applications needing several tasks to be accomplished at once.

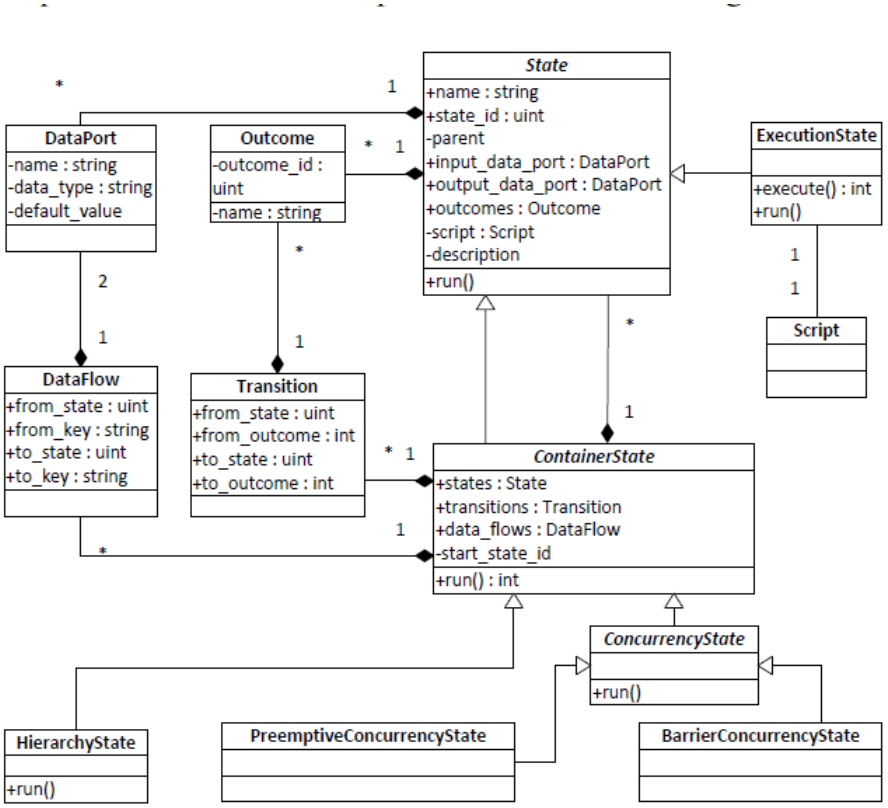


Figure 10: Basic state machine classes of RAFCON [9]

2.6 Scratch

When talking about visual programming languages it is worth mentioning Scratch. Scratch is a visual programming language intended for education [10]. The language is extremely easy to use, and is aimed mainly at children. It works by connecting different blocks together. Different blocks have different effects, which enables the user to create and execute programs. Even though the language is not really used for robot programming, it is relevant for this project in that it is intended for education, as well as simply sparking an interest in programming. Scratch being very easy to learn, as well as having a very simple user interface makes it extremely accessible to people with little experience in programming. Its over 80 million users [11] seem to indicate that this is an effective approach.

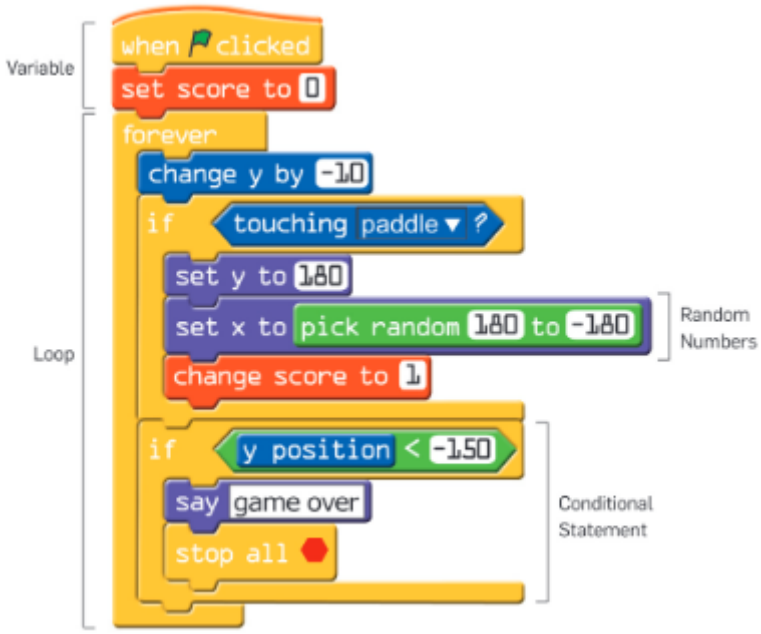


Figure 11: A sample scratch script [10]

3 Technologies

This section will introduce some of the different technologies that are utilized in this project.

3.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is a modeling framework and code generation facility [12]. It enables the programmer to build applications based on a data model. EMF produces Java classes from a model specified in XML Metadata Interchange (XMI). EMF works by specifying models in XMI (XML Metadata Interchange), and then producing a set of java classes for the given model. This removes the need for the programmer to manually create the classes from models, as EMF does it automatically.

A significant advantage of this appears when making changes to the model. The code can be regenerated, so any changes to the model is easily implemented in the code. This could be a significant amount of work when making changes to the model in later stages of development.

3.2 Xtext

Xtext is a framework for developing programming languages and domain-specific languages [13]. It is a part of the Eclipse Modeling Framework Project. Xtext provides a grammar language which can be used to specify a language. This enables Xtext to generate an ANTLR parser, in addition to classes for the model. Xtext provides a parser, linker typechecker and compiler. The Xtext grammar derives an Ecore model.

3.3 Xtend

Xtend is a Java dialect which compiles into readable Java code [14]. It is general purpose and high-level, and as it compiles to java code, it integrates with all java libraries. Xtend aims to provide a more syntactically compact and less verbose version of Java, in addition to providing some additional functionality. This includes operator overloading, type inference and extension methods. Xtend is mainly an object oriented language, but implements some features from functional programming. An example of this is lambda expressions. Xtend uses the same type system

as Java, and is statically typed.

3.4 Yakindu Statechart Tools

Yakindu Statecharts Tools is a tool for developing event-driven systems. It uses statecharts to graphically program reactive systems. It supports graphical editing of statecharts, and provides validation, simulation and source code generation of the models. Yakindu Statechart Tools uses a combination of graphical modeling and textual modeling.

3.5 Webots

Webots is a free and open-source robot simulator. It allows the user to add objects and robots to a simulated world. It includes many different prebuild robots and sensors, as well as actuators and objects. The user may also create their own models and add them to the simulation. The robots can be programmed using robot controllers. Webots supports robot controllers programmed in C, C++, Python, ROS, Java and MATLAB. Webots uses a fork of Open Dynamics Engine for collision detection and rigid body simulation. This enables it to simulate a realistic physics environment.

4 Implementation

This section will describe the implementation of the visual programming language. The implementation is based on the latest open source version of Yakindu Statechart Tools. How Yakindu Statechart Tools works will therefore first be discussed, before moving on to how it is extended in this project.

4.1 General approach

The aim of this project is to explore visual programming approaches for robots. Since finding a real robot well suited for testing proved to be difficult, as well as to ease the development process, it was decided to use a virtual robot simulator. The simulator used in this project is Webots, which was chosen based on its simplicity and ease of use.

The language itself is an extension of the language provided by Yakindu Statechart Tools. This supports programming with statecharts, and can compile down to several different languages. In this project the visual components compiles down to Java, which is then compiled to Java Byte code, which is then run by the robot simulator.

4.2 Yakindu Statechart Tools

Yakindu Statechart Tools is built upon the Eclipse IDE. Eclipse is a free and open-source IDE used for software development. Yakindu Statechart Tools is a large application, and this section will only discuss the parts that are relevant to this project. This will mainly include how it compiles statechart models to executable code.

As of version 4.0 the Yakindu Statechart Tools repository is read only, due to changes to the licensing. This means that later versions are not open source. This project is therefore based on the latest open source version.

4.2.1 Language

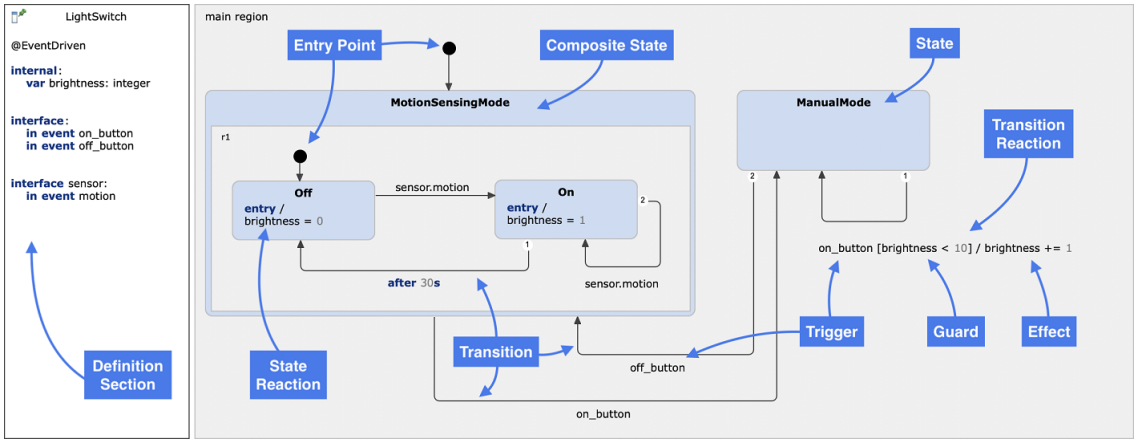


Figure 12: Overview of Yakindu Statecharts language [15]

Figure 12 shows an overview of the visual programming language available through Yakindu Statechart Tools. It shows most of the basic elements that can be used to create a program using statecharts.

The left side of the figure shows the definition section. In this section elements of the statechart can be defined, which includes variables, events and operations. These can be defined as interfaces, which can be exposed to and used by external code. The user can also specify whether execution should be event-driven or cycle-based. If event-driven is specified, the statechart will only respond to incoming events. If cycle-based is specified, the statechart will run continuously and process events as they are received.

The right side of the figure shows the main view the programmer will use to create the statechart. This is composed mainly of states, as well as transitions between states. The main region starts with an entry point which is the entry point to the statechart, and is accompanied by a transition which arrives at the first state that will be entered.

States and transitions define the behaviour of the program. These are organized in regions. Different regions are essentially different statecharts. This means that the programmer can specify different statecharts which can execute virtually concurrently. The program is always in a given state, and transitions change the state the program is in. Transitions are triggered by events, which can be raised outside the scope of the statechart, or raised by code inside the statechart.

Inside each state the programmer can add text-based code. This code can do several different things including raising events, checking conditions, and calling functions.

This text-based code enables adding complex functionality to the statecharts, and creating complete applications without the need of much external code.

The code inside each state is always tied to a trigger. Examples of such triggers are 'entry' and 'oncycle'. Code following 'entry' is executed upon entering the given state. Code following 'oncycle' is executed every cycle of the state machine while in the given state. This means that all execution of such code is triggered by some type of event, whether this event is raised inside or outside the statechart itself.

4.2.2 Grammar

The grammar of Yakindu Statechart Tools is written in Xtext. This parses the input and generates corresponding Java objects that is later used for code generation. Figure 13 and figure 14 shows the Xtext code for the grammar of 'StatechartScope', 'InterfaceScope' and 'ScopeDeclaration'. These together parse the input of an interface. The rule 'InterfaceScope' parses the declaration of the interface itself, with an optional name, while 'ScopeDeclaration' parses the fields of the interface.

```
StatechartScope:
    InterfaceScope | InternalScope | ImportScope;

InterfaceScope:
    {InterfaceScope} 'interface' (name=IDWithKeywords)? ':' (members+=ScopeDeclaration)*;
```

Figure 13: Grammar of StatechartScope in Xtext

```
ScopeDeclaration returns types::AnnotatableElement:
    {types::AnnotatableElement} annotations+=Annotation*
    (
        {VariableDefinition.annotationInfo=current}
        (const?='const' | 'var') ((readonly?='readonly')?) name=IDWithKeywords
        ':' typeSpecifier=TypeSpecifier ('=' initialValue=Expression)?
    |
        {EventDefinition.annotationInfo=current}
        (direction=Direction)? 'event' name=(ID | Keywords) (':' typeSpecifier=TypeSpecifier)?
    |
        {TypeAliasDefinition.annotationInfo=current}
        'alias' name=IDWithKeywords ':' typeSpecifier=TypeSpecifier
    |
        {OperationDefinition.annotationInfo=current}
        'operation' name=IDWithKeywords '(' (parameters+=Parameter (',' parameters+=Parameter)*)?
        (':' typeSpecifier=TypeSpecifier)?
    );
```

Figure 14: Grammar of ScopeDeclaration in Xtext

To generate Java classes for the parsed objects the Eclipse Modeling Framework is

used. Figure 15 shows the Ecore model for the type system of Yakindu Statechart Tools. This model is defined visually, but compiles to usable Java classes.

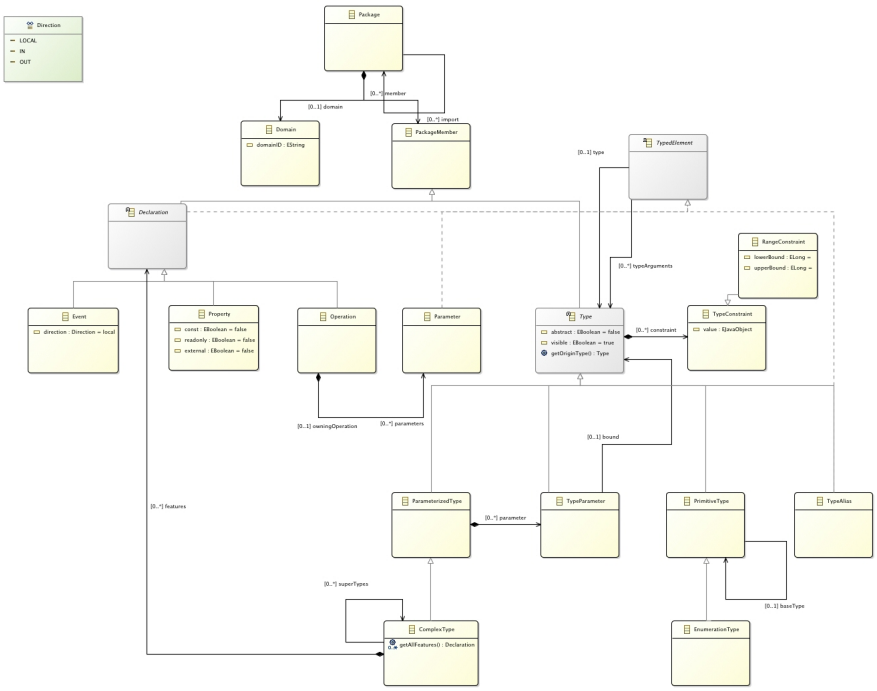


Figure 15: Ecore model of Yakindu Statechart Tools type system

Yakindu Statechart Tools can compile statecharts to many different languages, such as Java, Python and C. It uses the same grammar for all of these languages. Only the code generation stage differs in the compilation process for different languages.

4.2.3 Code generation

The code generation is written partly in Java and partly in the Java dialect Xtend. Xtend is used here as it is more concise and less verbose than plain Java. The code generation differs depending on the target language. The code generation stage takes the parsed objects and creates human-readable code in the specified language.

4.3 Webots

The implementation of this project uses the Webots API. Before describing the implementation it is necessary to take a look at how this API works.

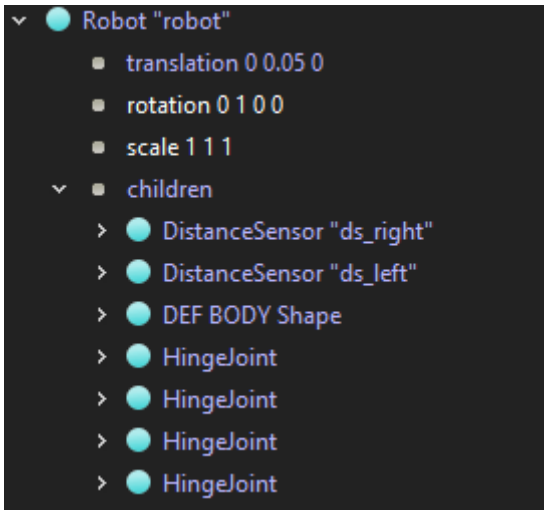


Figure 16: Nodes of a robot in Webots

A robot in Webots is composed of different nodes. A node can be the body of the robot, a sensor, a motor, or some other part of the robot. Figure 16 shows a robot with multiple nodes in Webots. These nodes correspond to different classes in the API, and can be programmed in a robot controller.

```
public class DistanceSensor extends Device {
  public void enable(int samplingPeriod);
  public void disable();
  public int getSamplingPeriod();
  public double getValue();
  // ...
}
```

Figure 17: Some methods of the class DistanceSensor

Figure 17 shows some of the methods available for the class 'DistanceSensor' in the Webots API. These classes are are utilized to create the robot controller during the compilation process.

4.4 Extending the language

This implementation is based on the Yakindu Statechart Tools language discussed in the previous section. That language is very general, and therefore needs more specification for the applications of this project.

Specifically, this project aims to produce code that can be run within the Webots

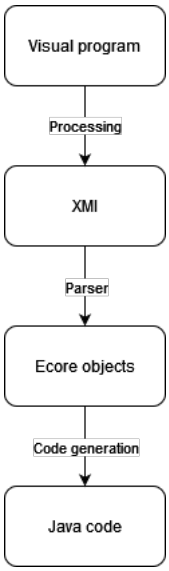


Figure 18: Overview of the compilation process

robot simulator. Webots provides its own API to control robots, and the language created here needs to use this API.

Figure 18 shows an overview of the compilation process. The programmer creates an application using the visual interface. This then gets processed to XMI which describes the different elements of the specified statechart. The parser then processes this data and creates Ecore objects. Ecore objects are Java object created by the Ecore model. In the code generation stage these objects are processed to create Java code.

```
robot ro
  in DistanceSensor ds_right
  in DistanceSensor ds_left
  out Motor wheel1
  out Motor wheel2
  out Motor wheel3
  out Motor wheel4
```

Figure 19: Robot definition section

The extension of the language relates mostly to the interactions with the different parts of the robot. To be able to program the robot the application needs to know what different inputs and outputs the robot has, i.e sensors and motors. This is done by extending the definition section. As figure 19 shows, the programmer can define a robot as well as the the elements it is composed by. This is separated into input and output elements. Input elements will typically be sensors that feed the application with some type of information, while the output elements typically are some type of motor that the application can control in some way.

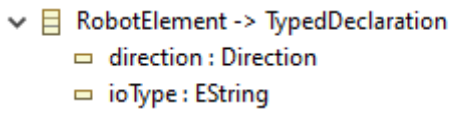


Figure 21: Robot element class

These elements have to be assigned a name that matches the respective elements name in Webots. This is to ensure that each reference to this element in the statechart code gets translated to a reference to the correct element in the compiled code.

```
RobotScope:
    {RobotScope} 'robot' (name=IDWithKeywords)? ':' (members+=RobotScopeDeclaration)*;

RobotScopeDeclaration returns types::AnnotatableElement:
    {types::AnnotatableElement} annotations+=Annotation*
    (
        {RobotElementDefinition.annotationInfo=current}
        (direction=Direction)? ioType=ID name=ID
    );
```

Figure 20: Robot scope parsing

The parsing of the robot elements is done in much the same way as interfaces, as shown in figure 20. First, one defines the scope with an optional name for the robot. Then, as many individual elements as necessary can be specified. Each element is defined by an optional direction (i.e. in or out), a type and a name. The type of the element has to match the type of the element in Webots, for example DistanceSensor or Motor.

```
protected def generateInit(RobotScope scope) '''
    private void initMotor(String name) {
        Motor m = getMotor(name);
        m.setPosition(Double.POSITIVE_INFINITY);
        m.setVelocity(0.0);
    }

    private void initDistanceSensor(String name, int TIME_STEP) {
        DistanceSensor ds = getDistanceSensor(name);
        ds.enable(TIME_STEP);
    }

    public void init(int TIME_STEP) {
        «FOR outputElement : scope.getRobotOutputElements.filter[ioType.equals("Motor")]»
        initMotor("«outputElement.name»");
        «ENDFOR»
        «FOR inputElement : scope.getRobotInputElements.filter[ioType.equals("DistanceSensor")]»
        initDistanceSensor("«inputElement.name»", TIME_STEP);
        «ENDFOR»
    }
'''
```

Figure 22: Generating functions to identify each element

Figure 22 shows the code that generates functions to initialize DistanceSensor and Motor elements. These elements needs some initialization to work properly, due to the way Webots work. Since each different input and output element works differently, there needs to be exlplicit code generation for each individual type of input and output element. This means that for each different type of sensor or motor one would have to manually write code generation for each one.

```
«FOR inputElement : scope.getRobotInputElements.filter[ioType.equals("DistanceSensor")]»
    public double get«inputElement.name»Value() {
        return get«inputElement.ioType»("«inputElement.name»").getValue();
    }
«ENDFOR»
«FOR outputElement : scope.getRobotOutputElements.filter[ioType.equals("Motor")]»
        public void set«outputElement.name»Velocity(double velocity) {
            get«outputElement.ioType»("«outputElement.name»").setVelocity(velocity);
        }
«ENDFOR»
```

Figure 23: Generating robot functions

Figure 23 show code generation for two function calls to the Webots API. This generates code allowing the programmer to access the functionality provided by the Webots API through the visual programming language. The code generation filters the elements of the robot by the field ”ioType”, which corresponds to the class it is in the Webots API. It then generates functions for this class which in turn calls some type of function defined in the Webots API, which in turn enables the programmer to get information from the sensors of the robot, or give instructions to the Motors.

```
public static void main(String[] args) {
  Statemachine stateMachine = new Statemachine();
  stateMachine.init();
  stateMachine.enter();
  while(true) {
    stateMachine.runCycle();
  }
}
```

Figure 24: Main method

Yakindu Statechart Tools is mainly intended to create state machines that are a part of a larger application. This means that by default it does not create an entry point, as it expects the programmer to create this separately. To make this implementation work on its own without the need for separate coding, a file containing a main method needs to be created during code generation. This main method is very simple: it creates and initializes the state machine, and then loops calling the runCycle() method.

This implementation is not event-driven, since Webots does not support this style and needs the controller to execute for each cycle of the simulation. This means that the state machine executes a cycle for each cycle of the simulation.

```
interface definitions:
  var counter: integer
  in event sensor_event
  in event turn_done

robot rob:
  in DistanceSensor ds_right
  in DistanceSensor ds_left
  out Motor wheel1
  out Motor wheel2
  out Motor wheel3
  out Motor wheel4
```

Figure 25: Definition sections

5 Results

The result of this project is a working version of the visual programming language capable of creating robot controllers for Webots. This language is not entirely visual, but uses a combination of visual and textual elements to create a stand-alone program. This combination allows the programmer to utilize visual elements to specify the flow of the application. Furthermore, textual commands can be used to specify the behaviour of the individual states of the application.

Initially, an interface needs to be created. This interface defines the different variables and events that can be used in the statechart. Figure 25 shows the definition of an interface and a robot. The interface defines variables and events that can be used in the statechart. Here, a variable 'counter' is defined, which is an integer. In addition two events are defined: 'sensor_event' and 'turn_done'. These events can then be used to trigger transitions between states.

The robot section defines the different elements the given robot is composed by. Figure 25 shows a robot composed by two distance sensors and four motors. These can then be accessed as objects in the statechart.

Figure 26 shows a simple example of a robot controller programmed with this language. The robot in this example is composed by two distance sensors and four wheels. The controller created by this statechart programs the robot to drive straight forward until it encounters an obstacle. It then turns away from the obstacle, and drives straight forward until the next obstacle.

The controller is entirely contained inside one region called 'main region'. This region is composed by two states: 'Drive' and 'Turn', representing the robot driving straight forward and the robot turning, respectively. The black circle in the top left of the region is the entry point. This is where the state machine is entered. The

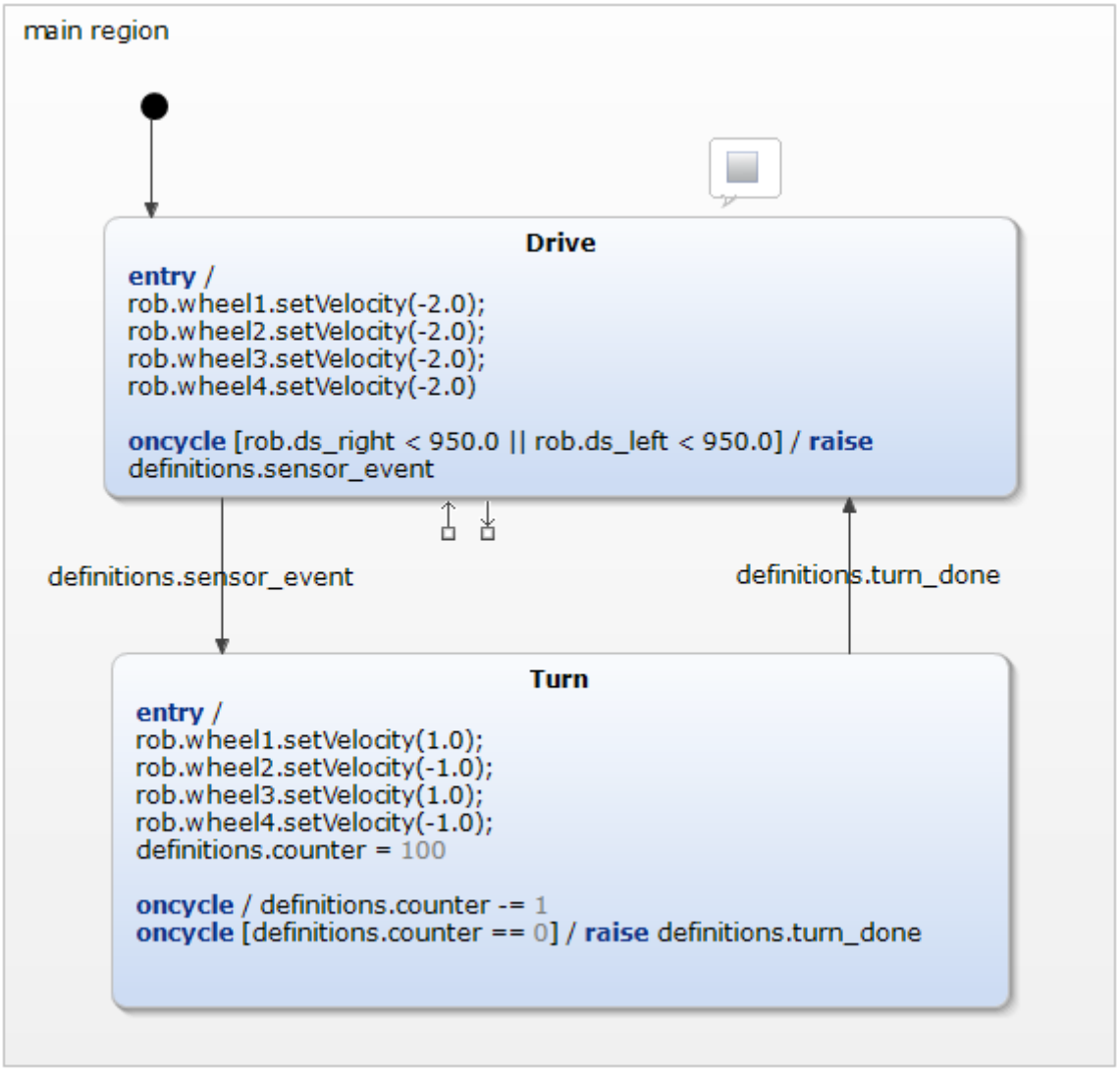


Figure 26: Main statechart region

arrow from the black circle then leads directly to drive, which is the first state that is entered.

Inside the state drive there is some code. The commands following 'entry \' is executed upon entering the state. Here the command 'setVelocity(-2.0)' is performed on all the wheels defined earlier. 'oncycle' defines commands that are executed every cycle of the state machine. The conditions inside the brackets are called guard conditions, and stop the commands following '\' from being executed unless the conditions inside the brackets evaluate to true. In this case, if the condition inside the bracket evaluate to true, the event 'sensor_event' is raised.

Between the states 'Drive' and 'Turn' there are arrows marked with labels. These arrows are transitions that are taken when the event matching the label is raised. When the event 'sensor_event' is raised while in the state 'Drive', the state machine will transition to the state 'Turn'.

This statechart is then compiled to readable Java code. Figure 27 shows the code that is executed upon entering the state 'Drive'. This is a relatively straight forward translation of the commands in the statechart.

```
/* Entry action for state 'Drive'. */
private void entryAction_main_region_Drive() {
    robotController.getwheel1().setVelocity(-2.0);

    robotController.getwheel2().setVelocity(-2.0);

    robotController.getwheel3().setVelocity(-2.0);

    robotController.getwheel4().setVelocity(-2.0);
}
```

Figure 27: Function that is executed upon entering the state 'Drive'

```
/* Entry action for state 'Turn'. */
private void entryAction_main_region_Turn() {
    robotController.getwheel1().setVelocity(1.0);

    robotController.getwheel2().setVelocity(-1.0);

    robotController.getwheel3().setVelocity(1.0);

    robotController.getwheel4().setVelocity(-1.0);

    sCIDefinitions.setCounter(100);
}
```

Figure 28: Function that is executed upon entering the state 'Turn'

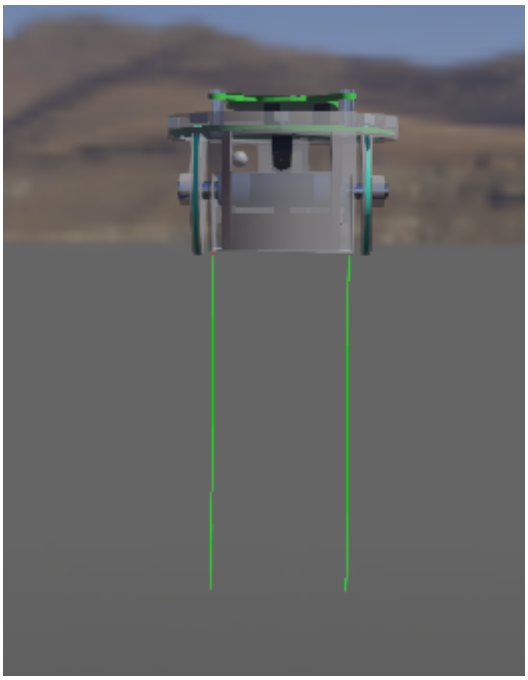


Figure 30: Green lines showing the infrared sensors on the robot

```
if ((robotController.getds_rightValue()<950.0 || robotController.getds_leftValue()<950.0)) {
    sCIDefinitions.raiseSensor_event();
}
```

Figure 29: Condition checked on each cycle while in the state 'Drive'

Figure 29 shows the compiled code for the condition expressed after 'oncycle' in the state drive. This reads the values from the sensors and checks it against the given value. If the condition evaluates to true the 'sensor_event' event is raised.

5.1 Line follower example

This section will take a look at an example of programming a robot to follow a line. This will use a simple robot with two wheels, and to infrared sensor pointing down beneath it (shown in figure 30).

Figure 31 shows the robot following the line in Webots. This environment needs to be set up before the robot controller can be run. The robot will try to follow the line using the two infrared sensors. Using a black line on a white background, the sensors return different values depending on the color of the material it detects. When the left sensor detects that it is no longer over the line, the robot should turn to the right, and when the right sensor detects that it is no longer over the line, the robot should turn to the left.

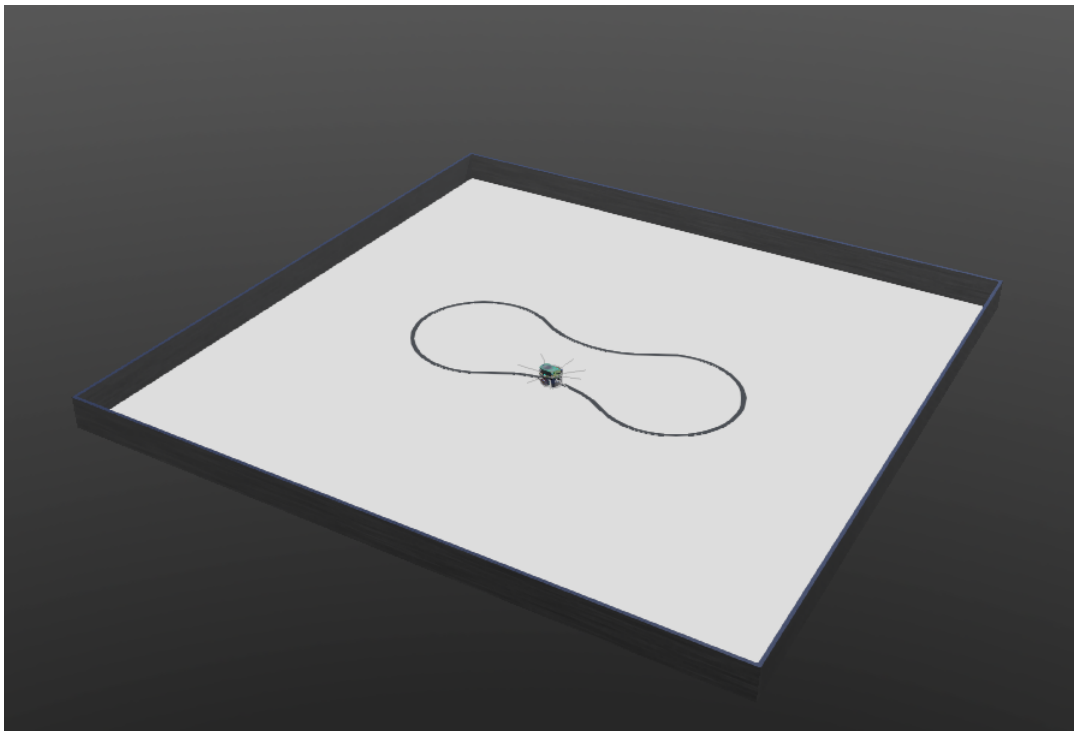


Figure 31: Picture of the robot following the line in Webots

```
interface:
   const speed: real = 1.55
   in event turn_left
   in event turn_right
   in event stop_turning

robot rob:
   in DistanceSensor irRight
   in DistanceSensor irLeft
   out Motor leftMotor
   out Motor rightMotor
```

Figure 32: Definition section for line follower example

Figure 32 shows the definition section, defining one interface and one robot. The interface defines one constant and three events. The constant controls the speed of the robot, while the events define the transitions between the states. The robot section defines the elements of the robot corresponding to how they are defined in Webots.

The main section shown in 33 defines the behaviour of the robot controller. It contains three states: ”Drive straight”, ”Turn Left”, and ”Turn right”.

The ”Drive Straight” state sets the velocity of the motors, making the robot drive forward on entry. For each cycle it checks the value of both sensors, and if either of them makes the criteria defined for not being over the line, it raises an event

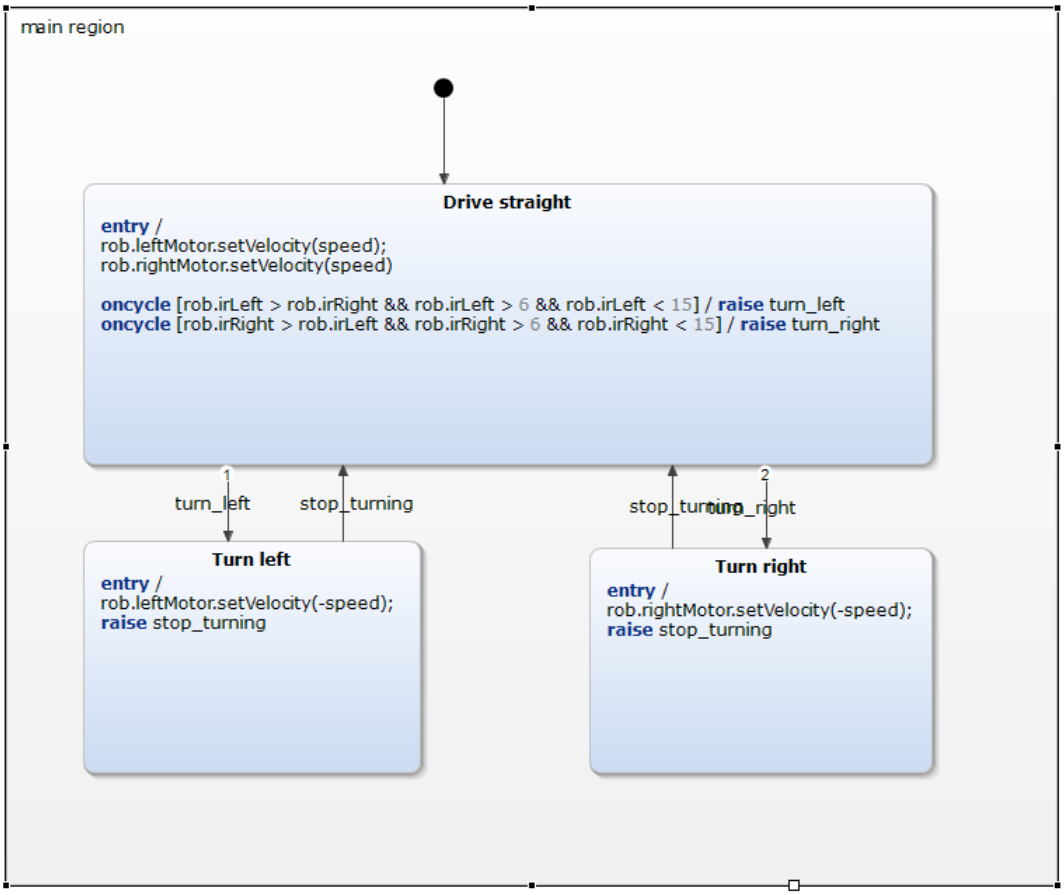


Figure 33: Main statechart region for line follower example

corresponding to the direction the robot should turn.

The ”Turn left” and ”Turn right” states are very similar. On entry they set the velocity of the wheel so that the robot turns, and then they return back to the ”Drive straight” state.

6 Discussion

This report has described the implementation of a visual programming language for robots. The language is relatively simple, and can be used to create robot controllers for the Webots robot simulator.

6.1 Webots

The goal of this project was to explore visual programming approaches for programming robots. As it was difficult to get a well-suited robot to use in this project, it was decided that using a robot simulator was the best alternative. This resulted in choosing Webots as a simulator to test the language. When creating robot controllers for Webots it is necessary to use the API they provide to get information, and manipulate the behaviour of the robots. This means that the compiled result of the visual programming language needed to utilize this API. As a result of this, the language is very tied to the API of Webots, and not of very much use outside this environment.

To adapt this language to other applications, for example embedded C code, the code generation would need to be changed. The grammar and the parser should stay the same, but the code generator needs to generate code that is compatible with the target platform. Yakindu Statechart Tools provides a compiler that compiles statecharts to C code, so only the additional robot specific code generation would need to be added.

6.2 Complexity

Complexity is somewhat relative and difficult to measure. However, there are some indications that this visual programming language is easier to use for novice programmers than using any other language with the Webots API.

Firstly, there is less text to write, meaning that there is less syntax to learn before one is able to start programming. Learning the syntax of a programming language can be a big road block when wanting to program.

Using boxes and arrows makes it easy to visualize how the program will behave in a given situation. Furthermore, states can be labeled in a way that makes it clear what happens when the program is in a given state. Together, this makes it very intuitive to program with Separation of Concerns, even for people who have little experience with this approach to problem solving.

This visual language is, however, not completely free from complexity. Since the language uses textual components, some syntactic knowledge is required. Some knowledge of programming conventions is also helpful, such as using '.' to access the fields of an object, or using parenthesis to call a function.

Moreover, some knowledge of the workings of statecharts is also required. The boxes and arrows are easy to understand, but the usage of more complex features such as history, choices and synchronization requires some understanding of how these work. It is possible to create simple application without these features, but creating something more advanced might require the use of them.

However, since the simple applications have very little need for these features, it can be argued that this gives the user more time to acclimate to the language before more knowledge is needed to create more complex applications. This means that it is easy for novices to start creating, while still retaining the functionality to create more advanced systems. This results in great flexibility for both newer and more advanced users.

6.3 Yakindu Statechart Tools

The decision to base the implementation of this project on Yakindu Statechart Tools was made based on the knowledge that it is a complete, working IDE that utilizes statecharts for visual programming. It was the closest available application to the goal of this project. However, since it is a large and complex application, it proved difficult and time consuming to extend the functionality in a way suited for these applications. This was exacerbated by the use of technologies that are not widely used, such as Eclipse Modeling Framework and Xtend.

6.4 Visual and textual components

The language presented here uses both visual and textual components to program robots. The tradeoff between these is usually that visual components make the language easier to comprehend, but generally also limits functionality. Textual components requires more knowledge of the language, but at the same time offer more functionality and freedom when programming. This project aimed to keep all the functionality, while still simplifying the structure of the program with visual elements. It is, however, possible to replace some textual elements with visual elements to further simplify the language.

6.5 Event-based vs cycle-based execution

The current implementation of the language only works with cycle-based execution. This means that the state machine executes continuously, and processes events as they are received. An event-based execution scheme would be preferable, as the state machine could go to sleep while waiting for events. This was difficult to achieve with Webots, as the steps of the simulation is tightly connected to the steps of the robot controller.

7 Conclusion and future work

This thesis has presented a visual programming language capable of programming robot controllers run in the Webots robot simulator. The visual programming language uses a combination of visual and textual elements that create a stand-alone application without the need for any external coding.

The language reduces complexity compared to using any of the other languages compatible with the Webots API. It necessitates less knowledge of syntax compared to these languages. It is also easier to visualize the behaviour of the program, since it uses boxes and arrows that can be manipulated.

7.1 Future work

Future work for this project would be to adapt the code generation for the language to be able to test it on a real robot. This project has focused entirely on getting the language to work on Webots. Since Webots has a very specific API, the code generation would need to be rewritten to be compatible with other systems.

Since the goal of this project was to explore the visual programming approach, the language is not complete. It was not prioritized to implement all classes and functions from the Webots API. To make the language fully compatible with all different classes in the API, the code generation would need to be expanded to accommodate all of them. This is not difficult, but tedious and time consuming, while adding little value to this project.

References

[1] P. Bjørseth, 'Using statecharts for robot programming', Dec. 2021.

[2] '2022 state of software engineers'. (2022), [Online]. Available: https://hired.com/2022-state-of-software-engineers/.

[3] M. A. Kuhail, S. Farooq, R. Hammad and M. Bahja, 'Characterizing visual programming approaches for end-user developers: A systematic review', *IEEE Access*, 2021. DOI: 10.1109/ACCESS.2021.3051043.

[4] D. Harel, 'Statecharts: A visual formalism for complex systems', *Behavioral and Brain Sciences*, pp. 87–114, 1987. DOI: 10.1016/0167-6423(87)90035-9.

[5] A. Repenning, 'Moving beyond syntax: Lessons from 20 years of blocks programing in agentsheets', *Journal of Visual Languages and Sentient Systems*, vol. 3, pp. 68–91, Jul. 2017. DOI: 10.18293/VLSS2017-010.

[6] M. Breen, 'Statecharts: Some critical observations', 2006.

[7] M. Klotzbuecher and H. Bruyninckx, 'Coordinating robotic tasks and systems with rfsm statecharts', 2012.

[8] M. Wächter, S. Ottenhaus, M. Kröhnert, N. Vahrenkamp and T. Asfour, 'The armarx statechart concept: Graphical programing of robot behavior', *Frontiers in Robotics and AI*, 2016, ISSN: 2296-9144. DOI: 10.3389/frobt.2016.00033. [Online]. Available: https://www.frontiersin.org/article/10.3389/frobt.2016.00033.

[9] S. G. Brunner, FranzSteinmetz, R. Belder and A. Dömel, 'Rafcon: A graphical tool for engineering complex, robotic tasks', pp. 3283–3290, 2016. DOI: 10.1109/IROS.2016.7759506.

[10] M. Resnick, J. Maloney, A. Monroy-Hernández *et al.*, 'Scratch: Programming for all', 2009, ISSN: 0001-0782. DOI: 10.1145/1592761.1592779. [Online]. Available: https://doi.org/10.1145/1592761.1592779.

[11] 'Scratch statistics', [Online]. Available: https://scratch.mit.edu/statistics/ (visited on 8th Dec. 2021).

[12] 'Eclipe modeling framework', [Online]. Available: https://www.eclipse.org/modeling/emf/.

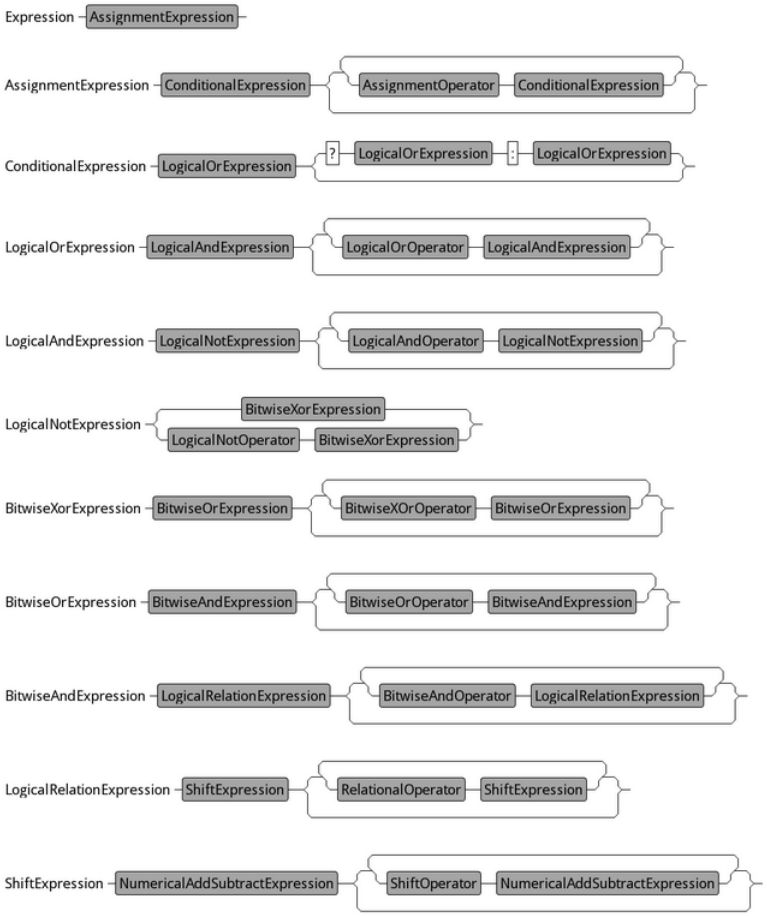
[13] 'Webots: Robot simulator', [Online]. Available: https://www.eclipse.org/Xtext/.

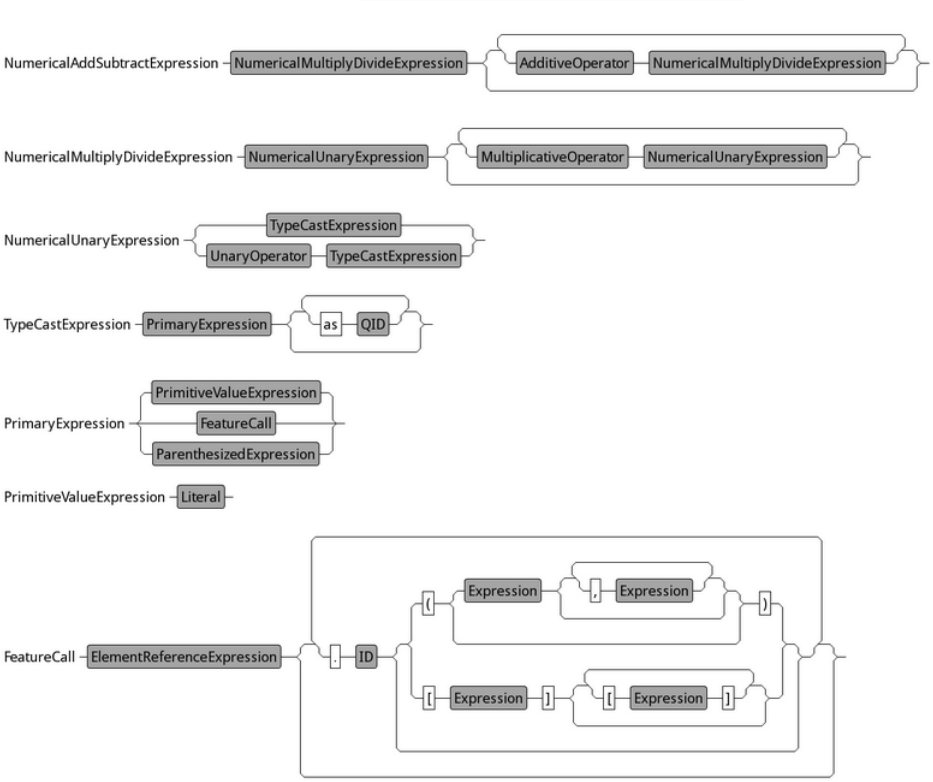
[14] 'Xtend', [Online]. Available: https://www.eclipse.org/xtend/.

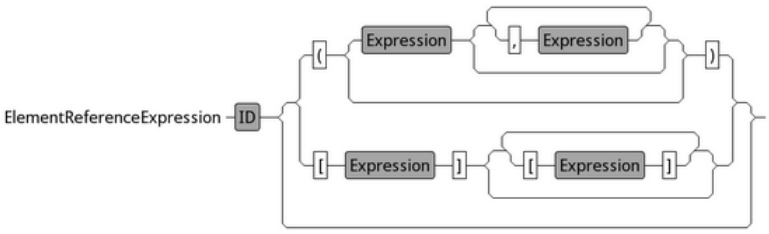
[15] 'Yakindu statechart tools quick reference', [Online]. Available: https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/quick_ref#quick_ref.

Appendix

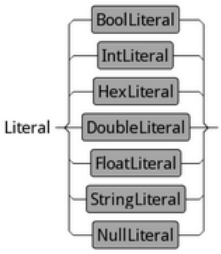
A Complete Yakindu Statecharts grammar



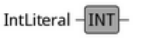

NumericalAddSubtractExpression
NumericalMultiplyDivideExpression
AdditiveOperator
NumericalMultiplyDivideExpression
NumericalMultiplyDivideExpression
NumericalUnaryExpression
MultiplicativeOperator
NumericalUnaryExpression
NumericalUnaryExpression
TypeCastExpression
UnaryOperator
TypeCastExpression
TypeCastExpression
PrimaryExpression
as
QID
PrimaryExpression
PrimitiveValueExpression
FeatureCall
ParenthesizedExpression
PrimitiveValueExpression
Literal
FeatureCall
ElementReferenceExpression
.
ID
(
Expression
,
Expression
)
[
Expression
]
[
Expression
]


ElementReferenceExpression
ID
(
Expression
,
Expression
)
[
Expression
]
[
Expression
]


ParenthesizedExpression
(
Expression
)

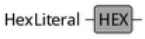

Literal
BoolLiteral
IntLiteral
HexLiteral
DoubleLiteral
FloatLiteral
StringLiteral
NullLiteral


BoolLiteral
BOOL

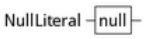

IntLiteral
INT

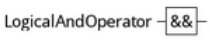

DoubleLiteral
DOUBLE


FloatLiteral
FLOAT


HexLiteral
HEX

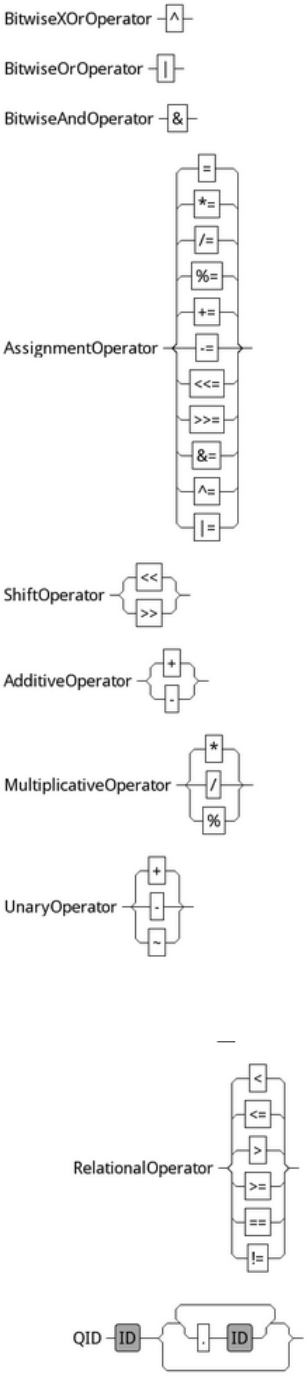

StringLiteral
STRING

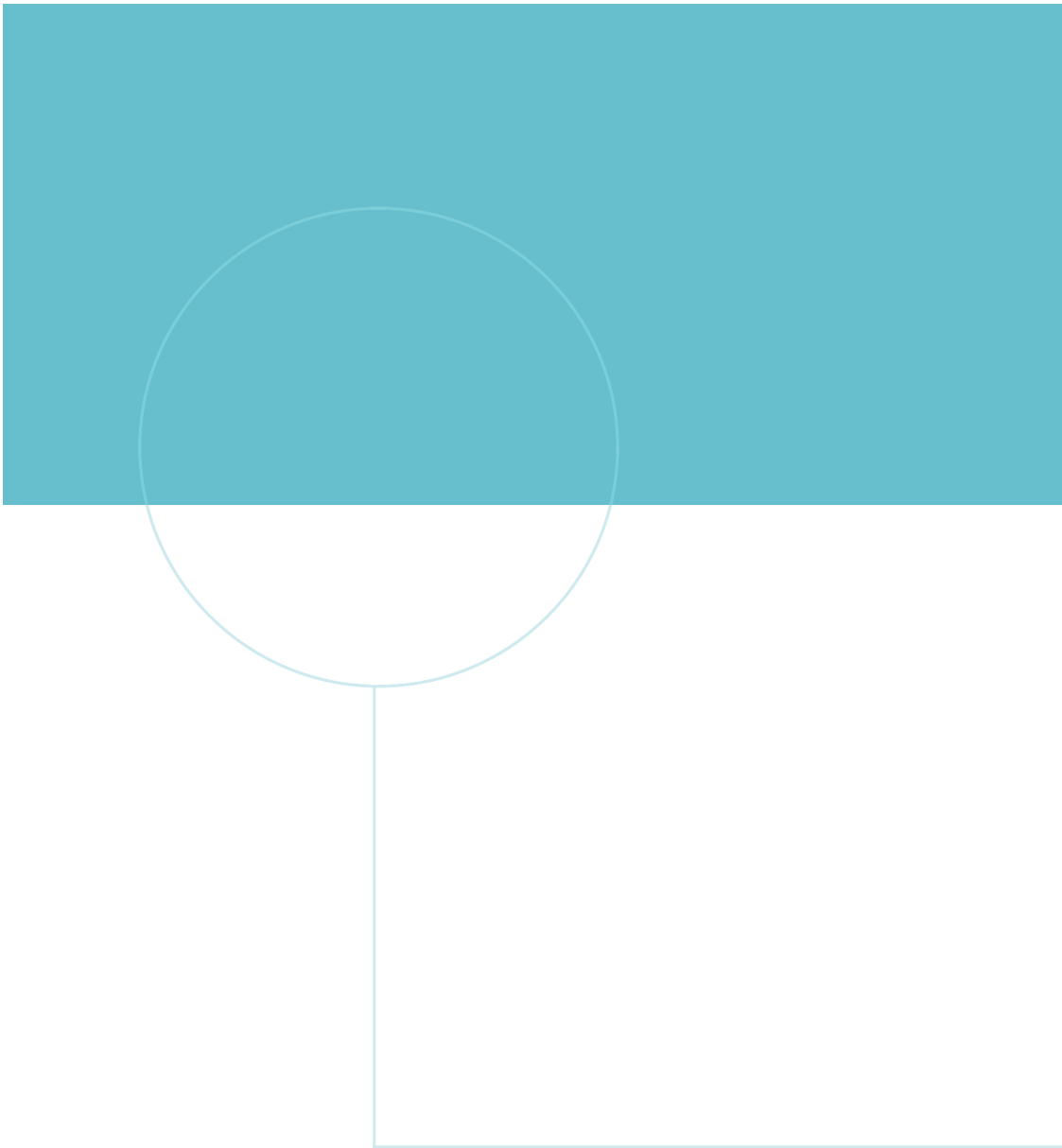

NullLiteral
null


LogicalAndOperator
&&


LogicalOrOperator
||


LogicalNotOperator
!


BitwiseXOrOperator
^
BitwiseOrOperator
|
BitwiseAndOperator
&
AssignmentOperator
=
*=
/=
%=
+=
-=
<<=
>>=
&=
^=
|=
ShiftOperator
<<
>>
AdditiveOperator
+
-
MultiplicativeOperator
*
/
%
UnaryOperator
+
-
~
RelationalOperator
<
<=
>
>=
==
!=
QID
ID
.
ID



NTNU

Norwegian University of
Science and Technology