# Reactive Programming using Procedural Parameters for End-User Development and Operations of Robot Behavior Control

March　２０１８

ERICH Floris Marc Arden

# Reactive Programming using Procedural Parameters for End-User Development and Operations of Robot Behavior Control

School of Integrative and Global Majors

Ph.D. Program in Empowerment Informatics

University of Tsukuba

March　２０１８

ERICH Floris Marc Arden

# Abstract

This thesis explores robot programming from the perspective of end-users. It takes under consideration not just the development of robot software, but also how the robot is operated. It is based on research in the fields of robotics, software engineering and human computer interaction. By combining these fields I hope to make a significant contribution to how humans interact with robots in the future.

Whereas in the past robots have been confined to factory environments, recently affordable robots such as NAO from Softbank Robotics and OP2 from ROBOTIS have become available for use in personal environments. These robots can be programmed to do various tasks desired by the user, such as performing demonstrations, communicating information retrieved from the Internet and interacting with objects in the environment. Even so, currently robots are not as ubiquitous in our environments as one might expect them to be.

Robots are deployed into an environment that they interact with through sensing and actuation. The robot senses for stimuli and actuates a response after processing the stimuli. The components of the robot that sense for stimuli are called sensors, and the components that actuate a response are called actuators. Sensors and actuators are physical components embedded inside of the robot.

To connect sensors and actuators, some processing is necessary. Because the robots that we consider in this thesis are of general purpose, the processing has to be reconfigurable, or programmable, using software. Hence the sensors and actuators of the robot have to be exposed within a software platform, and some means of programming has to be offered to reconfigure the processing done by the robot.

An example of a field in which such a platform might be used is Socially Assistive Robotics (SAR). In SAR, robots use their sensors and actuators to exhibit behaviors that model human social behaviors. Researchers hypothesize that using a robot in this way allows them to have great precision in teaching certain social behaviors [87]. Children also tend to react favorably to the toy-like appearance of the robot platforms considered in this thesis. Hence, therapists are interested in using robots as a therapeuthic instrument.

Currently a SAR experiment requires the following participating roles: A developer (either a person or a group of people), an operator (typically a single person) and a user (the therapist). Before the experiment, the therapist will communicate with the developer to design software that the therapist can use in the experiment. During the experiment itself there is also an operator present who controls the robot through the software written by the developer.

The need for communication between these three roles creates a long lead time before experiments can be performed. Details can get lost during communication,

which can cause bugs in the software. Ideally, the therapist would be able to develop and operate the robot software with minimal support from other people and groups. The therapist should in this case take on the roles of developer and operator.

Robot programming is however notoriously complex. Robots are distributed embedded systems operating in physical environments. Writing software for robots requires knowledge of physics, concurrency, computational constraints, how to deal with dynamic environments and how to ensure safety of both the user and the robot.

In Software Engineering, the problem of interaction between development and operations has recently gathered a lot of interest both in academic and in professional circles. Many organizations have adopted an approach called DevOps to try to improve the interaction between development and operations. DevOps is formally defined as a set of practices that tries to reduce the time required for changes in a system to evolve from development to operations, without negatively affecting quality [11]. In practice however organizations define DevOps as simply the interaction between development and operations [42].

In this research I aim to create a programming environment that allows end-users to develop and operate applications in Socially Assistive Robotics experiments. To that end I will answer the following research questions:

- What applications are used in SAR experiments?
- What other approaches exist to develop and operate robots?
- What are the components of the programming environment?
- How effective is the programming environment?
- How can the programming environment be used in different contexts?

To answer the question of what applications are used in SAR experiments, I performed a systematic literature review. Our search returned 24 papers, from which 16 were included for closer analysis. To do this analysis I used a conceptual framework inspired by Behavior-based Robotics. I was interested in finding out which robot was used (most use the robot NAO), what the goals of the application were (teaching, assisting, playing, instructing), how the robot was controlled (manually in most of the experiments), what kind of behaviors the robot exhibited (reacting to touch, pointing at body parts, singing a song, dancing, among others), what kind of actuators the robot used (always motors, sometimes speakers, hardly ever any other type of actuator) and what kind of sensors the robot used (in many studies the robot did not use any sensors at all, in others the robot frequently used camera and/or microphone). The results of this study can be used for designing software frameworks targeting Humanoid Socially Assistive Robotics.

Existing platforms exist that allow robots to be programmed. Major examples of this are the Robot Operating System (ROS), Choregraphe and Targets-Drives-Means (TDM). ROS is aimed towards robotics researchers and not end-users. Choregraphe is not explicitly aimed towards end-users, but aims to be a beginner friendly programming environment. TDM is aimed towards end-users, and a user-friendly programming interface for TDM has been developed for the Android smartphone operating system.

In ROS an application consists of nodes that are connected to other nodes through topics. Nodes can subscribe to topics, after which they will get notified when a node publishes to this topic. The structure created through this mechanism can be described as a graph in which the nodes and topics are vertices connected by edges signalling publishing and subscribing.

In Choregraphe an application consists of boxes that are connected to other boxes through ports. Boxes can have ports to start and stop the box, as well as data input and output ports. Boxes act like small machines that accept inputs, can do some processing and can produce output. Custom boxes can be build, but, as is shown in this thesis, even expert users of Choregraphe encounter difficulty in that task. It is especially hard to create boxes that combine different types of data.

Target-Drives-Means is a robot software architecture focused on end-user development, and it is based on the behavior-based robotics paradigm [7]. In TDM a program is a collection of behaviors. A behavior is a collection of action groups. A complex behaviour is a behavior that includes other behaviors. For complex behaviors the action groups of the included behaviors are joined to form the action groups of the complex behavior. An action group is a collection of action units. An action unit is composed of an action and a condition.

Platforms such as ROS, Choregraphe and TDM need facilities that go beyond connecting nodes (ROS), boxes (Choregraphe) or actions/conditions (TDM) in order to make them powerful enough for an end-user to develop new applications with. Users need to be able to develop custom nodes (ROS), boxes (Choregraphe) and actions/conditions (TDM). Currently the users can use the following for doing this:

- ROS: Developing nodes using supported programming languages such as C++ and Python.
- Choregraphe: Developing boxes using Python, subgraphs and specialized builders for movement and dialog.
- TDM: Developing actions and conditions using Python.

Both ROS and TDM require the user to have a good knowledge of programming to be a proficient user of the platforms. Choregraphe has more powerful facilities for end-users, however if a user wants to create behaviors, outside of movement and dialog, some experience of programming is necessary. I believe that an easier method for end-users to develop applications for robots exists. In this thesis I will present this and compare it with Choregraphe to show that this new method is indeed beneficial.

I call this new method Reactive Robot Programming (RRP). It is based on recent developments in software engineering, were programming languages are being extended with the capability to process streams of data [71]. Low level robot behaviors can be separated into three tasks: Sensing, Computation/Planning and Acting [7]. In RRP sensors and actuators can be connected using connectors and intermediary streams. The combination of sensors, streams, actuators and their connectors forms a graphical structure that I call the RRP Graph.

RRP is more usable than other solutions thanks to the following mechanisms:

- Streams are a natural way of reasoning about events happening in the real world.

- Seperating sensing, planning and acting makes it easy to reason about an RRP Graph.
- Having a small set of connectors to learn makes it easy to get started with using RRP.
- People are naturally visually oriented, a visual notation leverages this innate ability.

An important aspect of the RRP Graph is the usage of procedural parameters. In the RRP Graph, most connectors accept a procedure as a parameter. Practically, the connector in this case specifies what should be done, while the procedural parameter specifies how to do it. The following connectors are currently supported:

**map** Uses a procedural parameter to map inputs to outputs. Useful for example for doing a calculation on the input (e.g. a coordinate transform or distance calculation) or selecting a specific value from a structure (e.g. taking the value from a field of an object or looking up an index in an area).

**filter** Uses a procedural parameter to filter inputs that do not match a certain predicate. Useful for implementing range filters (e.g. filtering objects that are close or far) and category filters (e.g. filtering objects that have been seen before or are new; filtering faces that have been recognized or have not been recognized).

**timestamp** Adds a timestamp to the inputs. Useful for reasoning based on time (e.g. when combining two streams, ensure a maximum time difference between two events).

**sample** Samples an input at a certain rate. Useful for reducing unneeded computation and avoiding overloading an actuator.

**combineLatest** Uses a procedural parameter to combine inputs from multiple streams. Useful for sensor fusion.

**merge** Merges inputs from multiple streams. Useful when multiple streams produce the same type of output (e.g. when having redundant sensors such as two sonar sensors).

An interpreter called the RRP Runtime can read the RRP Graph and based on it initialize the sensors and actuators on the robot. The RRP Runtime will also ensure that data will flow from sensors to actuators through connectors as specified by the RRP Graph. As part of this the RRP Runtime removes the need for the user to take into consideration concurrency inside the software domain, however the user still has to consider concurrency in the physical domain (i.e. conflicts arising because certain sensors and actuators cannot be active at the same time due to logical constraints, for example an arm can only be in one position at the same time).

A Visual Programming Environment (VPE) enables the specification of programs using the RRP Graph by end-users. Procedural parameters can either be specified directly as a body of connectors, or can be stored as a helper in which case they can be reused by multiple connectors. The VPE is implemented as a web application to allow it to be used by multiple devices without needing seperate native applications. The VPE is a Single Page Application (SPA) and hence does not require reloading the page at any time during its usage. This is realized by using Web Sockets that maintain a client-server connection while the VPE is being used. Additionally the VPE has support for multiple users interacting simultaneously.

The VPE uses a graph database for storing the applications created by users. A graph database naturally supports the structure of the RRP Graph. Inside the graph sensors, actuators and streams are stored as nodes, while simple connectors are stored as edges. Connectors are considered simple if they connect a single input stream to a single output stream and either do not take a procedural parameter or define the procedural parameter directly as a body of the connector (i.e. they do not use a helper). In the other cases the connector will be stored as a node instead.

The VPE and the Runtime make use of various layers of abstractions to make it easier to add support for different databases (currently only the Neo4j graph database is supported, but support for XML files would be a valuable addition), robots (currently only Softbank NAO is supported) and even execution models (for example using either multithreading, multiprocessing or even grid computing) in the future.

Validation of the programming environment is done through various case studies and by performing a user study. In the case studies I show how the RRP Graphs can be used to construct various useful behaviours for a robot. In one such application the robot changes its tracking behavior of an object based on the distance to the object. I then show that this behavior can be ran on a robot [40]. Each case study is not only designed as RRP Graph but also implemented using the VPE. In the user study I show that end-users, people with little programming experience, can use the VPE to easily explain, debug and create behaviours. I also show that based on the time tasks took to complete in our VPE and a state of the art commercial platform (Choregraphe), our VPE is competitive with Choregraphe in the explanation and debugging task and exceeds the capabilities of Choregraphe in the creation task [44]. Based on self evaluation by the participants using NASA-TLX, our VPE and Choregraphe are competitive for the explanation and debugging tasks, while our VPE again exceeds the capabilities of Choregraphe in the creation task.

# Acknowledgements

Before I started my Ph.D. I was grateful to be an internship student in Professor Kenji Suzuki's Artificial Intelligence Laboratory. Joining this laboratory was a wonderful case of serendipity, as my previous research had mostly focused on Empirical Software Engineering and not Artificial Intelligence. I was however mesmerized by the technology students in this laboratory were developing, and decided to radically shift my research direction.

It was there that I found out about a new Ph.D. programme that had just started, called Empowerment Informatics, that focused not just on information systems and their development processes but also on the human factors of technology. On the day that I left Japan to return to my home country, less than ten hours before my flight, I took the entrance examination for Empowerment Informatics. After that I left Japan, not knowing whether I would return.

I was utmost delighted when my friends in Japan notified me about having seen that I was accepted into the Ph.D. programme. I was officially notified of my acceptance a few hours later by e-mail. In the Spring of 2015 I returned to the Artificial Intelligence Laboratory and started with the Ph.D.

First of all I would like to thank my main supervisor Professor Kenji Suzuki for twice allowing me to be a member of his laboratory. I would also like to thank my co-supervisors Professor Kazuhiko Kato and Professor Masakazu Hirokawa for their continued feedback and support throughout the Ph.D. Professor Fumiaki Mukakami and Professor Hiroaki Kawamoto also advised me for which I am grateful. In 2017 I was an internship student at AIST under the supervision of Saburo Takahashi and Dr. Noriaki Ando, I would like to thank them for all the things that I learned during my internship. During my Ph.D. I have continued the study of development and operations with my previous supervisors Dr. Maya Daneva and Dr. Chintan Amrit, I would like to thank them for collaborating with me even after I graduated for my masters. Without the support offered by the EMP Office my life would have been a lot more stressful, I am really glad for your support. I want to thank the current and former members of the Artificial Intelligence Laboratory, especially my current and former tutors and the 2017–2018 Ph.D. graduates, and my fellow students in the Empowerment Informatics Ph.D. programme, especially Maša Jazbec for our cooperation on the idMirror project. Finally I would like to express my gratitude to the Ph.D. Program in Empowerment Informatics and the Japanese government for the financial support that they have offered me throughout my stay in Japan.

# List of Publications

Content from the following publications is used in this thesis:

[1] Floris Erich, Chintan Amrit, and Maya Daneva. "A qualitative study of DevOps usage in practice". In: *Journal of Software: Evolution and Process* 29.6 (2017). DOI: 10.1002/smr.1885.

[2] Floris Erich, Masakazu Hirokawa, and Kenji Suzuki. "A Visual Environment for Reactive Robot Programming of Macro-level Behaviors". In: *Social Robotics*. 2017. DOI: 10.1007/978-3-319-70022-9_57.

[3] Floris Erich. "End-user Software Engineering of Cognitive Robot Applications Using Procedural Parameters and Complex Event Processing". In: *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. SPLASH Companion 2016. Amsterdam, Netherlands: ACM, 2016, pp. 47–48. ISBN: 978-1-4503-4437-1. DOI: 10.1145/2984043.2998538.

[4] Floris Erich and Kenji Suzuki. "Cognitive Robot Programming using Procedural Parameters and Complex Event Processing". In: *IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. 2016. DOI: 10.1109/SIMPAR.2016.7862376.

[5] Floris Erich, Chintan Amrit, and Maya Daneva. "A Mapping Study on Cooperation between Information System Development and Operations". In: *Product-Focused Software Process Improvement*. Ed. by Andreas Jedlitschka et al. Vol. 8892. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 277–280. ISBN: 978-3-319-13834-3. DOI: 10.1007/978-3-319-13835-0_21.

[6] Floris Erich, Chintan Amrit, and Maya Daneva. "Cooperation Between Information System Development and Operations: A Literature Review". In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '14. Torino, Italy: ACM, 2014. DOI: 10.1145/2652524.2652598.

The following additional co-authored publications are evidence of the author's interdisciplinary ability as expected as a contribution to developing the discipline of human informatics:

[1] Maša Jazbec, Floris Erich, and Hiroo Iwata. "A glance of cultural differences in the case of interactive device art installation idMirror". In: *AI & SOCIETY* (June 2017). ISSN: 1435-5655. DOI: 10.1007/s00146-017-0737-0.

[2] Maša Jazbec, Floris Erich, and Hiroo Iwata. “idMirror”. In: *Proceedings of the 23rd International Symposium on Electronic Art ISEA2017*. 2017, pp. 468–474.

[3] Maša Jazbec and Floris Erich. “Investigating Human Identity Using The idMirror Interactive Installation”. In: *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. CHI EA ’16. San Jose, California, USA: ACM, 2016, pp. 3851–3854. ISBN: 978-1-4503-4082-3. DOI: 10.1145/2851581.2891091.

[4] Gloria Ronchi et al. “Demo Hour”. In: *interactions* 23.5 (Aug. 2016), pp. 8–11. ISSN: 1072-5520. DOI: 10.1145/2973920.

# Contents

# Chapter 1

# Introduction

The idea that one day man made creatures would roam the world has existed for thousands of years. For example, in Greek mythology an automaton called Talos (or Talon) which was made of bronze was said to protect the island of Crete. In the 18th century automata such as the duck of Vaucanson entertained the people of Europe. Humankind has been able to see various advances in the field of robots, a term that was coined in a 1920 Czech science fiction play called "Rossum's Universal Robots". These days much of our quality of life is provided by goods produced by industrial robots. The effectiveness of robots would not have been achievable however without developments in another technological field, namely that of computers.

In the 19th century Babbage created early computers such as the difference engine, which was a large industrial device able to compute polynomial functions. In the 20th century Alonzo Church invented the lambda calculus for specifying computation. A few years later Church's student Alan Turing came up with a mathematical model of computation defining an abstract machine, referred to as a Turing machine. It was later proven that the lambda calculus of Church and Turing machines both had the same computional abilities, forming the Church-Turing thesis. Von Neumann was able to create a practical computer architecture based on the theoretical concepts of a Turing machine (which because of its inclusion of an infinite tape is not a machine that can practically be build). Today, the architecture of virtually every PC is based on the ideas of von Neumann. The ideas of Church were further developed by John McCarthy, who implemented a language based on the lambda calculus, called LISP (short for LISt Processing). McCarthy is also considered one of the grandfathers of the field of Artificial Intelligence, which seeks to embed intelligence into man-made artifacts.

Robots are commonly found in factories, however in our daily lifes we do not often encounter robots yet. Recently various affordable robot platforms have become available, such as the canine-shaped robot AIBO from Sony, humanoid shaped robots such as NAO and Pepper from Softbank and OP2 from Robotis. These robots are social robots, i.e. they are designed to interact with people. Social robots can be contrasted with industrial robots that typically operate in an environment away from people, to whom the robots could perform harm. Social robots are useful for supplementing the activities of professionals in various fields, such as customer service, therapy and elderly care. This thesis specifically focuses on applications within the field of Socially Assistive Robotics.

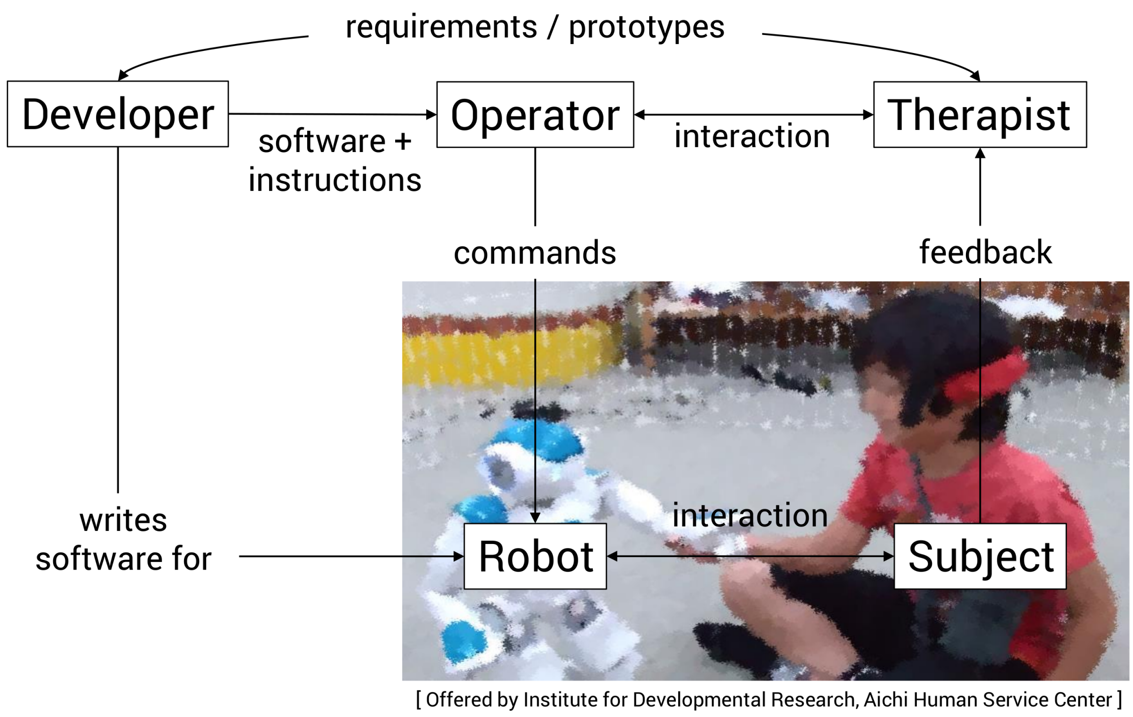


FIGURE 1.1: A schematic representation of the preparation and performance of an experiment in the field of Socially Assistive Robotics.

Even though social robots seem applicable for various fields, in practice they are only rarely encountered. One of the problems of social robotics is that software is necessary to make a robot do a useful task. Social robots however need to operate in complex environments that are both dynamic and inherintly concurrent. Dynamic refers to the environment constantly changing, e.g. objects can enter the environment freely, can mutate within the environment (e.g. changing pose) and can exit the environment at any time. Concurrent means that this dynamicity applies to multiple objects of different types. The software running on the robot needs to react intelligently to the stimuli being generated by the objects within the environment. This is challenging to realize for both novice and experienced programmers.

If social robots are to flourish, a simple approach has to be developed that hides the complexity posed by the environments in which users want to apply the robot.

This section introduces the reader to the application area of this research, the objectives of this research and the methodology for performing the research.

## 1.1 Socially Assistive Robotics

In recent years researchers have used robots with the purpose of socially assisting people. Such applications are reviewed in CHAPTER 2. One major contribution in this field was made by researchers of University of Hertfordshire in the United Kingdom. In a study they show that children are more likely to react favorably to a robotic actor than a regular person [87].

FIGURE 1.1 shows how an experiment in the field of Socially Assistive Robotics is typically performed. SAR Experiments consist of a preparation phase and an execution phase. There are five components which play a part in the experiment: The robot (which consists of both hardware and software), the developer who creates the robot, the operator who operates the robot during the experiment, the therapist who wants to use a robot within an experiment, and the subject that will physically interact with the robot.

During the preparation phase a therapist and a developer work together on specifying the requirements of the robot application. These requirements include aspects of both hardware (i.e. the robot) and software (that controls the robot). Typically an iterative process is used in which the developer elicitates requirements from the therapist and constructs prototypes that attempt to fulfil these requirements. This process can last from days to weeks to months, depending on the type of application necessary and the availability of existing hardware and software[1].

After a prototype has been developed that matches the requirements of the therapist, the hardware and software will be made available to the operator. An important part of the robot is a set of instructions on how to use it[2].

During the experiment the operator interacts with the therapist to determine which behaviors the robot should exhibit. Experiments in Socially Assistive Robotics use completely manual control, completely autonomous control or a mix of both[3]. In the case of completely manual control an operator teleoperates the robot. In the case of completely autonomous control the operator only needs to configure the robot to start the right control application. The robot itself interacts with the subject, which is observed by the therapist.

## 1.2 Research objectives

The main research objective is to create a *reactive robot programming* environment so that *end-users* can develop and operate applications for *Socially Assistive Robotics (SAR)* experiments with minimal support from a robot programming expert.

To be able to reach this objective, the following subobjectives were formulated:

1. To describe the current applications of Socially Assistive Robotics.
2. To describe the current approaches to develop and operate robot software.
3. To develop the reactive robot programing environment.
4. To test the effectiveness of the developed robot programming environment.
5. To describe the contexts in which the robot programming environment can effectively be applied.

[1]The field of Software Product Line Engineering studies how hardware and software can be most efficiently reused. CHAPTER 2 will look at this in more depth.

[2]The amount of documentation needed is a controversial topic in the field of Software Engineering. Traditional waterfall-like models of software development have often put too much emphasis on the production of documentation. Agile software development methods emphasize working software over documentation. Discovering how much documentation is necessary is a complicated pursuit and is very dependent on the context in which the software will be used.

[3]More on this in CHAPTER 2.

## 1.3 Methodology

This research was performed using the Design Science Methodology as described by Wieringa [98]. Two specific techniques which were applied were Systematic Literature Review and User Studies.

**Systematic Literature Review** When starting a research project on a topic in which the researcher has little or no experience, it is wise to start with a Systematic Literature Review. By performing a search on an academic literature database using well defined search terms and well thought out inclusion and exclusion criteria, a researcher can get an accurate overview of the state of research in that field. In specific fields such as medicine it is very common for researchers to perform SLR's periodically and publish these in order to make it easy for other researchers to grasp the research field. We based our approach to performing SLRs on the explorations by Kitchenham on performing SLRs in the field of Empirical Software Engineering [62].

**User Studies** When working on artifacts that primarily target a user and a measurement of usability of the artifact is required, a user study can be performed. Typically the user will be asked to perform a set of tasks and the performance of the user is measured both quantitatively (by for example recording the time taken to perform the task) and qualitatively (by for example interviewing the user about his experience using the artifact). We based our approach on performing user studies on the guidelines of performing studies in Human-Computer Interaction as written by MacKenzie [66].

# Chapter 2

# Literature Review

This chapter reviews the literature of Cognitive Robotics Architectures, Software Engineering for Robotics, Visual Programming for Robots, Programming by Domain Experts/End-Users and Applications of Socially Assistive Robotics. Each subject will be discussed in its respective section.

## 2.1 Cognitive Robotics Architectures

One of the goals of roboticians is to build robots that look like and behave like humans. Researchers such as Hiroshi Ishiguro have been able to create very humanlike robots. This thesis however does not specifically focus on humanlike robots. That said, robots are required to exhibit some degree of intelligence to get accepted as a companion by humans. It is hence interesting to look at solutions to bestow even as little as the illusion of intelligence into a robot.

Self regulating systems are one form of intelligent systems. Cybernetics studies the communication and control of animals and machines [96]. Using simple connections between sensors and actuators machines can be build that exhibit basic intelligence, as demonstrated by Braitenberg's vehicles [18]. According to Minsky's Society of Mind theory, a complex mind is constructed from many simple individual parts [72]. In the subsumption architecture a robot has multiple control layers that can inhibit each other [21]. Researchers as MIT have built various robots using the subsumption architecture [20], such as the sonar navigation robot Allen, six-legged hexapod Genghis, the robotic tour guide Polly and the humanoid robot Cog. Arkin coined the term Behaviour-Based Robotics and wrote a comprehensive review of the field [7].

Motor schemas is another architectural style for constructing behaviour-based robots [6]. It is typically used for reactive navigation in 2D and 3D spaces. Motor schemas can describe basic behaviours and can be combined to create more complex behaviours. For example, by combining the *avoid-static-obstacle* and the *move-to-goal* schemas by summing their vector fields a new schema *move-to-goal-while-avoiding-static-obstacle* can be created [7].

Many producers of robots develop their own architecture for programming and controlling the robot, leading to the proposal of an architectural description language to homogenize the process of developing frameworks in a formal way [82]. Many other projects try to offer middleware solutions for connecting software components, such as the Robot Operating System [81], YARP [45] and

OROCOS [25].

## 2.2 Software Engineering for Robotics

This section will look at techniques for constructing software which can aid in the Robotics Software Engineering process.

Research has focused on introducing Component-Based Software Engineering practices in robotics [23, 24]. A shift from code-driven to model-driven designs was proposed in order to manage the complexity of robot software [88]. Model-Driven Software Development (MDSD) for Robotics attempts to construct robot software using models [22]. MDSD for Robotics is seen as a direction to move to a higher level of abstraction [89]. Some of the first applications of MDSD in the field of robotics were an application on the Sony Aibo robot [17] and an application using Lego Mindstorms [60]. Some of the projects that explore MDSD for Robotics are (according to a review article [83]) the European Union funded BRICS project [26], RobotML (which is part of the French research project PROTEUS) [36], SmartSoft [88], and $V^3CMM$ [4].

While MDSE might focus on a single product with many variable components, Software Product Line Engineering (SPLE) is increasingly being applied in industry as an approach to construct families of software [29]. The approach for enabling the combination these building blocks together is based on Software Product Line Engineering (SPLE) for Robotics [50]. SPLE divides the process of developing software into two phases: Domain Engineering and Application Engineering [79]. The goal of Domain Engineering is to create models of a domain in which it is beneficial to reuse a significant portion of behaviour. The goal of Application Engineering is to use the models created during the Domain Engineering phase to develop working applications. The HyperFlex toolchain attempts to offer this kind of capabilities [51]. Robots that operate in a complex environment might benefit from software that can adapt itself at runtime [52]. SmartTCL is an architecture that supports managing run-time variability [65].

Various other European Union projects that explore software engineering issues in the field of robotics are underway. The VERSATILE project studies how robots can be applied in highly reconfigurable production lines. The ROS Industrial project studies how the Robot Operating System can best be used for industrial robots. The RobMoSys project studies further how models can be used to support robotic systems.

## 2.3 Visual Programming for Robots

We already mentioned Lego Mindstorms as a platform for programming robot applications. While Lego Mindstorms is a great project that allows children to learn how to construct and program robots, the programming tools do not scale to larger applications due to the following limitations (adopted from [74, chapter 3]):

- Predetermined values for block parameters such as motor speeds.
- Only a single global control loop.

- No branching structures such as if and switch.
- Only a single speed setting for the differential drive.

RoboFlow is a visual programming language that allows end-users to program a robot to perform mobile manipulation tasks [3]. It uses the flowchart-based programming paradigm, in which a program consists of decision and action nodes. Action nodes make the robot perform some behaviour while decision nodes can control the flow of a program based on environmental sensing.

Choregraphe is a graphical tool for programming humanoid robots such as Pepper and NAO [80]. For those robots it is the de facto standard visual programming tool. The programming paradigm of Choregraphe resembles flowcharts. Boxes have input and output ports, on which they can receive data (input) or transmit data from (output). The boxes themselves are provided as box libraries.

The RT System Editor of OpenRTM-aist is a tool for visually configuring Robot Technology Components (standardized by the Object Management Group) [5]. Once expert programmers have developed the components needed to achieve a certain kind of task, a domain user can combine the components using a drag and drop interface and arrow connectors.

Targets-Drives-Means (TDM) is a runtime environment and proposed visual programming environment for robots [13–15]. The basic building blocks of programs in TDM are behaviours. Behaviours are sorted using score calculators. At any time, the behaviour with the lowest calculated score will be activated. Behaviours contain a list of actions that will get activated in parallel if a per-action-assigned condition is valid. The action and conditions themselves are written using the Python programming language. Additionaly the behaviours can act as states in a state machine in which the conditions can act as transitions.

## 2.4 Programming by Domain Experts and End-Users

Professional Software Engineers typically spend a lot of time to become experts in the domain that they develop software for. Instead of developing tailor made software for a specific user, computer scientist try to move the software engineering process to a higher level of abstraction. Abstractions can be collected in a platform aimed towards Domain Experts or End-Users, who can use the abstractions to solve the problems they face. The discipline of Software Product Line Engineering (which we shortly discussed in the section on Software Engineering for Robotics) tries to offer this abstraction to Domain Experts and the discipline of End-User Programming tries to offer this abstraction to regular end-users.

Many systems have been developed for programming using the visual capabilities of people. This section reviews the ones that most contributed to this research.

### 2.4.1 End-User Programming

Logo is one of the first educational programming languages developed, originally being released in 1967 [1]. Its developers were Wally Feurzeig, Seymour Papert and Cynthia Solomon. Logo was originally a text based programming language, however various graphical versions of Logo have been implemented. Logo used a turtle figure which acted as an onscreen cursor, allowing the user to easily create graphics. It has also been implemented in dialects such as NetLogo and StarLogo, which are today still used for visualizing mathematical models and artificial intelligence algorithms. The Logo programming language is based on the Lisp programming language which was invented by John McCarthy [68].

HyperCard was an end-user programming tool for the Apple Macintosh [64]. It allowed users to create a graphical interface and an underlying database. Additionally users could write scripts using a simple scripting language called HyperTalk. More recently Mac computers have featured a tool called Automator which allows users to create scripts for performing automated tasks.

AgentSheets [84] is a VPL aimed towards educating computer science in public schools. It allows the definition of a program using "if A then B" statements, where A is a condition and B is one or more actions to be performed in sequence.

Etoys [61] is a project, which was directed by Alan Kay while working at Disney, aimed to support constructionist learning. It allowed the user to easily modify the Morphic graphical interface using a block based programming language.

Spreadsheets are also a tool used by end-users for creating programs. By allowing the user to create formulas in which functions and mathematical operators can be combined with cells, numeric processing applications can be written. Spreadsheet software such as Microsoft Excel also allows for the writing of scripts to further customize the spreadsheets using Visual Basic. Similarly, a product called Microsoft Access allows the creation of database applications by using a drag-and-drop user interface creator combined with the SQL query language and Visual Basic. These more advanced features of Excel and Access are however oriented towards people with some programming education.

Scratch is a modern Visual Programming Language [85]. It allows a programmer to sequentially connect blocks. Blocks start by reacting to an event, after which a sequence to steps is executed. A web-based block-based programming language inspired by Scratch called Blockly was developed by Google [46]. Google then used Blockly in their platform for end-users to create applications for Android, called App Inventor [99]. This project is now being maintained by MIT. A Scratch based programming environment that incorporates features from functional programming called Snap was developed by Brian Harvey and Jens Mönig [57].

Scratch and Snap are mostly used for creating 2D animations. Alice, a project formerly led by Randy Pausch, is a block based visual programming language for creating 3D animations [34].

### 2.4.2 Automata and Flow-Based Programming

Various system notations and programming environments exist for modeling a program as rectangles or circles that are connected using arrows. These models can be either descriptive, i.e. describing how an existing system functions or validating an existing system, or prescriptive, i.e. describing how a system to be build should function. Prescriptive models can be used for communication between people and in some cases can be used to generate new artifacts such as source code.

State diagrams or state tables can be used to model automata, and both notations have equal modeling capabilities. Modern computers are based on automata theory, using a physical architecture invented by von Neumann and a computational architecture based on Turing Machines [93]. It seems logical that the underlying model of computation should be used to model computation using software. The model of a Turing Machine is however quite verbose and hard to use for modeling high level applications that are used today.

Simple machines such as a stopwatch or a turnstile can be modeled using a simple type of automaton called a finite-state machine (FSM). Such a machine consists of a set of states and transition functions between the states. Finite-state machines can be found in many embedded computing applications because of their simplicity and flexibility to model simple machines. Two traditional formalisms for finite-state machines are Mealy and Moore machines. Mealy machines [69] define the actions performed in a state within the state itself, hence it does not matter how the state is entered, the resulting actions will be the same. Moore machines [73] define the actions performed in a state on the transitions to the state, in which case it does matter how the state is entered. Practical robotics programming environments such as TDM and ROS (inside a package called smach) include an FSM. Harel statecharts are another FSM formalism [55], extending state diagrams with hierchical modeling, concurrency and communication (broadcasting) formalisms. While FSMs are useful for modeling simple machines, they do not have the full computational capabilities of a Turing machine, and are not typically used in highly interactive applications.

Automata are aimed towards system engineers, however there are alternative notations such as flow charts that are more commonly used by software engineers or even by end-users. Flow charts model the sequence of actions performed by a system, which differentiates them from state diagrams, that instead model the states a system can be in. The arrows in flow charts show the direction of interaction after a task has been completed. A particularly successful implementation of flow charts for domain experts is Business Process Modeling Notation, which is standardized by the Object Management Group [27]. Research by Green and Petre shows that flow charts are a user friendly technique for modeling sequential behaviour [54].

While most of the beforementioned diagramming notations are used as means of communication between people, a technique called Dataflow Programming tries to extent flowcharts to be a useful technique for programming new systems. There are many examples of systems adopting the dataflow programming paradigm, such as LabVIEW, Max/MSP, Pure Data and Simulink. Typically these systems are used by domain experts who have technical knowledge but might have little programming experience.

## 2.5 Applications of Socially Assistive Robotics

We performed a Systematic Literature Review (SLR) in order to accomplish the first research objective, to describe the current applications of Socially Assistive Robotics. Because the literature review regarding this subject was performed in order to answer one of the main research questions of this thesis, a Systematic Literature Review (SLR) was performed [62]. An SLR allows for an exhaustive synthesis of the literature regarding a certain topic to be performed.

The main question is how Humanoid Robots are being applied in Socially Assistive Robotics experiments. This question is split up into the following subquestions:

1. What robot was developed or used?
2. What are the goals of the robot application?
3. How was the robot controlled?
4. What kind of behaviours did the robot exhibit?
5. Which type of sensors were used?
6. Which type of actuators were used?

The third subquestion, how the robot was controlled, is the hardest to answer. The autonomy of the human subjects and of the robot have to be considered. Experiments which have real human subjects and a robot which is controlled manually, but covertly, are called Wizard of Oz experiments. Various other combinations of "Wizard" and "Oz" are suggested to reflect on various levels of autonomy of the human subjects and of the robot [90]. Experiments which are performed with real human subjects and a completely autonomous robot can be considered as "Wizard *and* Oz". As this research is about *applications* of robotics, studies which do not have real human subjects have been excluded. Autonomy of a robot is a spectrum ranging from completely autonomous to completely manual/teleoperated. Studies were classified based on three levels of this spectrum: Autonomous, mixed and manual.

### 2.5.1 Methodology

To select studies for inclusion in this review the search term *(social or socially) and assistive and (robots or robotics) and humanoid* was applied to the *Web of Science* database. The composite search string allows to find a compact yet complete list of papers indexed by the database searched. This search was last performed on the 1st of February 2016 and returned 24 papers. One inclusion criterium was used, that to be included, a paper should discuss a primary study of the implementation of a robotic application and should hence not be limited to a theoretical treatise of robotic application construction or be a review article. These exclusion criteria were used: (1) Paper is written in a language other than English and (2) paper is inaccessible through the academic literature databases of the University of Tsukuba.

To decide whether the paper met the inclusion criteria and did not meet any of the exclusion criteria at least the title and abstract for every paper returned by the search were studied. All the papers included in the study were completely read, and data was extracted from them using the conceptual framework.

### 2.5.2 Conceptual Framework

While doing the research a conceptual framework was iteratively constructed to store the knowledge gained during the study. If any concept would get added

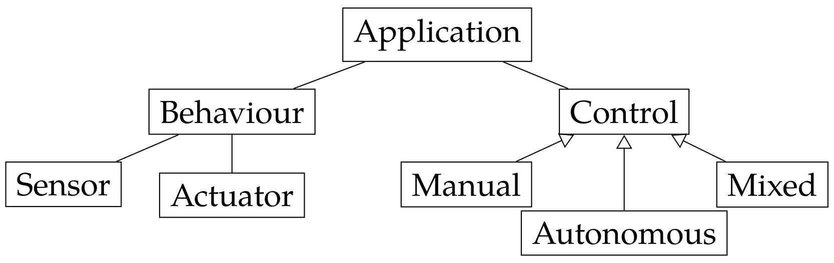
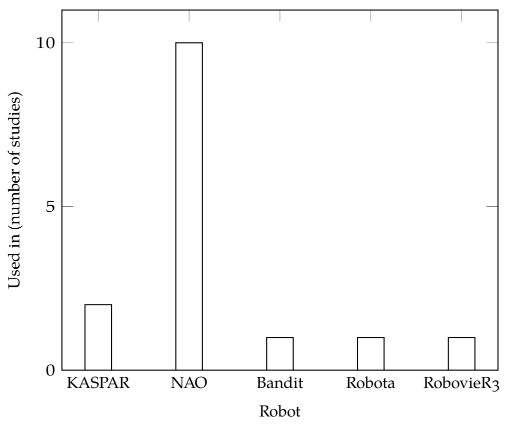


FIGURE 2.1: Behaviour-based Conceptual Framework of SAR applications. Regular lines represent association. The open triangle arrow represents an is-a relationship.

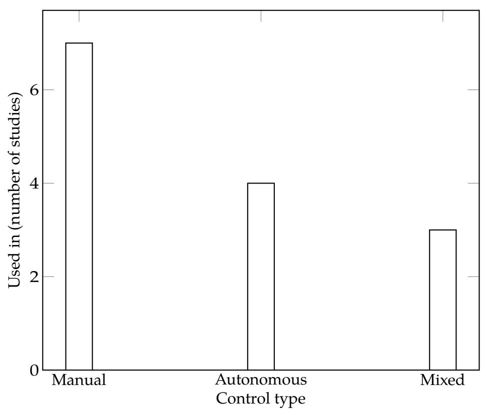
or removed to the framework a verification would be performed to determine whether this influenced the data extraction from any of the previous papers studied. This approach is based on the Grounded Theory methodology of performing qualitative research [53]. If this research would be repeated by a different researcher (while using the same methodology), the conceptual framework constructed should be similar.

FIGURE 2.1 shows the conceptual framework that was used to evaluate the studies. The conceptual framework defines the following concepts:

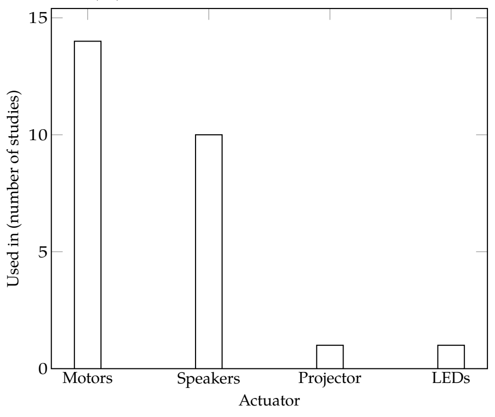
- Robot: Each study used one or more robots for which the an application was developed.
- Goal: Each robotic application tries to accomplish one or more goals.
- Behaviour: To achieve the goals the robot has to exhibit one or more behaviours.
- Actuator: To exhibit a behaviour a robot has to use one or more actuators, which are hardware components which can make changes to the world. In this research we are interested in the type of actuator.
- Sensor: Sensors are hardware components which retrieve information from the environment in which the robot is deployed. We only concern ourselves with sensors providing exteroception, i.e. perception of things happening outside of the robot embodiment [7].
- Control: Robotic applications define behaviours in terms of combinations of sensors and actuators. Control concerns how components of these two types are connected. We define three types of control: Autonomous, mixed or manual (Wizard of Oz).
- Autonomous Control: Control exercised by some automated system, such as algorithms running on the robot. Any control action taken by the robot in response to an action performed by a research subject is also considered autonomous.
- Manual Control: Control by a human operator through a programming interface of the robot.



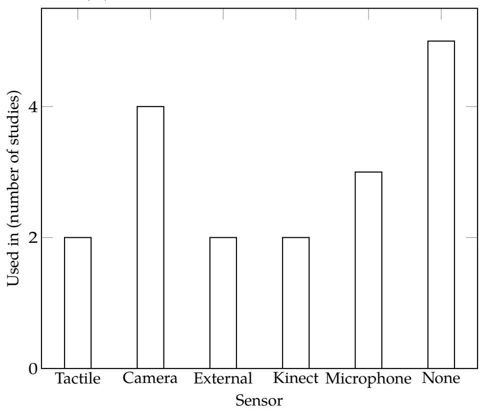
(A) The robots that were used.



(B) How the robot was controlled.



(C) Actuators used.



(D) Sensors used.

FIGURE 2.2: Quantitative evaluation of the studies.

- Mixed Control: Mixed control by both some automated system and a human operator.

Metadata for the studies also includes the title, authors, year of publication and research goals.

### 2.5.3 Results

Quantitative and qualitative data was collected from the studies. FIGURE 2.2 shows the quantitative results. TABLE 2.1 shows the qualitative results. These results are used to determine the requirements of the robot programming environment.

TABLE 2.1: Information gathered from the studies.

| Ref. | Robot | Application Goal | Control | Behaviours | Actuators | Sensors |
|---|---|---|---|---|---|---|
| [31] | KASPAR | Teach children with ASD to identify their body parts and increase their body awareness | Mixed | Reacting to touch; identifying body part and asking participant to match; identifying sequence of body parts and asking the participant to match; and singing a song while dancing and encouraging the participant to join | Motors, speakers | Tactile |
| [86] | KASPAR | Teach children with ASD about various topics related to their self, their body and social interaction | Autonomous | Respond to touch in different areas by moving the body of the robot | Motors | Tactile |
| [47] | NAO | Assist staff in kindergartens | Manual | Singing a song while dancing, initiating personal contact, playing a game, falling/getting up, giving explanations | Motors, speakers | Microphone |
| [92] | NAO | Assisting in elderly care | Autonomous | Providing environmental information; playing music; managing phone calls; monitoring self treatment; monitoring the environment; providing video calls | Motors, speakers, projector | Microphone, camera, external sensor network |
| [2] | NAO | Play a role-taking game with a patient | Manual | Pretend play using talking, gesturing and playing music | Motors, speakers | None used |
| [48] | NAO | Interact with teachers | Mixed | Detecting nearby people, talking to people, grasping | Motors, speakers | Camera |
| [8, 9] | NAO | Deliver a letter | Manual | Walking, bowing, handing over a letter, waving | Motors | None used |
| [12] | NAO | Test and train children with ASD about attention skills | Mixed | Asking questions while moving naturally | Motors, speakers | Camera network |
| [100] | NAO | Interacting with an ASD child | Manual | Sitting, moving / dancing, speaking | Motors, speakers | None used |
| [75] | NAO | Engage with and instruct hotel guests | Autonomous | Looking at hotel guests, reading from a script | Motors, speakers (Text-To-Speech) | Kinect |
| [95] | Bandit | Assist individuals post-stroke | Autonomous | Giving instructions / feedback / motivating, pointing, nodding | Motors, speakers | Wire puzzle |
| [87] | Robota | Interact with an autistic child | Manual | Move according to operator's instructions | Motors | None used |
| [63, 94] | Robovie R3, NAO | Tutor sign language to a child | Mixed | Indicating signs from Turkish Sign Language | Motors, LED's, speakers | Kinect, camera, microphone |
| [67] | NAO | Teach exercises to prevent back pain | Manual | Demonstrate exercises to subjects | Motors | None used |

# Chapter 3

# Development and Operations

This chapter gives a short summary of DevOps, which refers to interaction between development and operations. It is largely based on a Systematic Literature Review performed in 2014 [41, 42] and a qualitative interview based study performed between 2014 to 2016 [43].

For the purpose of this research we define DevOps as interaction between development and operations. DevOps is necessary because software systems have become more complex, whereas in the past one could install a software package on their PC and use it standalone, these days software is often either completely hosted remotely (e.g. Software-as-a-Service) or has remote components. We say that his type of software has a significant operational component. The software needs to be both developed *and* operated to deliver value.

Even though agile software development methodologies such as Scrum and Extreme Programming are considered as modern, they do not typically concern how the developed software has to be operated. DevOps is not a new methodology but rather an extension to existing methodologies. Some organizations and researchers have tried to create methodologies for combining agile software development with DevOps, examples of this being the Scaled Agile Framework (SAFe) and Disciplined Agile Delivery (DAD) While these newly developed methodologies are interesting artifacts of study, in this thesis we are not concerned with the specific implementations of DevOps, but rather the high level philosophy behind DevOps.

We explore the interaction between development and operations at multiple levels, ranging from the individual to the whole organization. The interaction is supported through a set of principles and practices, which we will also shortly review. The chapter will also discuss how DevOps could be applied in the field of robotics. The same drivers of DevOps adoption in the field of Information Systems also exist in the field of robotics: Robotic technology no longer runs completely standalone but has some remote components (e.g. for monitoring the functioning of the system, remote control, installing/updating software components, etcetera).

## 3.1 Levels of DevOps

DevOps originated from the fields of Enterprise and Information Systems. It can be applied on multiple levels and mostly impacts the individuals, teams and departments within an organization.

**Individual** DevOps at the level of the the individual refers to what skills an individual person should have to be able to function in a software project with a significant operational component. Typically organizations hire separate personnel for the tasks of developing and operating software. The discipline of development is typically practiced by software engineers or software developers. The discipline of operations is typically practiced by system operators or system administrators. DevOps applied to the individual level means that a developer should get enough skills to be able to support an operator in their daily work, and vice versa, operators should get enough skills to be able to support developers in their daily work.

**Team** The team level of DevOps refers to what roles a team working on software should include. Traditionally organizations had seperate teams focused on developing and operating software. DevOps applied to the team level means that a team should be responsible for both developing and operating software. Practically this means that a team needs members that have experience in both development and operations.

**Department** The department level of DevOps refers to how organizations should be structured. Tradionally organizations have a department that focuses on developing software and a department that focuses on operating software. DevOps applied to the organization level means that organizations have departments that are both developing and operating software. Practically, departments should focus on both developing and operating software. This can be achieved by organizing departments by services provided instead of disciplines practiced.

## 3.2 Principles and practices of DevOps

Various practices are adopted which can either been seen as being correlated with organizations that adopt DevOps, or are thought to lead to better interaction between development and operations. This section will discuss some major practices of both categories.

**Culture** Culture refers to the culture within a team and organization. Organizations seek to create a culture in which the disciplines of both development and operations are given enough consideration. Development personnel should realize that the value of a system is only produced when a system is being succesfully operated. To support this, developers need to consider operations when formulating the requirements of a system. Systems should also be documented in a way that operators can easily make the system operational, independent of the help of the original developers.

**Automation** Automation refers to automation of the software process itself. It is logical that a field focuses on automating work turns its attention inwards to optimize its own performance. Automation in DevOps takes a holistic perspective, considering automation of the software process from development to operations, instead of considering automation within each discipline seperately. For example, when developers check software into version control, a watch process can trigger a test suite, and if that test suite reports no errors, another watch process can automatically deploy the software into a beta testing environment.

**Measurement** Measurement refers to measurement of the efficiency and effectiveness of the software process. Again, DevOps takes a holistic perspective, measuring various aspects of the software process from development to operations, instead of only performing measurements within each discipline seperately. For example, many organizations measure the time between elicitation of a feature, implemention of the feature and availability of the feature to the users of a system.

**Sharing** Sharing refers to how development and operations personnel share space, responsibilities, resources, knowledge and information. Sharing is required for development and operations personnel to effectively work together.

**Monitoring** Monitoring refers to keeping track of the efficiency and effectiveness of a running system. Some aspects of a system can be monitored without any modifications to the software itself, e.g. CPU and memory usage. Other aspects require the software to be modified, e.g. to log the actions performed by users. In either case having knowledge of the software process can improve the analysis capabilities of development and operations personnel. For example, an inefficient implementation of some algorithm can cause a higher CPU and memory usage. Development and operations personnel can also work together to visualize monitoring. Many organizations have for example implemented dashboards that report on the quality of service of systems, which are easily viewable within the offices of personnel.

**Lean** Lean refers to a method for finding and eliminating waste within a process. Lean in DevOps focuses on finding and eliminating waste created by a lack of interaction between development and operations. For example, when development personnel release software without proper documentation, operations personnel might have problems with making the software operational.

## 3.3 Usage in organizations

We studied how six organizations apply DevOps [43]. Four organizations were based in The Netherlands, one organization was based in the United Kingdom and one organization was based in the United States. We labeled the organizations FinCom1, FinCom2, SupportCom, PortalCom, UtilCom and CommunitySoft. The organizations were recruited at an industry conference on Agile Software Development and through personal connections. We recruited senior level employees at each organization to participate. We interviewed the participants, asking them questions about the implementation of DevOps within their organization.

The following list summarizes our findings for each organization:

**FinCom1** Motivation: Reduce lead time for new projects; improve problem solving; increase feedback.

Implementation: Introduced DevOps teams; Adopted DevOps Engineer Job Title; Put development and operations personnel under same management.

Problems: Employee discomfort with openness; Resistance against increased reponsibilities of development personnel; Development personnel considering Ops work as chaotic.

Results: Improved lead time.

**FinCom2** Motivation: Reduce system downtime; Reduce workforce size; Support automation.

Implementation: Using Xebia and UrbanCode frameworks for measuring progress; Introduced policy of making software available to actual users each iteraction; Introduced time and location overlap between development and operations personnel; Trained employees in System Thinking.

Problems: Employees focusing on production rather than improving production capacity; Management skepticism due to lack of evidence of effectiveness.

Results: Improved testing for one team.

**SupportCom** Motivation: Reduce miscommunication between development and operations personnel; Reduce release time of SaaS product.

Implementation: Introduced a specialized DevOps team.

Problems: Development personnel considering Ops work as ad hoc and chaotic; Reduced capabilities for management oversight.

Results: Increased problem solving capabilities; Improved communication between development and operations personnel; Fewer escalations of issues.

**PortalCom** Motivation: Release software more often; Free up resources to work on new features instead of problem solving; Increase product quality; Increase process velocity.

Implementation: Experiment with one team as DevOps team; Automation (version control and environment provisioning tooling).

Problems: Restructuring needed; Team members needed to have a wide skillset.

Results: Increased process velocity; Less time spend on setting up environments.

**UtilCom** Motivation: Deal with challenges in developing and operating complex scalable service architecture.

Implementation: Operations personnel trained in software development techniques; More respect given to people in a DevOps role than before; Adopted systematic approach to operations; DevOps is considered as a role held by select people; Single team explicitly called DevOps team; Implicit focus on DevOps by all dev and ops.

Problems: No terminology for evaluating progress; New metrics were needed for evaluating progress.

Results: Reduced escalations, false alarms and duplicate alarms.

**CommunitySoft** Motivation: Get more direct feedback from stakeholders.

Implementation: Introduced Continuous Integration as coordination tool; No separation between development and operations roles; Constant cautious experimentation.

Problems: Hard to find balance between producing and improving production capability; Not every member has sufficient skills for DevOps approach; Difficulty automating everything.

Results: Working software is regularly being delivered to stakeholders; DevOps as supporting instrument of adopting technical practices such as Continous Integration and Continuous Delivery.

## 3.4 Relevance to Robotics

DevOps originated from the fields of Enterprise and Information Systems and its practices can hence directly be applied to Cloud Robotics. On a philosophical level however DevOps touches upon a major problem in robotics: Robots are not just hard to develop but are also hard to operate. We hence formulate two perspectives of DevOps, that of DevOps by End-Users and that of DevOps for Cloud Robotics.

### 3.4.1 DevOps by End-Users

For a robot to be useful it needs to be developed with the operational goals of the end-user in mind. Robots have been highly successful in industrial applications for which it is easy to discover the operational characteristics of the robot, such as its size, functions, power usage, etcetera. In the past few years more general purpose industrial robots such as Baxter from Rethink Robotics have become available.

The general purpose robots that this thesis is concerned with, such as the Humanoid robot NAO, have had mixed success in fulfilling the goals of end-users. NAO is suitable to perform tasks such as dancing and tasks that are mostly composed of prerecorded tasks such as scripted motion and scripted speech. The robot has been less successful in more dynamic tasks, although the usage of NAO in robot soccer shows that it is quite a versatile robot. The development of NAO for robot soccer has however involved university students in the fields of Computer Science and Mechanical Engineering, who are not the end-users which the programming environment presented in this thesis is targeted towards. One of the ways in which the end-user is aided in developing and operating software for the NAO robot is through Choregraphe, a visual robot programming environment.

Features such as being able to easily deploy software to a robot can greatly aid in the development and operations of robot software. However the most benefit of a DevOps approach is gained when considering robots that are operating in real world environments that use external services to supplement local capabilities, i.e. Cloud Robotics. In the following subsection we will discuss DevOps applied to Cloud Robotics.

### 3.4.2 DevOps for Cloud Robotics

Robots can both use a cloud environment for supporting their behavior, or can be controlled from the cloud environment. Examples of the robot supporting their behavior through a cloud is the usage of cloud services for performing object and speech recognition. While these tasks can be performed on the robot itself, cloud services can offer more accurate results due to their increased computational and storage capabilities. Robots can also be controlled from the cloud environment, either through teleoperation, by pushing applications to the robot or by pushing updates to the software running on the robot.

Robots might use a network for communication between components embedded within the embodiment of the robot. Additionally a group of robots might communicate, potentially forming a swarm of robots. Platforms for Cloud Robotics, such as the European Union supported Rapyuta project, typically offer a computational model in which functionality can easily be transferred from running on a robot to running in an external cloud environment.

The value of a robot in this case depends not just on the performance of the hardware and software running on the robot itself, but also on the availability of the cloud services. Cloud services have to be designed so that they are resilient to failure, elastic to varying loads and responsive to the requests of the robot consuming the service.

# Chapter 4

# Reactive Robot Programming

A major limitation of current visual programming tools for robotics is that they rely on expert programmers to deliver a library of functionality that an end-user can then use. It is currently impossible to completely remove the need for expert programmers to write certain parts of a robotic system. Some tasks require a significant level of expertise, such as controlling low level hardware, such as sensors and actuators, and interacting with the operating system controlling the robot. Given a robot and a library that controls the low level hardware and interacts with the operating system, we believe it is possible to make available for the end users tools that allow him or her to perform more of the programming tasks typically performed by expert programmers. To this effect we propose a paradigm called Reactive Robot Programming (RRP), which allows end-users to create behaviours for robots. These behaviours can then be used in higher level tools such as Choregraphe, OpenRTM-aist or TDM.

RRP is a programming paradigm for developing and operating reactive software for robots, with a focus on End-User Programming. Robots are a type of Reactive System, which are systems that react to stimuli from an environment and can respond to these stimuli, potentially attemping to inhibit them [97]. Reactive systems have become the main type of system that computer users interact with, however an entirely different class of systems exist, that of transformational systems, which purpose is to transform some input to an output while receiving minimal intermediary instructions. Examples of transformational systems are compilers and translation tools. Examples of reactive systems commonly used these days are applications for smartphones and graphical applications on computers. RRP is an application of Reactive Programming in the field of robotics.

Reactive Programming is a paradigm for developing reactive systems. It has gained a lot of attention after Microsoft released the Reactive Extensions programming API. This was first made available for the Microsoft .NET framework and later was ported to Java (RxJava), Python (RxPy), JavaScript (RxJS) and various other languages. The name Reactive Extensions refers to the Reactive Programming Paradigm implemented using Extension Methods. Extension Methods allow programmers to extend an existing class with new methods, without modifying the existing class. From the perspective of this research, Extension Methods are an implementation detail of ReactiveX, Reactive Robot Programming does not require a language to support Extension Methods.

Reactive Extensions are motivated by the Observer/Observable design pat-

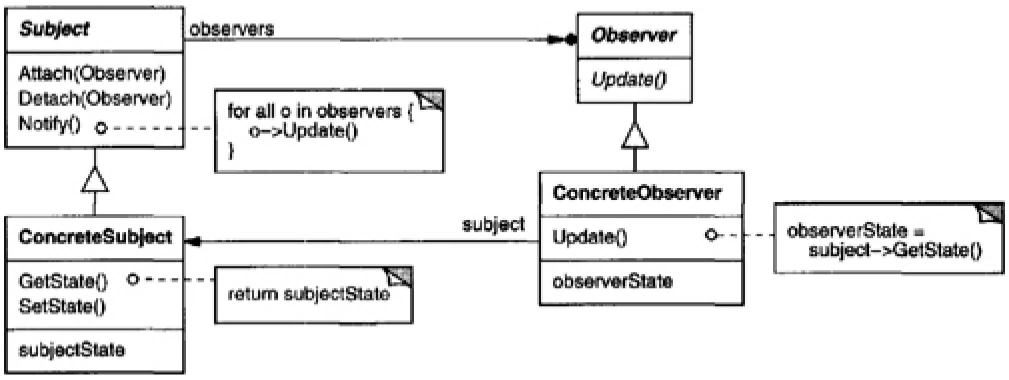


FIGURE 4.1: Structure of the Observer pattern (using the Object Modelling Technique, a predecessor of the Unified Modeling Language). [From: 49, Chapter 5]

tern [49], which FIGURE 4.1 shows. In this design pattern there is one observable subject that can notify a group of observer objects of events. Depending on the implementation, the observer objects either get passed a payload describing the event, or they have to query the subject.

Reactive Programming can be seen an extension to Event-Driven Programming. In Event-Driven Programming, code is triggered by the occurance of a certain event. Visual programming languages such as Scratch are Event-Driven, i.e. a set of steps is sequentially executed when a specified event occurs. The programmer can select the type of event that should trigger the execution.

Reactive Programming focuses on modelling the logic of how events are related. In RRP for example, events detected by sensors can be combined using various connectors (such as combine and merge). The programmer in this case specifies how to process the events leading to a certain actuator output.

In simple domains, such as 2D/3D animations and form-based graphical user interfaces, event-driven programming is often adequate as there is a one-on-one mapping between events and actions. In more complex domains, such as video games and robotics, there is a more complex relationship between events and actions, and we would argue that in this case reactive programming is more beneficial.

A useful particularity of the Observer pattern is that it was shown by Meijer to be the dual of the Enumerator (iterator) design pattern [70], a duality not noticed in the traditional design pattern literature. This implies that whereas the iterator pattern allows for *interactive* behavior in which the enumerator pulls data from the enumerable, the observer pattern allows for *reactive* behavior in which the subject pushes data to the observer.

In the late 20th century researchers were researching a variant of Reactive Programming using Functional Programming Languages such as Haskell. Early applications of FRP included Virtual Reality [38], animation [39], user interfaces [32], visual tracking [78] and robotics [59, 76, 77]. In the 21st century (Functional) Reactive Programming became a mainstream technique with the release of APIs and languages such as Reactive Extensions, the Java 9 Stream API, SodiumFRP [16] and Elm [33].

In Reactive Programming a programmer uses streams and connectors. Some of these streams are sources of data, some are final destinations (sinks) and some are temporary streams created to make a graph more manageable. Stream con-

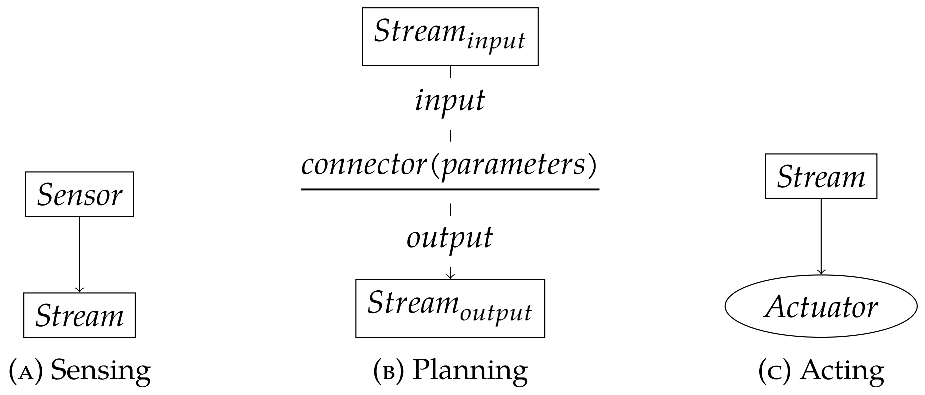


FIGURE 4.2: Main elements of the RRP Graph.

nectors are a typical part of a software architecture, in which various dimensions for stream connectors exist [91]:

- Delivery: Best effort, exactly once, at most once or at least once.
- Bounds: Bounded or unbounded.
- Buffering: Buffered or unbuffered.
- Throughput: Atomic units or higher order units.
- State: Stateless or stateful.
- Identity: Named or unnamed.
- Synchronicity: Synchronous, asynchronous or time out synchronous.
- Format: Raw or structured.
- Cardinality: Binary or N-ary (multi-sender, multi-receiver or multi-sender/-receiver).

The following sections will introduce the graph representation constructed for specifying behaviors in RRP, the connectors that are currently implemented and various sample applications.

## 4.1 Graph Representation

For teaching and describing Reactive Programming techniques there are four notations frequently used:

**Marble diagrams** These diagrams give an example of input given to a connector and the output produced by the connector. This notation mostly originated from Reactive Extensions and can be used for giving examples of behaviours, but are less suitable for creating new behaviours.

**Data Flow Diagrams** These diagrams give a model for how the data flows from sources through connectors to destinations. This notation is more suitable for describing behaviours and is similar to the notation that we use.

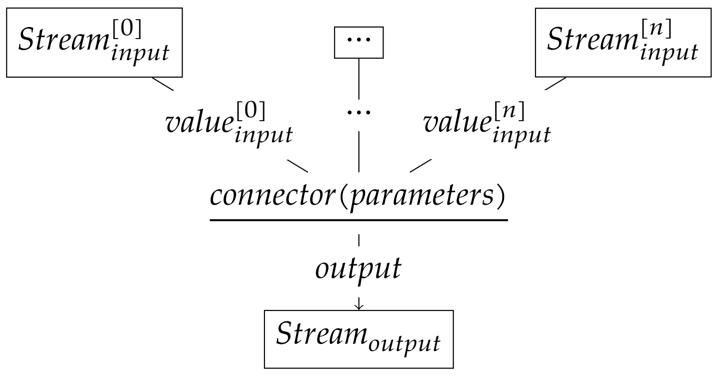


FIGURE 4.3: Merging

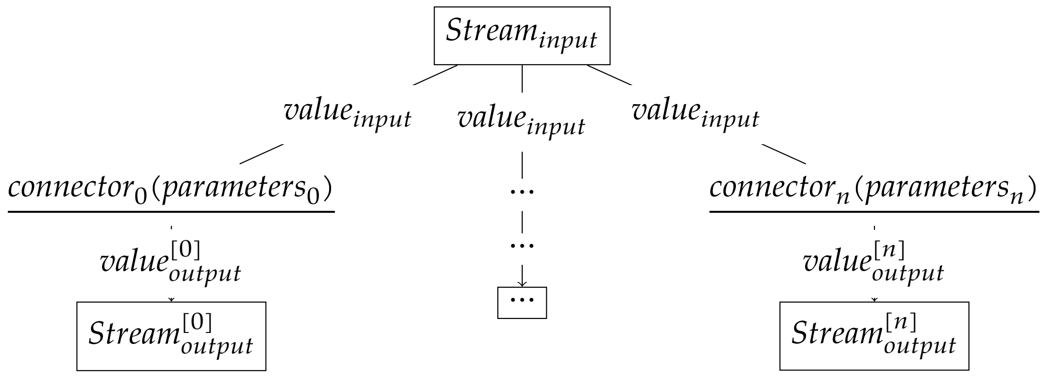


FIGURE 4.4: Branching

**Flowcharts** These diagrams give a model for how the control flows between steps. This notation is more suitable for describing sequential behaviour, and is hence different from the notation that we use.

**Functional programming notation** Functional programming gives denotational semantics for specifying programs. These map closely to mathematical formula and can be used to proof the validity of a functional program. Our mathematical notation is inspired by functional programming notation but does not fully comply with it.

The RRP Graph is a model of robot behavior that separates the concerns of sensing, planning and acting. FIGURE 4.2 depicts the graph respresentation. SUBFIGURE 4.2A shows that sensors can send data to a stream, which in this case will be called a sensor stream. SUBFIGURE 4.2B shows that sensors can be connected using connectors that take parameters. Zero, one or multiple parameters can be defined for a connector. SUBFIGURE 4.2C shows that streams can send data to an actuator, in which case the stream is called an actuator stream.

Sensing means getting information from the environment, either proprioceptive or exteroceptive. Proprioceptive sensing means retrieving data from sensors that measure some property of the robot. Examples of such sensors are rotary sensors for getting joint angles or temperature sensors for getting joint temperature. Exteroceptive sensing means retrieving data from sensors that measure some property from outside of the embodiment of the robot. Examples of such sensors are vision (via a camera), audio (via a microphone) or touch (via bumpers or capacitive sensors). In RRP sensors are not just low level hardware components

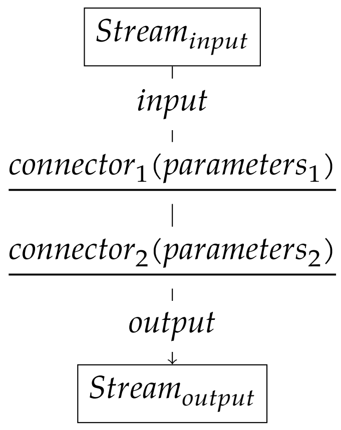


FIGURE 4.5: Anonymous stream

but also higher level software components such as Computer Vision algorithms. Two frequently used sensors for example are a red ball sensor and a color blob sensor. The robot can be programmed to start sensing by adding a Sensor Stream.

Acting means acting on to the environment, again either on the internal environment of the robot or on the external environment. In the case of acting this distinction is however less clear, as a change in the internal environment of the robot will often lead to a change on the external environment. For example if a robot has to pick up a ball that lies in front of the robot, it needs to control its joint angles so as to move to a desired location. There are also side effects of internal actuation. For example a robot can stiffen its joints, which might not change the pose of the robot but will make the motors of the robot produce more heat. Some actions the robot might take are hard to observe externally, such as the robot storing a face in its memory. The robot can act by sending an input to an Actuator Stream.

To process sensor data the programmer can add connectors to streams. This way a Directed Acyclic Graph is formed, in which the sensor streams are sources, streams with no sensor nor actuator are regular vertices and actuator streams are sinks. Connectors are labeled edges. The parameters given to the connectors are modeled as properties of the edges.

FIGURE 4.3 shows how multiple streams can send their data to a single connector. In this case the connector defines how to synchronize the data from the multiple input streams. RRP currently supports a simple merge in which the latest value of the input sensor is used, and a more complicated merge in which a procedural parameter specifies how to combine the inputs.

The dual of merging, i.e. splitting, is not explicitly supported, however branches can be created by having multiple connectors taking inputs from the same input stream. FIGURE 4.4 depicts this. Additionally one could imagine connectors that have multiple inputs and outputs. In order to keep the paradigm simple RRP does not currently support these.

To reduce the size of graphs anonymous streams can be used. In this case streams in between sensor and actuator streams can be hidden. FIGURE 4.5 depicts this.

In the next section we will introduce the connectors which are currently supported.

## 4.2 Procedural Parameters

To be able to offer a small set of high level connectors we make use of Procedural Parameters. A procedural parameter is a parameter that itself is a procedure [30]. Procedures are higher order functions, which means that they can be treated as any other variable [10]. To ensure validity, functions are expected to be pure, that means that they are free of side effects [71]. If functions are inpure they can make changes to a system that can not be seen by merely looking at the graph.

An example of using procedural parameters is the following (in pseudocode):

```
1: $numbers = list(1 \dots 10)$
2: $square = number \rightarrow number^2$
3: $numbers.foreach(square)$
4: $numbers \Leftrightarrow list(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)$
```

Line 1 defines a list of numbers from 1 to 10. Line 2 defines a function that takes as input a number and returns as output the square of the number. Line 3 uses an operator called *foreach* of the list of numbers to which the lambda function square is given as argument. Line 4 shows the resulting value of the list *numbers*, which got squared. Note that in this example *square* is a function that is free of side effects, however *foreach* has side effects as it modified the list to which it is applied.

The same example can be implemented in Python as follows:

```
1 >>> numbers = list(range(1,11))
2 >>> square = lambda number: number**2
3 >>> squared = map(square, numbers)
4 >>> list(squared)
5 [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

The notation used for this example is equal to doctest, in which a line starting with a triple greater-than sign (>>>) is a line inputted and a line not starting with a triple greater-than sign is expected output. Line 1 defines a list of numbers ranging from 1 to 11 (exclusive). Line 2 defines a function that takes as input a number and returns as output the square of the number. The syntax for lambda functions in Python is defined as *lambda parameters : body* where *parameters* is a list of parameters separated by a comma. If no parameters are supplied this can be omitted. Body is the body of the function, and can contain a single Python statement. Line 3 defines a variable called *squared*, which contains the return value of applying the *square* function and the *numbers* list to the *map* function. *map* is a built-in function of the Python language. Line 4 converts the *squared* variable to a list. Line 5 shows the output, a list of squared numbers. Note that in this example, unlike the example in pseudocode, the list of numbers has not been altered. This is because unlike the foreach operator such as used in the pseudocode example, the Python example uses the map function that produces a new list of numbers and is hence free of side effects. Other languages, such as JavaScript, actually implement an *array.forEach* function, but in the Reactive Programming style use of such functions with side effects should be avoided.

## 4.3 Connectors

The following connectors are currently implemented:

**map** Uses a procedural parameter to map inputs to outputs. Useful for example for doing a calculation on the input (e.g. a coordinate transform or distance calculation) or selecting a specific value from a structure (e.g. taking the value from a field of an object or looking up an index in an area).

**filter** Uses a procedural parameter to filter inputs which do not match a certain predicate. Useful for implementing range filters (e.g. filtering objects that are close or far) and category filters (e.g. filtering objects that have been seen before or are new; filtering faces that have been recognized or have not been recognized).

**timestamp** Adds a timestamp to the inputs. Useful for reasoning based on time (e.g. when combining two streams, ensure a maximum time difference between two events).

**sample** Samples an input at a certain rate. Useful for reducing unneeded computation and avoiding overloading an actuator.

**combineLatest** Uses a procedural parameter to combine inputs from multiple streams. Useful for sensor fusion.

**merge** Merges inputs from multiple streams. Useful when multiple streams produce the same type of output (e.g. when having redundant sensors such as two sonar sensors).

**subscribe** Sends inputs to an actuator. Used whenever the robot should actually make a change to change the environment.

I also overload the operator $\rightarrowtail$ to mean "passes messages to", that is:

$$A \rightarrowtail connector(parameters) \rightarrowtail B$$

Where $A$ and $B$ are streams, means that "$A$ passes messages to a *connector* parameterized by *parameters* that passes messages to $B$". Alternatively one could define a tuple $(A, connector, parameters, B)$, e.g. $(A, subscribe, \emptyset, B)$ means that "$A$ passes messages to the *subscribe* connector, which takes no parameters, that passes messages to $B$". In Python we would write: $B = A.subscribe()$. The message from $A$ to the connector is an output of $A$ and an input to the connector given the specified parameters. The message from the connector to $B$ is an output of the connector and an input to $B$.

### 4.3.1 Map

It will often occur that the output of a sensor needs to be transformed in some ways to be usuable as input for an actuator. One example is when the position of an object in the camera frame might need to be transformed to a position in the end-effector frame (inverse kinematics). Another example is when a data structure

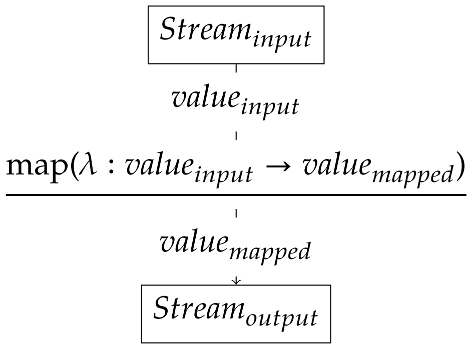


FIGURE 4.6: Diagram of the map connector

is given and the programmer is only concerned with a subset of the data given by one field of the structure. Yet another example is when a list of items is given and the programmer is only interested in one element of the list for which the programmer knows the index. A final example is when an object is given and the programmer wants to perform some calculation over the object, for example getting the distance to an object.

In all of these cases the *map* connector can be used. This connector applies to each input value a procedural parameter $\lambda$ and puts the result in the output stream.

**Definition 4.3.1.** $Stream_{input} \rightarrowtail map(\lambda : value_{input} \rightarrow value_{mapped}) \rightarrowtail Stream_{output}$: Takes as input value $value_{input}$ from $Stream_{input}$. Produces output $value_{mapped} = \lambda(value_{input})$ to $Stream_{output}$. Also see FIGURE 4.6.

## 4.3.2 Filter

Depending on the application, not all sensor data might be of interest. For example, a programmer might only be interested in objects that are either close or that are far. It might also be the case that a programmer is only interested in objects of a specific category, or in objects that do not fit in that category, e.g. faces that are new and faces that have been seen before.

In these cases the filter connector can be used to remove spurious data. This connector applies to each input value a procedural parameter $\lambda$ and puts that same value in the output stream if $\lambda$ returned true.

**Definition 4.3.2.** $Stream_{input} \rightarrowtail filter(\lambda : value_{input} \rightarrow value_{filter}) \rightarrowtail Stream_{output}$: Takes as input value $value_{input}$ from $Stream_{input}$. Produces output $value_{input}$ to $Stream_{output}$ if $value_{filter} = \lambda(value_{input}) = True$. Also see FIGURE 4.7.

## 4.3.3 Timestamp

When reasoning based on time it is beneficial to add a timestamp to data. This is especially useful if the sensor that produces the data does not provide a timestamp already. Theoretically, in realtime applications one might even want to add a timestamp to sensor data that already contains a timestamp from the sensor, to ensure that the timestamp will be equal throughout the RRP Graph. RRP does however not currently offer any realtime programming support.

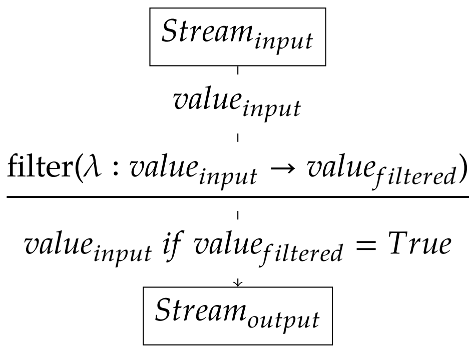

$Stream_{input}$
$value_{input}$
filter($\lambda : value_{input} \to value_{filtered}$)
$value_{input}$ if $value_{filtered} = True$
$Stream_{output}$

FIGURE 4.7: Diagram of the filter connector

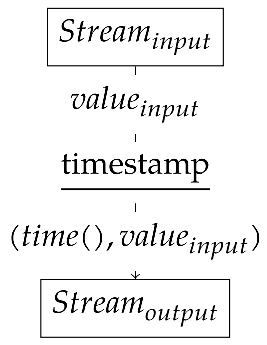


FIGURE 4.8: Diagram of the timestamp connector

One case in which timestamps can be used is when a programmer combines two streams and wants to ensure a maximum time difference between events being generated from the two streams. The CombineLatest connector, which will be introduced shortly, combines the latest known value from a set of streams. It could however be that one of these streams contains a very old value. By first timestamping the input streams, then combining the timestamped streams, and then filtering out values where there is a too big difference between the timestamps, a programmer can produce the desired result. This is quite an advanced use case which a novice programmer might not encounter.

The *timestamp* connector takes any item as input and puts the timestamped input into the output stream.

**Definition 4.3.3.** $Stream_{input} \rightarrowtail timestamp() \rightarrowtail Stream_{output}$: Takes as input a value $value_{input}$ from $Stream_{input}$.

Produces output $(time(), value_{input})$ to $Stream_{output}$, where $time()$ is a function that returns the current time. Also see FIGURE 4.8.

### 4.3.4 Sample

Some sensors can produce data as a rate which is unnecessarily high. This can cause computation overhead and can even overload actuators that accept inputs at a lower rate.

For example, the robot NAO has camera which produces frames with a resolution of 1280x960 pixels at 30 frames per second. When tracking objects (e.g. visual servoing) this is a decent frame rate. However when the programmer wants

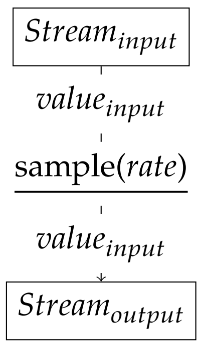


FIGURE 4.9: Diagram of the sample connector

the robot to switch tracking modes (for example, from head tracking to full body tracking) such a high frame rate is unnecessary. In this case the programmer can decide to use only one or two frames per second, however we do not want to change the capture rate of the camera itself as a high frame rate is required for the motor control of the actual tracking tasks.

To compensate for this a programmer can use the *sample* connector. *sample* takes the latest value, each *rate* milliseconds, from the input stream and puts this value in the output stream.

**Definition 4.3.4.** $Stream_{input} \rightarrowtail sample(rate) \rightarrowtail Stream_{output}$: Creates a timer with a duration of *rate* miliseconds. When the timer expires, takes as input a value $value_{input}$ from $Stream_{input}$, produces output $value_{input}$ to $Stream_{output}$ and restarts the timer. Also see FIGURE 4.9.

### 4.3.5 CombineLatest

Many applications require the combination of data produced by multiple sensors. For example to determine whether an object is on the left right side of the robot torso, the summation of an object's radian angle in the robot's camera frame (located in the head of the robot) and the head yaw rotation can be used. Such sensor fusion applications can be performed using the *CombineLatest* connector. This connector combines the latest value of one input stream with cached values of the other input streams. For each input value on one of the input streams, the *CombineLatest* connector retrieves the latest (cached) value of the other input streams and applies the function $\lambda$ on the values. It then inserts the results of the procedural parameter call in the output stream.

**Definition 4.3.5.** $Stream^{[0]}_{input}, \dots, Stream^{[n]}_{input} \rightarrowtail combineLatest(\lambda : value^{[0]}_{input}, \dots, value^{[n]}_{input} \rightarrow value_{combined}) \rightarrowtail Stream_{output}$.

Let $i$ be the index of the stream on which the latest input was received;

Then $Stream^{[i]}_{input}$ is that stream.

Takes as input a value $value^{[i]}_{input}$ from $Stream^{[i]}_{input}$ and stores this in the cache.

Produces output $\lambda(value^{[0]}_{input}, \dots, value^{[i]}_{input}, \dots, value^{[n]}_{input})$ to $Stream_{output}$, where $value^{[j]}_{input} \in \{input^{[0]}, \dots, value^{[n]}_{input}\}$ and $i \neq j$, is a cached value of $Stream^{[j]}_{input}$. Also see FIGURE 4.10.

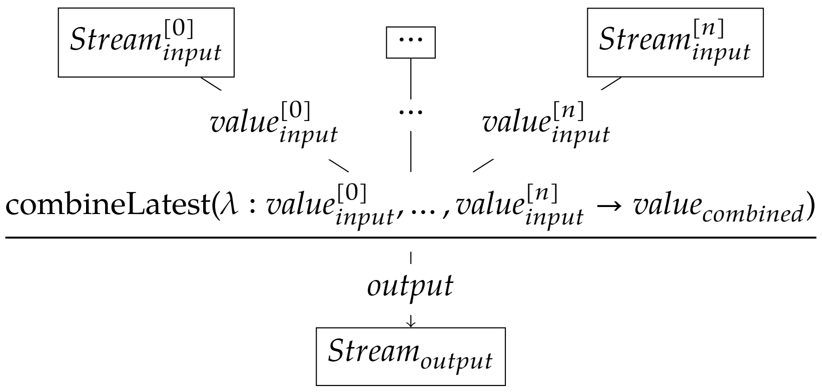


FIGURE 4.10: Diagram of the combineLatest connector

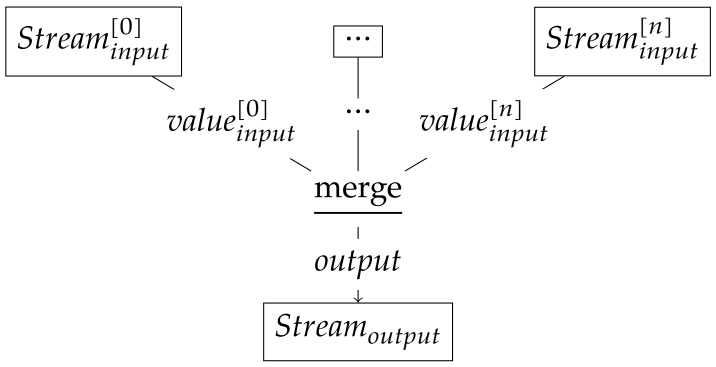


FIGURE 4.11: Diagram of the merge connector

### 4.3.6 Merge

When there are multiple sensors that produce similar data a programmer might want to merge these. For example, when detecting color balls of three colors these three ball sensor streams can be merged into a single stream. The *merge* connector outputs an input given to it from multiple streams.

**Definition 4.3.6.** $Stream^{[0]}_{input}, \dots, Stream^{[n]}_{input} \rightarrowtail merge() \rightarrowtail Stream_{output}$. Where:
Let $i$ be the index of the stream on which the latest input was received;
Then $Stream^{[i]}_{input}$ is that stream.
Takes as input a value $value^{[i]}_{input}$ from $Stream^{[i]}_{input}$.
Produces output $value^{[i]}_{input}$ to $Stream_{output}$. Also see FIGURE 4.11.

### 4.3.7 Subscribe

Most connectors in RRP are free of side effects, i.e. they cannot influence any value outside of the RRP Graph. This makes the RRP Graph easy to understand: Nothing is changed about the robot unless the programmer explicitly defines so. For robotics applications it is obvious that there is a need of side effects. To introduce side effects the programmer can use a special connector called subscribe. This connector sends its inputs to an actuator. Note that, as explained earlier, in RRP any change to the state both inside the robot or outside the robot is done via an actuator.

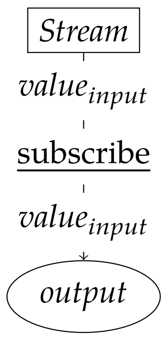


FIGURE 4.12: Diagram of the subscribe connector

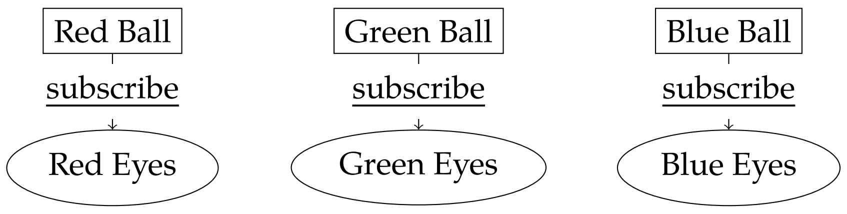


FIGURE 4.13: Reflect ball color sample application RRP Graph

**Definition 4.3.7.** $Stream_{input} \rightarrowtail subscribe() \rightarrowtail Actuator$. Takes as input a value $value_{input}$. Sends $value_{input}$ to $Actuator$. Also see FIGURE 4.12.

## 4.4 Sample Applications

In this section I will show various sample applications using the diagram notation used earlier in this chapter.

### 4.4.1 Reflect ball color

The goal of the robot is to look for a colored ball, and if spotted change the eye colors to reflect on the ball detected.

The following assumptions can be made:

- The robot detects balls of the colors red, green and blue.
- The following sensors are available: Red Ball, Green Ball, Blue Ball.
- The following actuators are available: Red Eyes, Green Eyes, Blue Eyes.

FIGURE 4.13 shows how this application can be implemented as an RRP Graph.

### 4.4.2 Say ball color

The goal of the robot is to look for a colored ball, and if spotted say the color of the ball detected.

The following assumptions can be made:

- The robot detects balls of the colors red, green and blue.

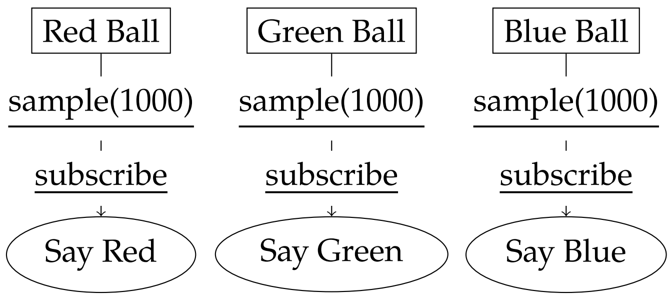


FIGURE 4.14: Say ball color sample application RRP Graph

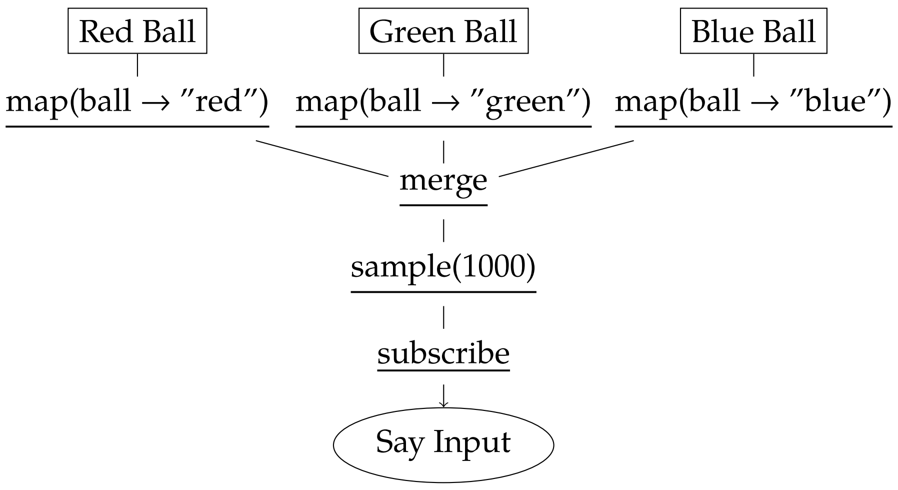


FIGURE 4.15: Say ball color with merge sample application RRP Graph

- The following sensors are available: Red Ball, Green Ball, Blue Ball.
- The following actuators are available: Say Red, Say Green, Say Blue.
- The robot has to say the color once per second.

FIGURE 4.14 shows how this application can be implemented as an RRP Graph.

### 4.4.3 Say ball color with merge

The say ball color implementation has some duplication, which can be removed by using the merge connector.

The following assumptions are made:

- The robot detects balls of the colors red, green and blue.
- The following sensors are available: Red Ball, Green Ball, Blue Ball.
- The following actuator is available: Say Input.
- The robot has to say the color once per second.

FIGURE 4.15 shows how this application can be implemented as an RRP Graph.

A map connector is used to create a text string for each ball color. In this case the input value (*ball*) is actually not used for generating the output (the string containing the color of the ball). Using the merge connector the ball strings are

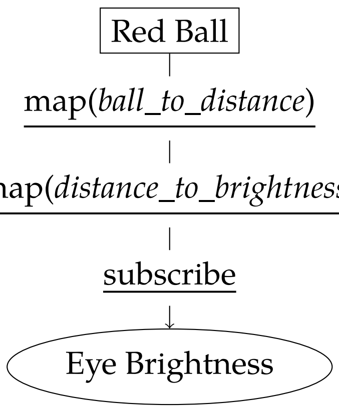


FIGURE 4.16: Distance to brightness sample application RRP Graph

merged into a single (anonymous) stream. Then a sample of this merged stream is taken every second using the sample connector. Finally using subscribe the ball string is send to an actuator that makes the robot say the input value.

### 4.4.4 Distance to brightness

The goal of the robot is to look for a red ball, and if spotted change the brightness of its eye LEDs based on the distance to the ball.

The following assumptions are made:

- The robot detects red balls using a *Red Ball* sensor.
- There is a procedure called *ball_to_distance*, which can calculate the distance to the ball given a ball as input.
- There is a procedure balled *distance_to_brightness*, which can calculate a brightness value given a distance as input.
- There is an actuator called *Eye Brightness*, which takes a brightness value as input and changes the intensity of the eye LEDs to the input.

FIGURE 4.16 shows how this application can be implemented as an RRP Graph.

### 4.4.5 Track far balls with head, close balls with head and body

The goal of the robot is to track far away balls with its head, and balls that are close with both its head and its body.

The following assumptions are made:

- The robot detects balls using a *Ball* sensor.
- There is a procedure called *ball_to_distance*, which can calculate the distance to the ball given a ball as input.
- There are the following two actuators: (1) *Head Tracker*, (2) *Head and Body Tracker*.

FIGURE 4.17 shows how this application can be implemented as an RRP Graph.

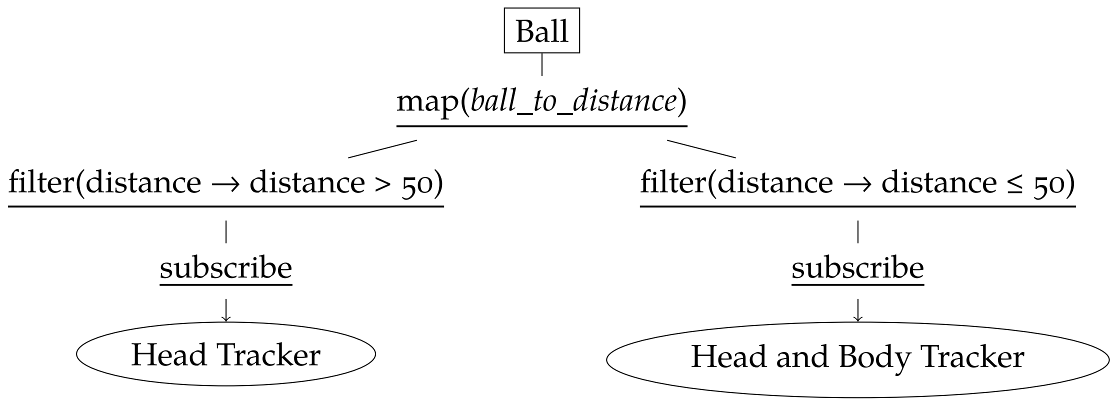


FIGURE 4.17: Simple tracking sample application RRP Graph

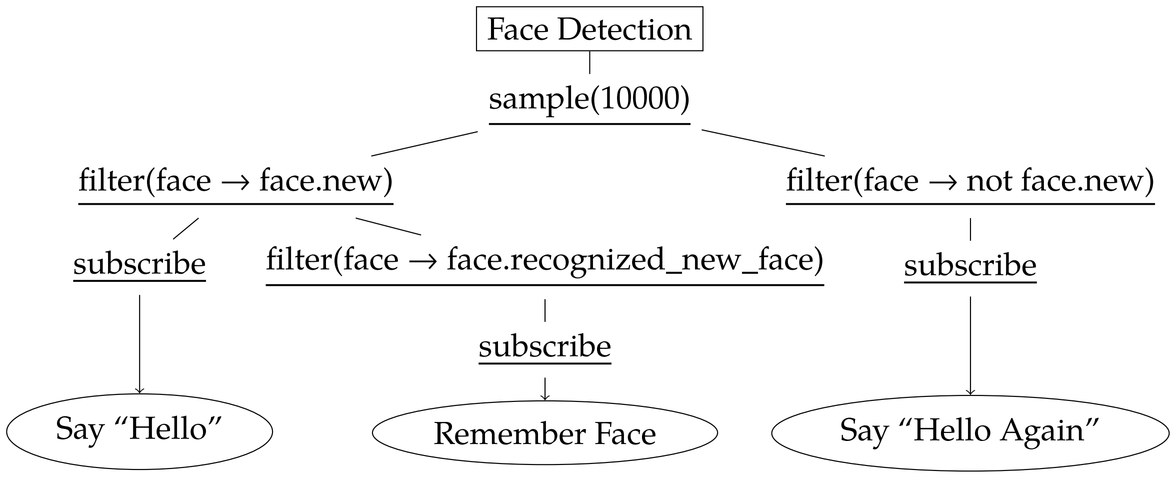


FIGURE 4.18: Greeting sample application RRP Graph

### 4.4.6 Greeter

The goal of the robot is to greet people, either saying "Hello" to people who the robot has not seen before, and "Hello Again" to people who the robot has seen before.

The following assumptions are made:

- The robot detects faces using a Face Detection sensor.
- Faces have a field called *new* which is *True* if the face has not been seen before.
- Faces have a field called *recognized_new_face* if a new face was detected and the quality of the detection was good enough to remember the face.
- There are the following three actuators: (1) Say "Hello", (2) Say "Hello Again", (3) Remember Face.
- The robot checks for faces every 10 seconds.

FIGURE 4.18 shows how this application can be implemented as an RRP Graph.

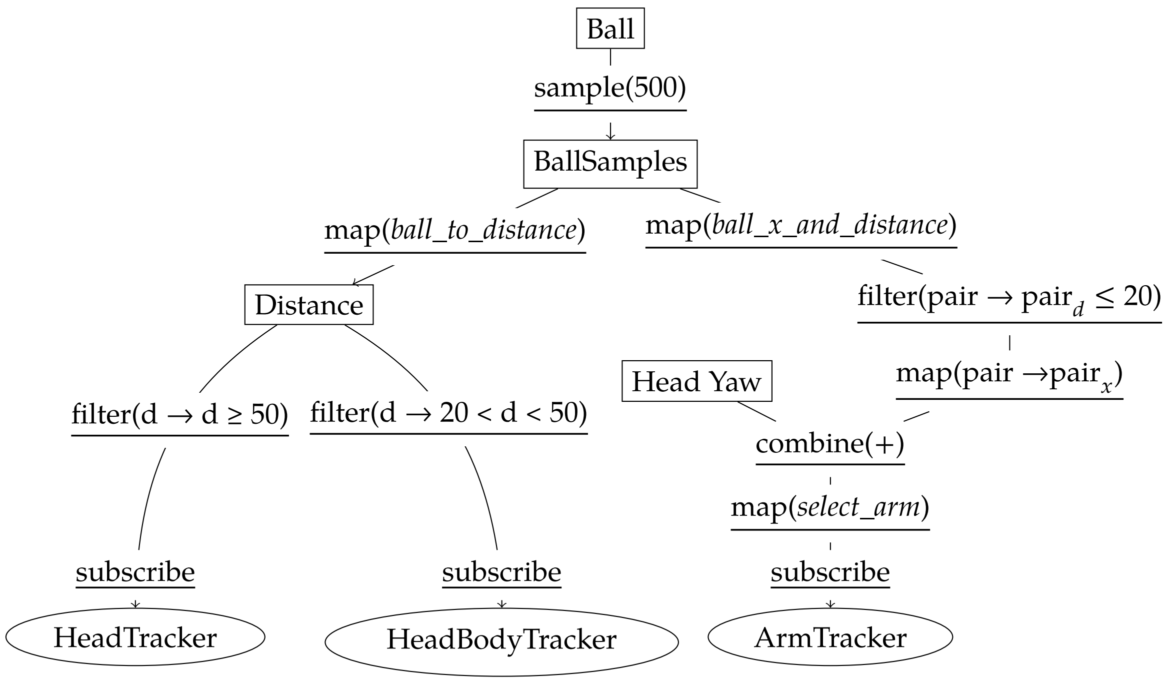


FIGURE 4.19: Complex tracking sample application RRP Graph

### 4.4.7 Track far balls with head, medium balls with head and body, close balls with nearest arm

The goal of the robot is to track far away balls with its head, medium distance balls with the head and body, and close distance balls with the arm nearest to the ball.

The following assumptions are made:

- The robot detects balls using a Ball sensor.
- The head yaw angle is retrieved using a Head Yaw sensor.
- There is a procedure called *ball_to_distance*, which can calculate the distance to the ball given a ball as input.
- There is a procedure called *ball_x_and_distance*, which can calculate the distance to the ball given a ball as input and returns a pair of distance $d$ and ball x-angle $x$.
- There is a procedure called *select_arm*, which given an input angle relative to the torso frame of the robot returns either "left" if the angle is negative or "right" if the angle is positive.
- There are the following three actuators: (1) Head Tracker, (2) Head and Body Tracker and (3) Arm Tracker, where the Arm Tracker actuator accepts as input value the arm to use for tracking.

FIGURE 4.19 shows how this application can be implemented as an RRP Graph.

### 4.4.8 Fixing a timing bug in the tracking example

The previous example contains a bug. If a ball has moved from close to medium or far distance, the *combine(+)* connector will have a cached version of the ball. When the head of the robot moves, the *Head Yaw* sensor stream will produce a new angle. The new head yaw angle will be combined with an old ball angle, the arm to use for tracking the ball at its old angle will be computed and the arm tracker will be enabled with that arm value. This will happen even though the ball is not at a close distance! A solution to this problem requires the use of the timestamp operator.

By adding a timestamp to the value of the head yaw and ball position we can check whether the ball has been close to the robot in a fixed time period. For example, we can say that if a ball was spotted closely to the robot in the previous second, we will assume the ball is still close. This also requires us to modify the parameter to the combine operator to take into account that it now gets two pairs as input:

- Ball angle and timestamp
- Head yaw and timestamp

We define the following procedure for combining these: combinator $= h, b \rightarrow (h_v + b_v, h_t - b_t)$ in which $h$ is the head yaw, $b$ is the ball angle, and the timestamp connector makes pairs with the keys $v$ as value and $t$ as timestamp. Just as in the previous example, ball angles are combined using a summation. For the timestamp we substract the time that the ball was detected from the time that the head moved. This is the time between seeing a ball and moving the head. If the ball was seen after moving the head, this value will be negative.

The following procedure can now be used to filter out cases in which the head moved but a ball has not been detected as close within the last second: $a \rightarrow a_t < 1sec$. After filtering out spurrious head yaw movements we can discard the timestamp. This is done in a similar way as that the distance was discarded earlier in the graph.

FIGURE 4.20 shows how the bug has been fixed in the tracking example.

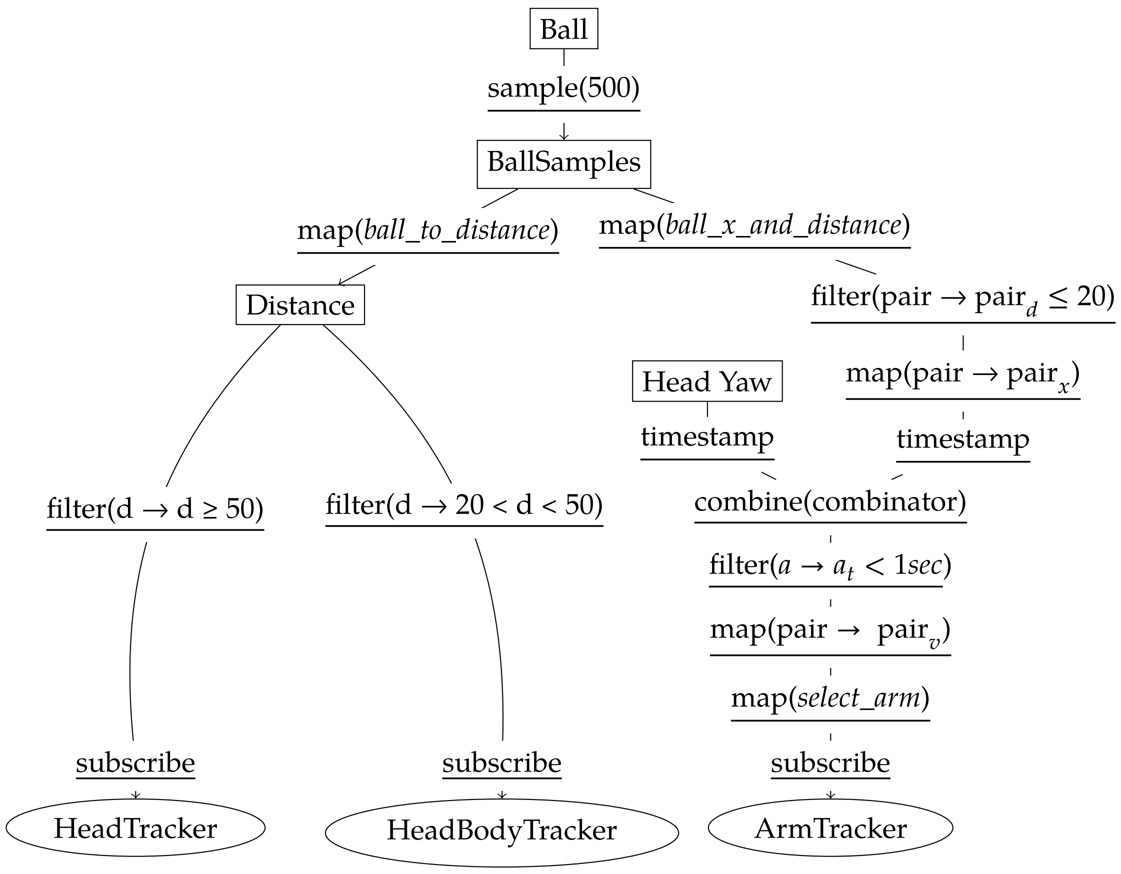


FIGURE 4.20: Complex tracking sample application RRP Graph

# Chapter 5

# Interpreter for Reactive Robot Programming

RRP is a programming paradigm which can currently be applied by using an interpreter. The interpreter can load an RRP Graph and execute it on a robot. The RRP Interpreter is aimed towards more experienced users, novice users do not directly have to interact with the interpreter as they can use a Visual Programming Environment, which is explained in the next chapter.

In this chapter we present the requirements of the interpreter and how it was implemented.

## 5.1 Requirements

The requirements of the interpreter are as follows:

- Runs on a control PC.
- Can load an RRP Graph from various data sources.
- Can run an RRP Graph on various robots.
  - Initialize sensors;
  - Process events from sensors as specified by the RRP Graph;
  - Send outputs to actuators.
- Supports the following connectors:
  - Map;
  - Filter;
  - Timestamp;
  - Sample;
  - CombineLatest;
  - Merge;
  - Subscribe.
- Includes functionality for exploring the behaviours stored in a data source.

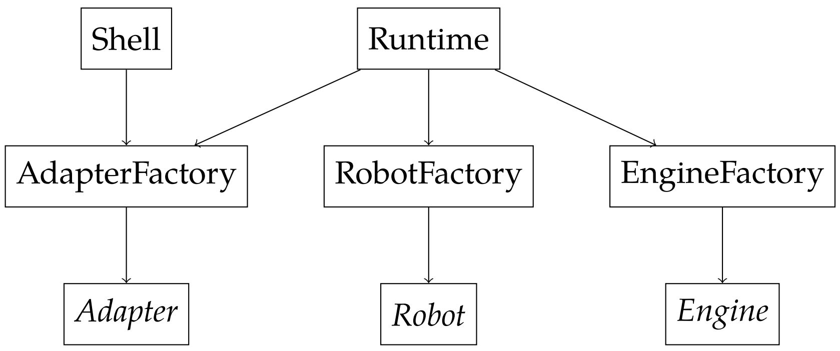


FIGURE 5.1: High level overview of the interpreter architecture.

- It should be easy to add support for a new robot.
- It should be easy to add support for a new data source.

## 5.2 Implementation

Currently RRP supports one robot, one data source and a limited set of sensors and actuators on the robot. Because the RRP Interpreter is continuously being developed, this section shows a snapshot of the current state.

The RRP Interpreter was developed using the Python programming language with support for the Softbanks NAO robot and the Neo4j Graph Database as data source. It includes a shell for exploring the behaviours stored in a data source, and a runtime environment for running behaviours on the robot.

FIGURE 5.1 gives a high level overview of the interpreter architecture.

### 5.2.1 Shell

The shell is an interactive console application that allows an operator to explore the behaviours stored in the data source and also to stimulate sensors on a running instance of an RRP Graph on the robot. TABLE 5.1 shows a list of the commands supported by the shell.

### 5.2.2 Runtime environment

The runtime environment can initialize an RRP Graph on the robot. To accomplish this, it uses an engine that specifies the threading behavior. By using a factory method an engine can be created. This section discusses the standard multithreaded engine. An experimental single threaded engine using an event loop is also under development, as well as an experimental engine using multiprocessing.

The engine itself runs in a thread, that when started initializes the helpers for a graph and loops over the sensor streams of a graph. The sensor streams are the streams that have a sensor attached to them. The engine will initialize the sensor streams, the connectors and the actuator streams.

The implemented model of RRP Graphs closely matches the description in the previous chapter. As could be seen in the examples, there are two ways of imple-

Table 5.1: Commands supported by the shell

| Command | Description |
|---|---|
| Connect | Connect to an RRP daemon |
| Disconnect | Disconnect from an RRP daemon |
| Stimulate | Send input to a sensor |
| Show programs | Show programs (behaviours) in the data source |
| Load program | Load a program for exploring in the shell |
| Show streams | Show all streams in the program |
| Load stream | Loads a stream for exploring in the shell |
| Show out streams | Shows all outgoing streams from the loaded stream |
| Show in streams | Shows all incoming streams to the loaded stream |
| Show out ops | Show all outgoing connectors from the loaded stream |
| Show in ops | Show all incoming connectors to the loaded stream |
| Show start streams | Show all streams that have a sensor attached |
| Show commands | Show a list of supported commands |
| Exit | Exit from the shell |

menting a procedural parameter, namely by declaring the procedural parameter directly in the graph or by referring to an existing procedure. In the RRP Interpreter referring to an existing procedure is done by defining the procedure as a *helper*. Helpers are initialized by the RRP Interpreter before the streams are initialized.

Stream initialization follows this algorithm:

1. If the stream is a sensor stream:
   (a) Get the sensor from the robot model;
   (b) Initialize the sensor if it is not currently running.
2. If the stream is an actuator stream:
   (a) Get the actuator from the robot model;
   (b) Initialize the actuator if it is not currently running.
3. Create an instance of the stream and store this for later use;
4. Loop through the outgoing connectors for this stream:
   (a) If the output stream is not initialized yet:
      i. Initialize the stream using this algorithm;
      ii. Initialize the connector using a connector factory, connect it to the initialized stream and store it for later use.
   (b) If the output stream is already initialized (then the connector has to be a many to one connector, as a stream can have only one input connector):
      i. Retrieve the connector instance;
      ii. Adds the current stream as source to the connector;
      iii. Connect the current stream to the connector.

The engine automatically creates threads for the sensors used and manages the subscriptions for connectors. The map, filter, timestamp, combineLatest and merge connectors inherit the thread of their input stream. The sample connector runs in its own thread as it is maintaining a timer.

After the RRP Graph is initialized the sensors will be active on the robot and will feed data to their connectors. The currently implemented connectors behave as follows:

**Map**
- Initialization:
  1. If the procedural parameter is a body, evaluate it using Python's *eval* function and store it.
  2. If the procedural parameter is a helper, it has already been initialized when the engine initialized the graph, so store the reference to the helper.
- Input received:
  1. Pass the input to the procedural parameter;
  2. Pass the result of the previous step as input to the output stream.

**Filter**
- Initialization:
  1. If the procedural parameter is a body, evaluate it using Python's *eval* function and store it.
  2. If the procedural parameter is a helper, it has already been initialized when the engine initialized the graph, so store the reference to the helper.
- Input received:
  1. Pass the input to the procedural parameter;
  2. If the result of the previous step was True, pass the input to the output stream.

**Timestamp**
- Initialization: Store the start time.
- Input received: Pass a tuple containing the input and the current time minus the start time to the output stream.

**Sample**
- Initialization: Start a timer that fires according to the sample period.
- Input received: Set the latest value to the input.
- Timer fires:
  1. If there is a latest value stored, pass it to the output stream and reset the latest value.
  2. If there is no latest value stored, do nothing.

**CombineLatest**
- Initialization: Initialize a dictionary with as keys each input stream to store the latest value received from them.
- Input received:
  1. Store the input in the dictionary with as key the input stream;
  2. If an input has been received on every input stream, pass the input values from the dictionary to the procedural parameter;

Table 5.2: Actuated joints of the robot

| Joint | Supported rotation | | |
|---|---|---|---|
| Head | | Pitch | Yaw |
| Right hand | | Open & Close | |
| Right wrist | | | Yaw |
| Right shoulder | Roll | Pitch | |
| Right elbow | Roll | Pitch | YawPitch |
| Right hip | Roll | | Yaw |
| Right knee | | Pitch | |
| Right ankle | Roll | Pitch | |
| Left hand | | Open & Close | |
| Left wrist | | | Yaw |
| Left shoulder | Roll | Pitch | |
| Left elbow | Roll | | Yaw |
| Left hip | Roll | Pitch | YawPitch |
| Left knee | | Pitch | |
| Left ankle | Roll | Pitch | |

3. Pass the result of the previous step as input to the output stream.

**Merge**
- Initialization: No specific initialization needed.
- Input received: Send the input to the output stream.

**Subscribe**
- Initialization: No specific initialization needed.
- Input received: Send the input to the actuator associated with this subscription.

Another behavior worth describing is the termination of an RRP Graph. This is initialized when the operator sends the termination signal to the RRP Runtime. Because there are various threads running these have to be shut down to cleanly terminate. Basically the RRP Runtime will walk through the RRP Graph and in post-order stop the streams and their input connectors. For sample connectors the RRP Runtime will also stop the timer thread. For sensor streams the RRP Runtime will stop the sensor thread. Finally the engine thread will be stopped.

### 5.2.3 Robot: Softbank NAO

Table 5.2 shows the joints of the robot for which the angle can be sensed and actuated. In the case of right hip yawpitch and left hip yawpitch only a single motor is used for actuation.

The following sensors are currently supported:

**Red ball** Detects red balls in the camera frame.

**Blob** Detects blobs of a specified color in the camera frame.

**WordRecognized** Detects words from a dictionary.

**FaceDetection** Detects faces in the camera frame.

**JointPosition** Gets the position of a joint, typically in radians, but for hands this is a percentage.

The following actuators are currently supported:

**EyeLedColor** Changes the colors of the eye LEDs.

**RememberFace** Remembers a detected face for later recognition.

**SayInput** Uses TTS to say a message inputted.

**SayParameter** Uses TTS to say a message given as parameter when initializing the actuator.

**Tracker** Changes tracking mode to a parameter given when initializing the actuator.

**SetJointPosition** Changes position for a joint specified as parameter when initializing the actuator to a joint angle given as input to the actuator. Allows for joint groups to be used and arrays of angles as input.

**Print** Prints a message to the console.

While adding sensors and actuators is not complicated, currently an experienced programmer is expected to provide a robot specific framework. The currently supported sensors and actuators were added for the purpose of creating sample applications and conducting experiments. For end-users to use RRP some cooperation with an experienced software engineer is needed to add the sensor and actuator support for the desired application domain.

### 5.2.4 Data Source: Graph Database

Currently the only supported data source is a graph database called Neo4j. Graph databases store data in a graphical format called a labeled property graph. Labeled property graph have the following properties [37]:

- They contain nodes (vertices) and relationships (edges).
- Nodes have properties associated with them, which are defined as key-value pairs.
- Nodes can be assigned one or more labels.
- Relationships have a name, a direction and a start and end node.
- Just like nodes, relationships can have properties.

Neo4j is a schemaless database, which means that any desired structure has to be enforced by applications reading and writing from/to the graph. FIGURE 5.2 shows an abstract description of the graph database model used. Nodes are drawn as circles and relationships as arrows. Properties are collected inside a rectangle

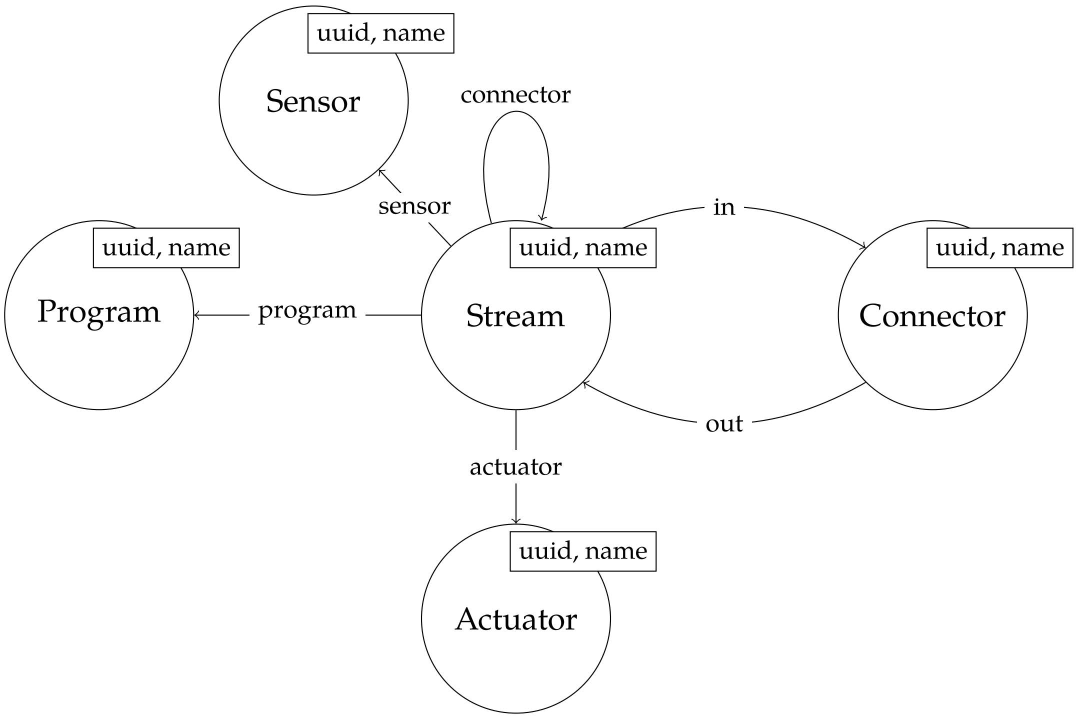


FIGURE 5.2: Abstract description of the graph database model

in the corner of the node to which they belong. Properties on relationships are ignored in the figure. Connectors can be modeled either as a relationship between two streams or as a node. The following connectors are modeled as relationships: Map with body, filter with body, sample, timestamp and subscribe. The following connectors are modeled as nodes: Map with helper, filter with helper, combineLatest and merge.

Neo4j supports querying the graph database using a query language called Cypher. For example, the following Cypher query returns a program with its streams:

```
MATCH (p:Program)<-[:program]-(s:Stream)
WHERE id(p) = 351
RETURN p, s
```

# Chapter 6

# Visual Programming Environment for Reactive Robot Programming

The RRP Interpreter introduced in the previous chapter provides the technical infrastructure for executing RRP Graphs on a robot. End-users expect a more friendly environment, which is why a Visual Programming Environment was developed. As has been shown earlier, many Visual Programming Environments for learning programming and for novice programmers already exist, however few of them are aimed towards programming robots.

At the same time, the reactive programming paradigm is often teached by using diagrammatic notations, some examples have been shown in CHAPTER 4. These diagrams are however not executable, and besides that are aimed towards teaching more experienced programmers about reactive programming. We introduce a tool for creating executable diagrams of applications according to the principles of RRP. Whereas the technical term for these diagrams is RRP Graphs, this chapter refers to RRP Graphs using the word "behaviours". Our tool is called the RRP VPE, but this chapter refers to it as "the VPE".

This chapter discusses the requirements of the VPE and shows how we implemented it.

## 6.1 Requirements

This section splits the requirements of the VPE between *functional* (about functionality offered by the VPE) and *non-functional* (about technical features of the VPE) requirements. The functional requirements are:

- Allows the management of behaviours (adding, removing, editing).
- Allows modelling of behaviours:
  - Adding/removing/editing sensor streams;
  - Adding/removing/editing connectors;
  - Adding/removing/editing actuator streams.
- Allows the management of reusable code fragments (adding, removing, editing).

- Allows starting/stopping applications on the robot.

The non-functional requirements are:

- Implemented as a web application.
- Supports both mouse and touch interaction.
- Allows for multiple users to interact with the VPE at the same time.
- Modern interaction without page refreshes.

## 6.2 Implementation

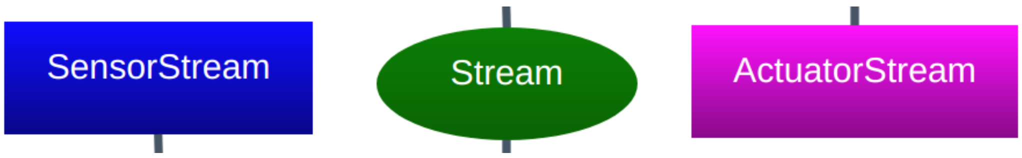
We implemented the VPE as a web application using Node.js and the ExpressJS framework. The Pug template language is used for specifying views. The VPE has been implemented as a Single Page Application, which allows for a very smooth user experience without any page refreshes. Communication between the client side (web browser) and server side (Node.js/ExpressJS web application) is performed using the socket.io library. The user interface has been developed using the Bootstrap CSS framework, JSPlumb graphing library and KnockoutJS data binding library. Care was taken to support both mouse and touch interaction in the user interface, by for example not using specific mouse events and having elements of adequate size for touch interaction.

On the server side major domain constructs such as behaviours, streams, connectors and helpers each have their own module that handles communication with the client, a model that specifies their structure and a data access object (DAO) that reads and writes to the graph database. On the client side there are also modules to communicate with the server side modules, as well as models for controlling behaviour and data.

The usage of web sockets and broadcasting of changes made to behaviours enables multi-user support, which means that users can work together. A limitation of the current system however is the lack of locking mechanisms, which means that to avoid errors from occuring two users should coordinate their changes.

### 6.2.1 Mapping behaviours to RRP Graphs

RRP Graphs do not directly map to behaviours in the VPE because the VPE does not currently support anonymous streams. This means that whenever we used an anonymous stream in the RRP Graph examples, for example between two connectors, for the corresponding behaviour in the VPE we added an extra stream. One benefit of this approach is that the input value for procedural parameters can always be derived from the input stream name, and can hence be omitted from the diagram. To ensure that the user can write stylistically correct Python code, stream names are converted to arguments using the following procedure: The first letter is made lowercase, the first letter after a space is capitalized and spaces are removed. For example, a stream named "Red ball" will be offered as argument as "redBall".



(A) Sensor stream (B) Stream (C) Actuator stream

FIGURE 6.1: Three types of streams in RRP

FIGURE 6.1 shows the three types of streams supported in RRP: Sensor streams (6.1a), ordinary streams (6.1b) and actuator streams (6.1c). The programmer can recognize streams in multiple ways:

- By color: Blue for sensor streams, green for ordinary streams, magenta for actuator streams.
- By location: Sensor streams are on the top of a branch, ordinary streams are in the middle of a branch, actuator streams are at the bottom of a branch.
- By incoming/outgoing connectors: Sensor streams only have outgoing connectors, ordinary streams have both outgoing and incoming connectors, actuator streams only have incoming connectors.
- By shape: Sensor streams and actuator streams are squares, ordinary streams are ovals.

## 6.3 Walkthrough

FIGURES 6.2 TO 6.15 show a walkthrough of adding a simple behaviour; starting to track an object when it is close to the robot. Some steps have been omitted from the process for the sake of brevity.

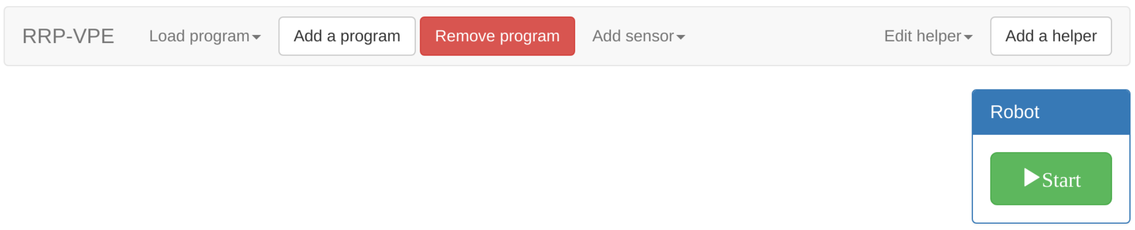


Figure 6.2: A blank program (behaviour) has been loaded.

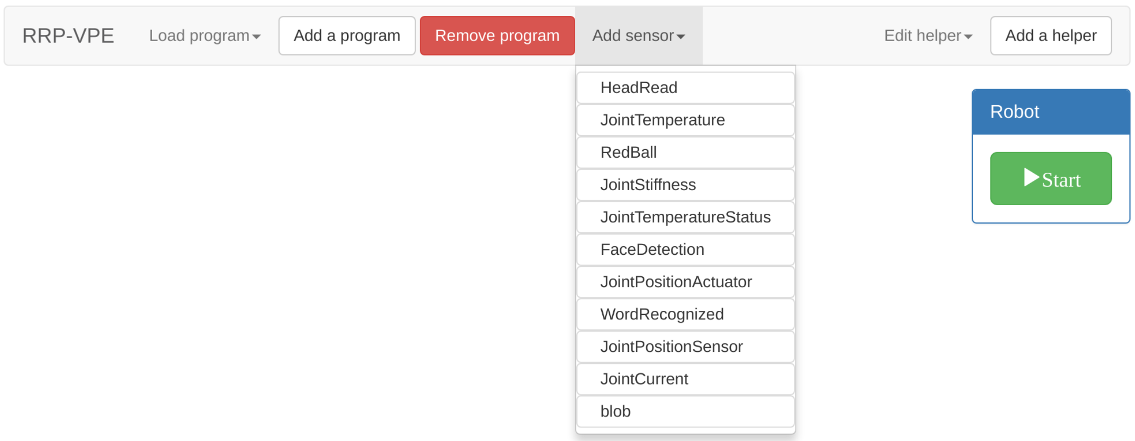


Figure 6.3: By clicking on "Add sensor", a list drops down in which the user can select the sensor to add. Assume the user selects *RedBall*.

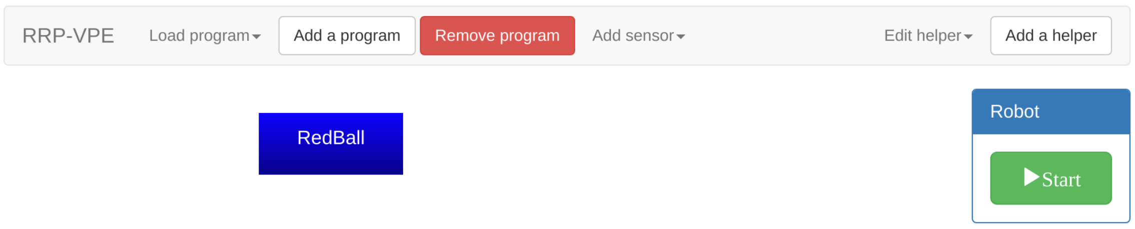


FIGURE 6.4: A sensor stream for the selected sensor has been added to the behaviour.

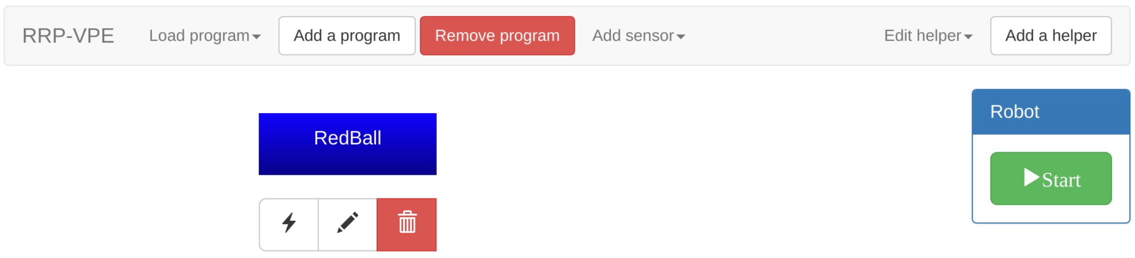


FIGURE 6.5: After having added a sensor stream the user can select it and a menu appears below it. For sensor streams this menu has three buttons: Thunderbolt: To add a connector starting from the sensor stream; Pencil: To change parameters of the sensor stream; Trash can: To remove the sensor stream.

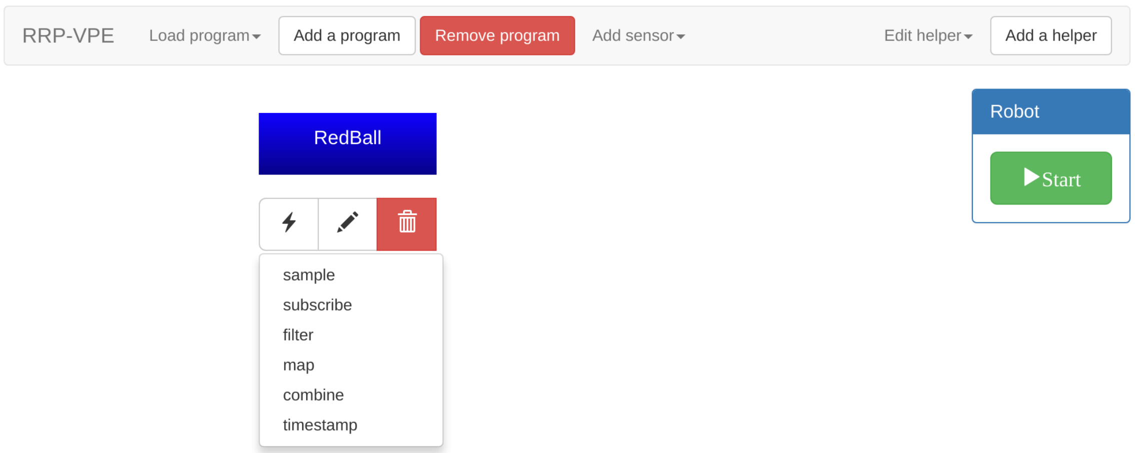
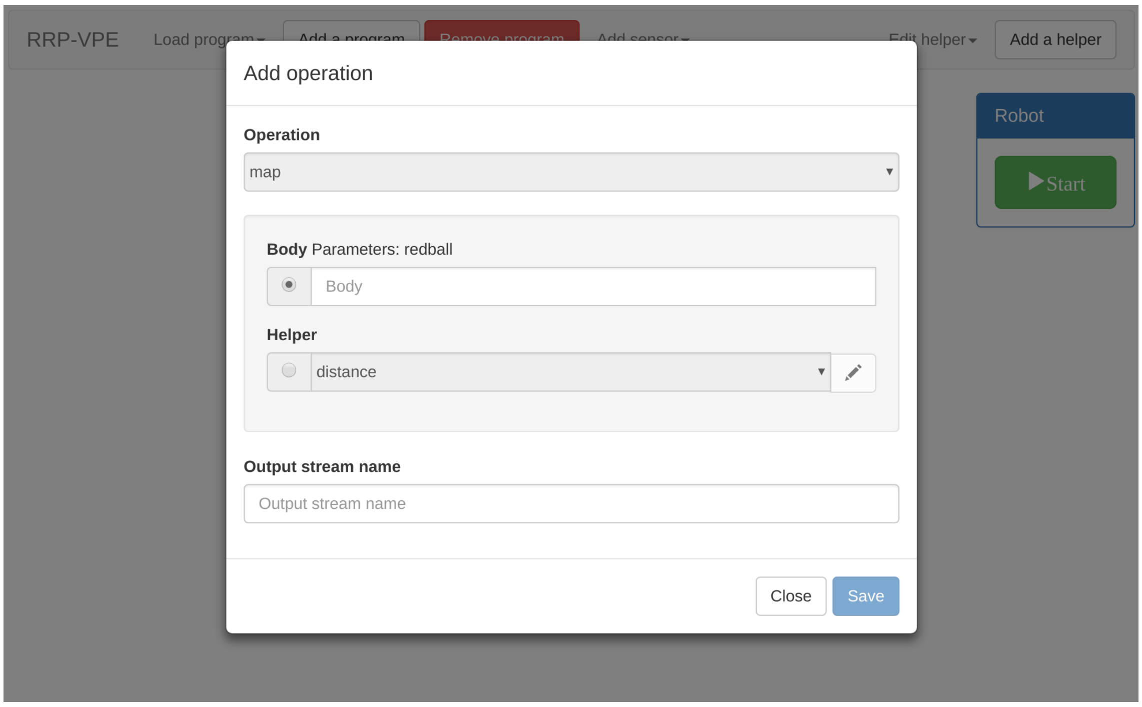


Figure 6.6: After clicking on the thunderbolt icon a menu dropped down showing the connectors that can be added to the behaviour.



Add operation

**Operation**

map

**Body** Parameters: redball

Body

**Helper**

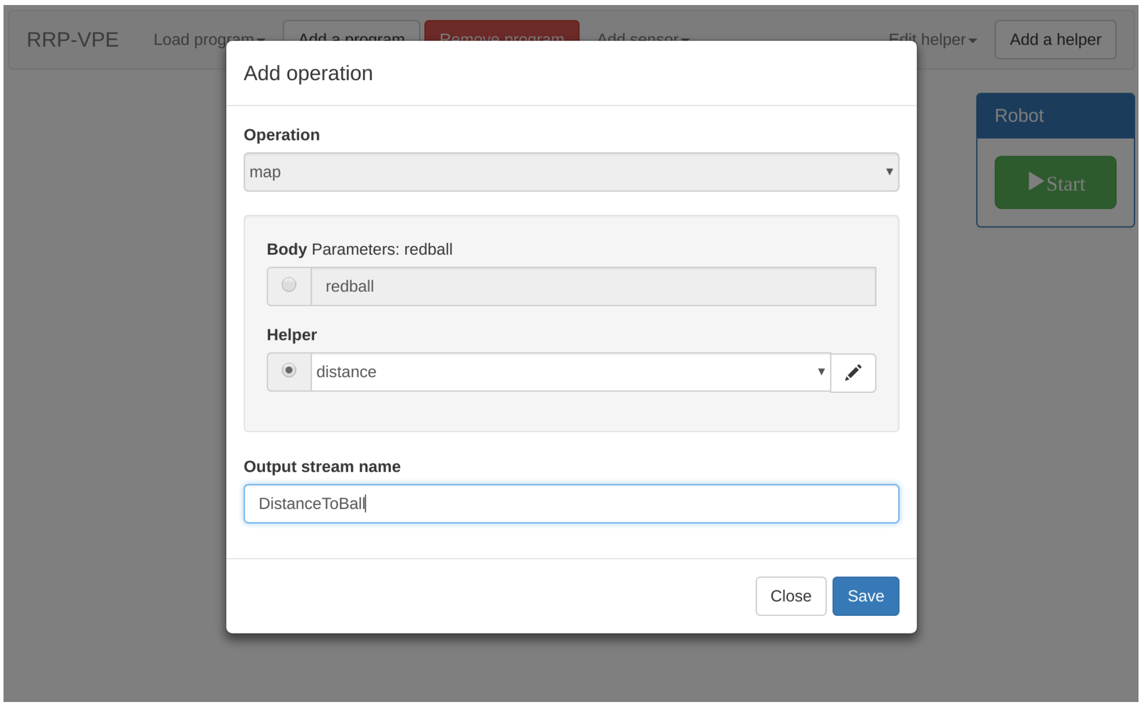
distance

**Output stream name**

Output stream name

Close Save

Figure 6.7: After clicking on the map connector a modal window opens in which the user can specify the procedural parameter and enter a name for the output stream to be generated. For specifying the procedural parameter the user has two options: (1) Specify the procedural parameter as a body; and (2) Select a helper to use as procedural parameter.



Add operation

**Operation**

map

**Body** Parameters: redball

redball

**Helper**

distance

**Output stream name**

DistanceToBall

Close Save

FIGURE 6.8: The user selects the helper *distance*, enters *DistanceToBall* as output stream name and clicks on Save.

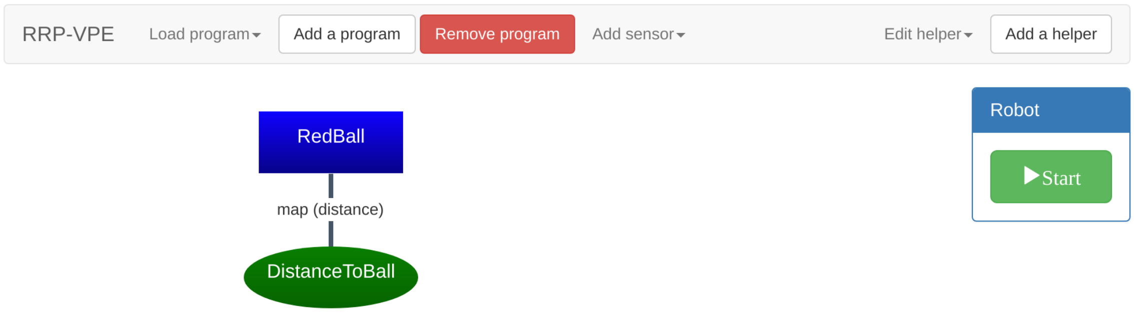
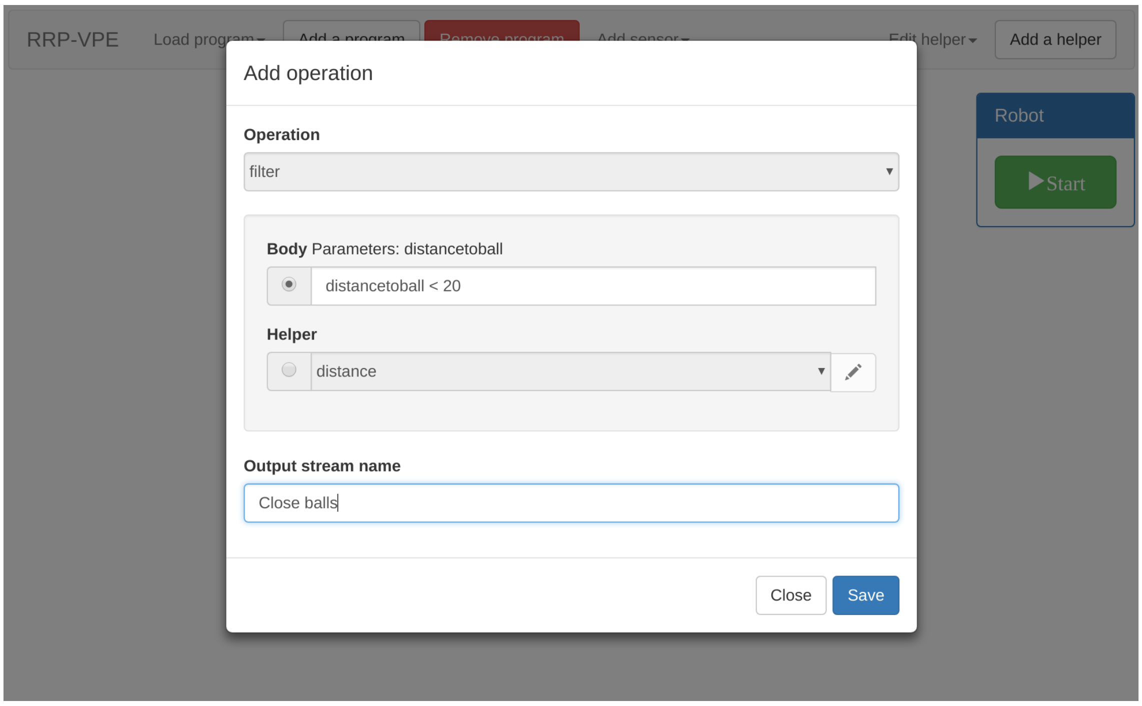


FIGURE 6.9: An output stream has been added and the RedBall sensor stream has been connected with the output stream using a map connector taking the helper distance as procedural parameter.



Add operation

Operation: filter

Body Parameters: distancetoball

● distancetoball < 20

Helper

○ distance

Output stream name

Close balls

FIGURE 6.10: The user has clicked on the *DistanceToBall* stream, followed by clicking on the thunderbolt icon and selecting the filter connector and entering *distancetoball* < 20 as body and *Close balls* as output stream name.

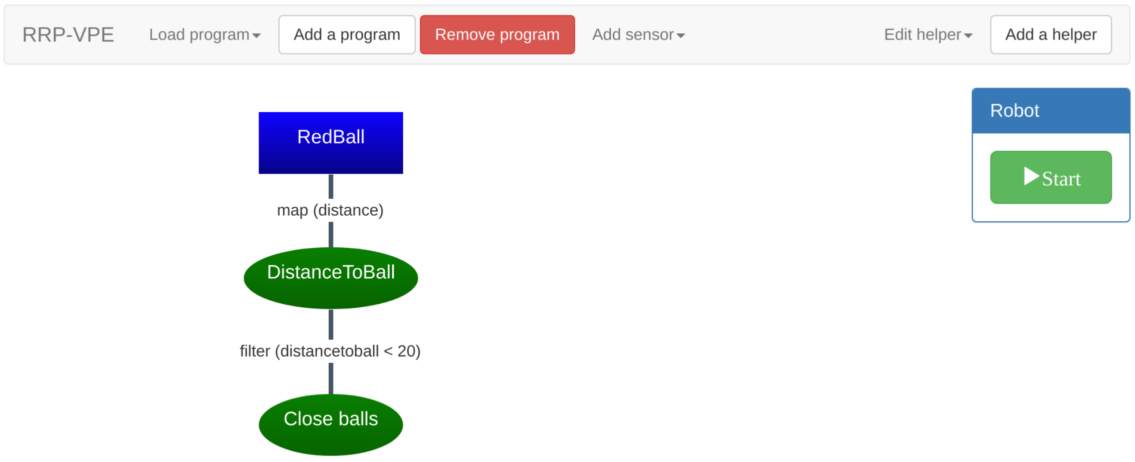
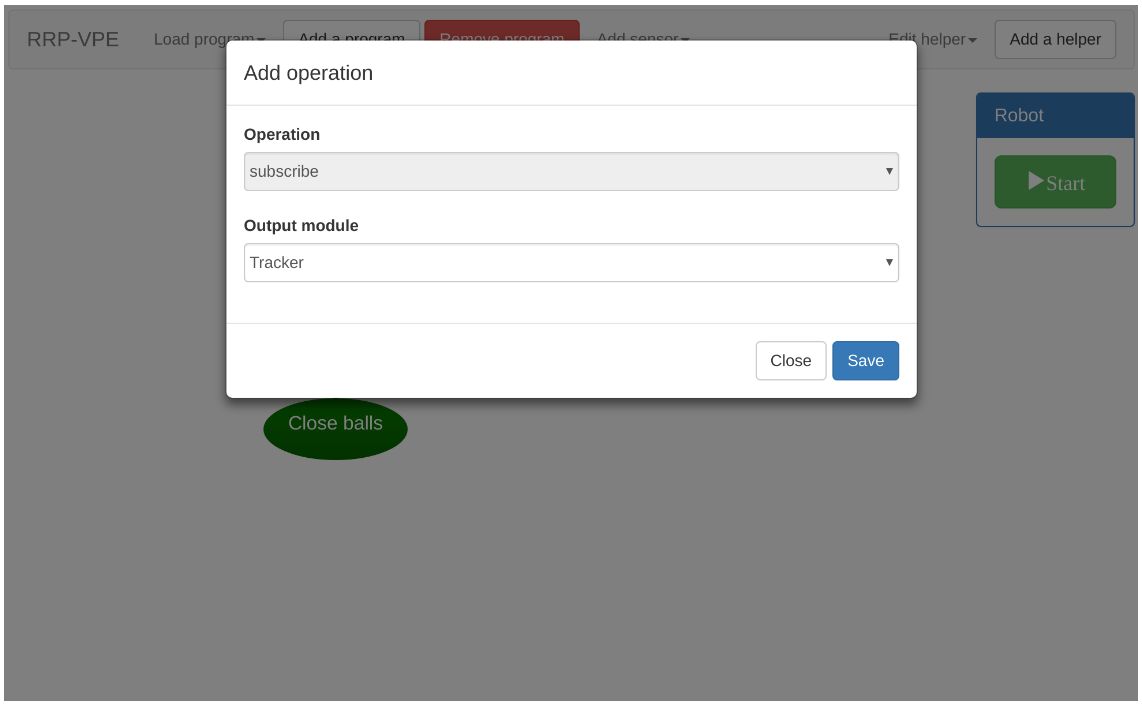


FIGURE 6.11: After clicking on Save, an output stream has been added and the *DistanceToBall* stream has been connected to the *Close balls* stream with a *filter* connector taking as procedural parameter *distancetoball* < 20.



Add operation

Operation: subscribe

Output module: Tracker

Close | Save

Figure 6.12: The user has clicked on the *Close balls* stream, followed by clicking on the thunderbolt icon and selecting the subscribe connecor and selecting the Tracker output module.

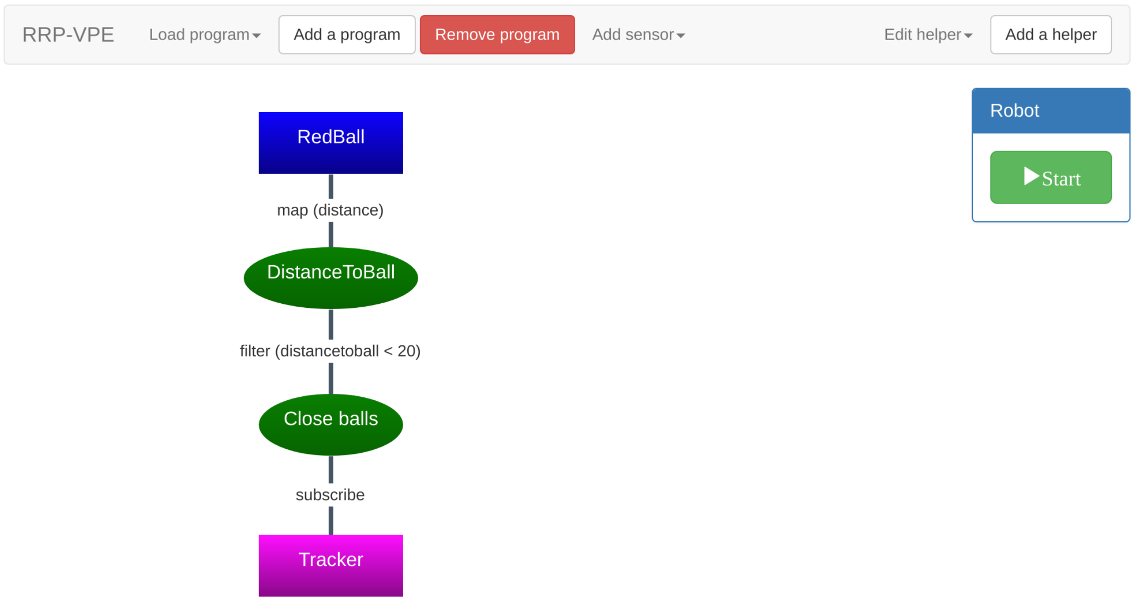


Figure 6.13: After clicking on Save, an actuator stream has been added and the *Close balls* stream has been connected to the *Tracker* actuator stream with a *subscribe* connector.

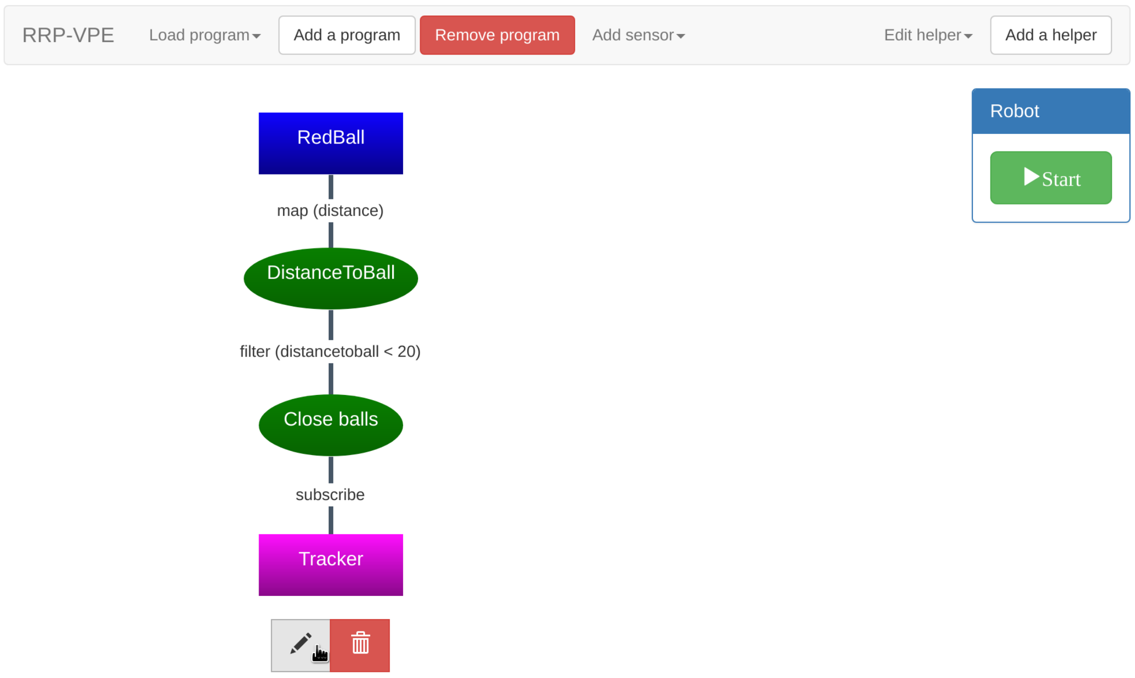
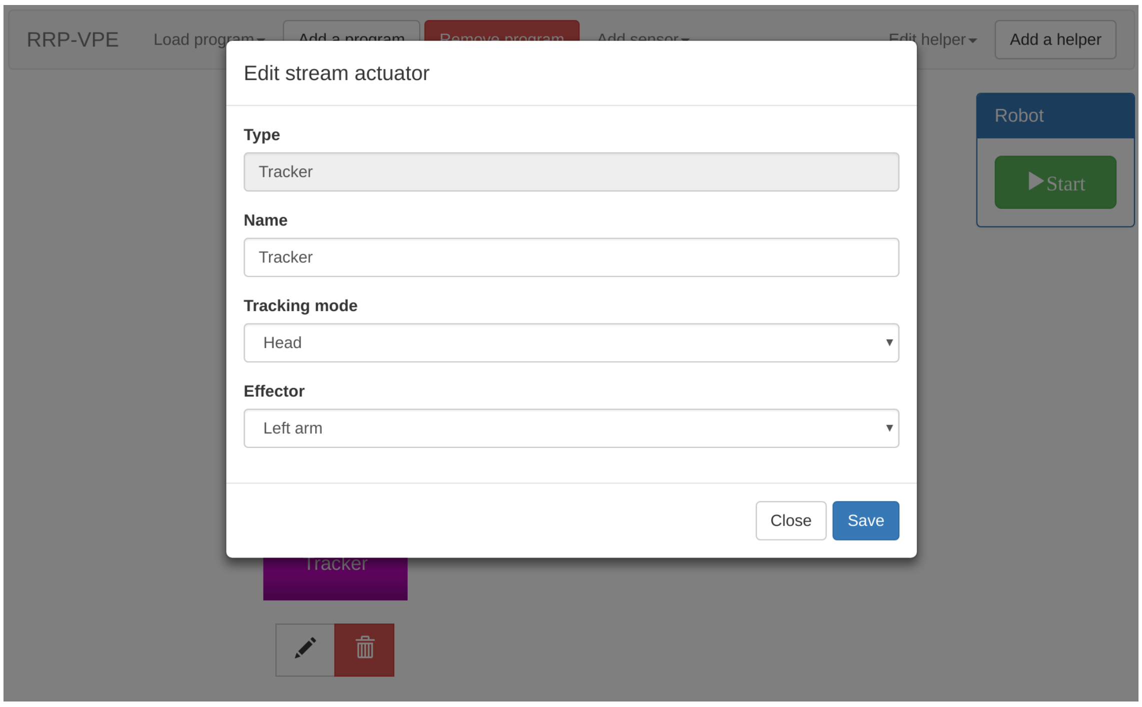


FIGURE 6.14: The user has clicked on the *Tracker* actuator stream and clicks on the pencil icon from the dropdown menu.



Edit stream actuator

Type
Tracker

Name
Tracker

Tracking mode
Head

Effector
Left arm

FIGURE 6.15: The user selects the *Head* as tracking mode and the *Left arm* as effector. Then the user clicks on Save. After this the program is ready to be run on the robot.

# Chapter 7

# User Experiment

We performed a user study to verify the effectiveness of the VPE for end-users. This chapter introduces the method used and the results obtained.

## 7.1 Method

We recruited eight participants (mean age 28, standard deviation 3, 6 male, 2 female) who were students and faculty in laboratories of the University of Tsukuba, to take part in a comparative study using the Visual Programming Environment for RRP and a state-of-the-art programming tool for novice programmers, namely Choregraphe. All of these participants had experience with doing experiments in the field of Human-Robot Interaction. Additionally we recruited one participant who is a therapist (age 30, male). For this therapist the experimental conditions differed slightly, and hence the data from this participant is not used in the mean/standard deviation in any table and not included in any graph. At the start of the experiment we asked participants to self-evaluate their skills regarding various topics related to programming and robotics.

During the experiment participants were asked to perform three tasks using each tool:

1. An explanation task, in which the experimenter showed the participant a behaviour diagram and asked the participant to explain how the robot would behave when running the program.

2. A debugging task, in which the experimenter showed the partipant a behaviour diagram that contained a mistake and asked the participant to explain what was the mistake in the diagram.

3. A creation task, in which the experimenter described a behaviour of the robot to the participant, and asked the participant to create the behaviour.

For the explanation task the participant had to explain a diagram for a different behaviour in each tool. Differences in the explanation task performance were hence both caused by difference in the actual behaviour and differences in the diagram notation of the tools. For the debug task the participant had to find a different mistake in a behaviour diagram for the same desired behaviour in each tool. Differences in the debug task performance were hence both caused by differences in the mistake in the behaviour diagram and differences in the diagram

TABLE 7.1: Experimental conditions

| Property | Condition | | | |
|---|---|---|---|---|
| | $\alpha$ | $\beta$ | $\gamma$ | $\delta$ |
| First tool | Choregraphe | RRP | Choregraphe | RRP |
| Second tool | RRP | Choregraphe | RRP | Choregraphe |
| Explain task (CHOR) | Say ball color | | Mirror ball color | |
| Explain task (RRP) | Mirror ball color | | Say ball color | |
| Debug task | Distance to brightness | | | |
| Create task | Tracking | | | |

notation of the tools. For the creation task the participant had to create a diagram for the same behaviour in each tool. Differences in the creation task performance were hence only caused by the different notation on the tools.

The experiment was counter-balanced by having half of the participants first perform the tasks with RRP and then with Choregraphe, and the other half first with Choregraphe and then with RRP. This was to correct for a learning effect which can occur because the tasks were similar and the tools have some common characteristics. Another factor for which counter-balancing was applied was for the explanation task, as it was not sound to make the subject explain the same behaviour twice.

In the case of Choregraphe, for each task the experimenter explained the boxes used for the behaviour. In the case of RRP, the experimenter explained the sensors streams, intermediary streams, actuator streams and connectors. The experimenter gave the participant 10 minutes to complete each task. In the case of participant nine, the maximum duration for the creation task was extended to 15 minutes in advance of the experiment. After performing the task the experimenter asked the participant to fill in a self-evaluation survey of the workload. The survey used was a standard survey from the NASA-TLX assessment tool. We did a "raw TLX" evaluation, as we did not perform a pairwise comparison [56].

The experimenter gave an instruction manual to the participant. During the experiment the participant could look back at the instructions when needed.

## 7.2 Tasks

Because of counterbalancing on both the tool factor and the explanation task factor there are four different experimental conditions, described by TABLE 7.1. The behaviours performed by the participants and referred to in the table are as follows:

1. Explain:
   - Mirror ball color: Color eyes of the robot based on detection of colored balls.
   - Say ball color: Make the robot say the color of colored ball detected.
2. Debug (Distance to brightness): Change the brightness of the robot eye LEDs based on distance to a ball.

Table 7.2: Prior skills of the participants according to a likert scale obtained by self evaluation. Mean does not include participant 9.

| Skill | Participant | | | | | | | | | Mean |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9* | |
| Visual programming | 2 | 2 | 2 | 1 | 4 | 4 | 5 | 3 | 2 | 2.5 |
| Robot programming | 4 | 2 | 1 | 2 | 4 | 3 | 5 | 4 | 1 | 3.5 |
| NAO programming | 4 | 2 | 1 | 1 | 4 | 4 | 5 | 3 | 1 | 3.5 |
| Choregraphe | 3 | 1 | 1 | 1 | 4 | 4 | 5 | 3 | 1 | 3 |
| Complex Event Processing | 4 | 1 | 1 | 1 | 4 | 2 | 5 | 2 | 1 | 2 |
| Functional Programming | 1 | 3 | 1 | 1 | 4 | 3 | 5 | 3 | 1 | 3 |
| Reactive Programming | 4 | 1 | 1 | 1 | 4 | 2 | 5 | 2 | 1 | 2 |
| Median | 4 | 2 | 1 | 1 | 4 | 3 | 5 | 3 | 1 | |

3. Create (Tracking): Make the robot track far balls with its head, close balls with its body and arm.

Each experimental condition was assigned to two participants in the experiment.

Figures 7.1, 7.2, 7.3 and 7.4 show the behaviour diagrams that participants either had to explain, debug or create.

## 7.3 Results

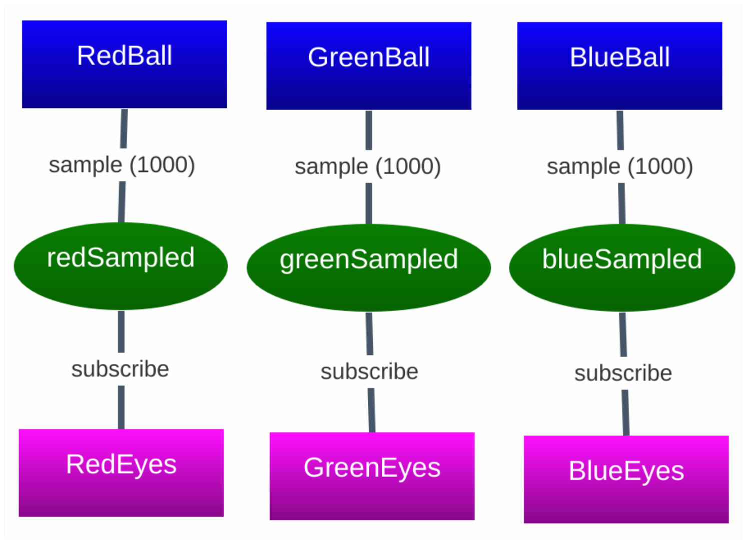
Table 7.2 shows how subjects evaluated themselves in terms of skills that might influence the experimental results. From this data it can be seen that there was a wide variety in the skill levels of the participants, ranging from having very little experience with most skill (e.g. participants 2, 3, 4 and 9) to having a high level of experience with most skills (e.g. participants 1, 5 and 7). The median skill level of the participants was average. We also note that some participants that are HRI researchers (participants 2, 3 and 4) rate their skills similarly to the therapist (participant 9) that we recruited for this experiment.

### 7.3.1 Task Duration

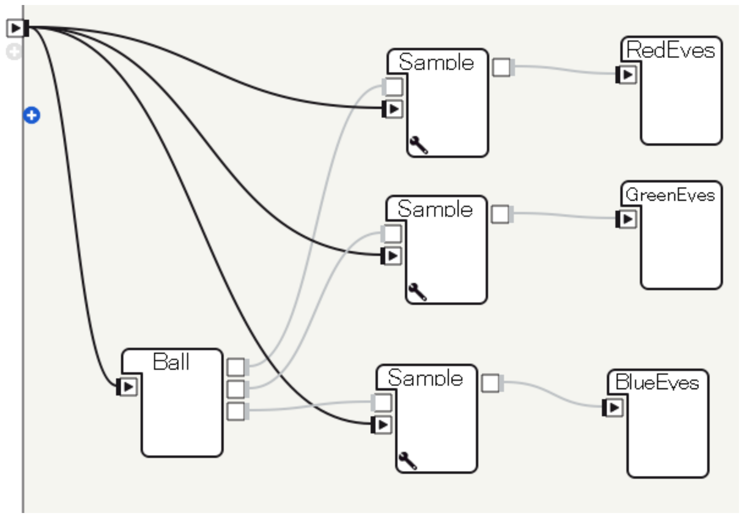
Table 7.3 shows the time it took for each participant to complete the tasks using each tool. In the table headers, CHOR stands for Choregraphe. Figure 7.5 graphs the mean and standard deviation of the duration for each task.

The time taken to complete each task increased as the tasks became harder. As mentioned earlier, the participants were given ten minutes to complete each task (15 minutes for the creation task for participant 9). The text DNC signals that a participant was not able to complete the task in time or made a mistake which invalidates their result. The latter was the case for one participant using RRP in the debugging task, one participant using Choregraphe in the debugging task and for each participant using Choregraphe in the creation task.

For the explanation task and the debugging task there was no statistically significant difference between using RRP and Choregraphe. For the creation task

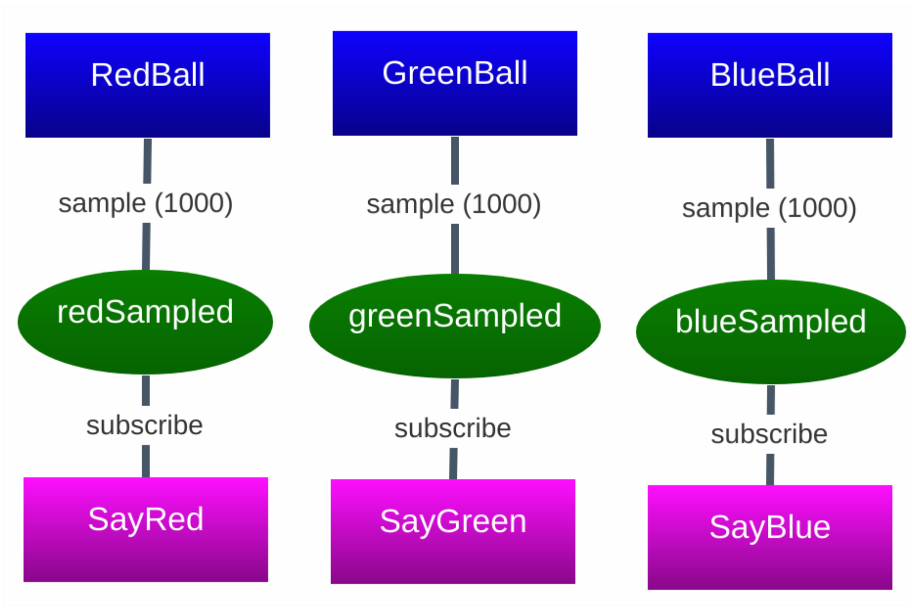


(A) Explain task A implemented using RRP

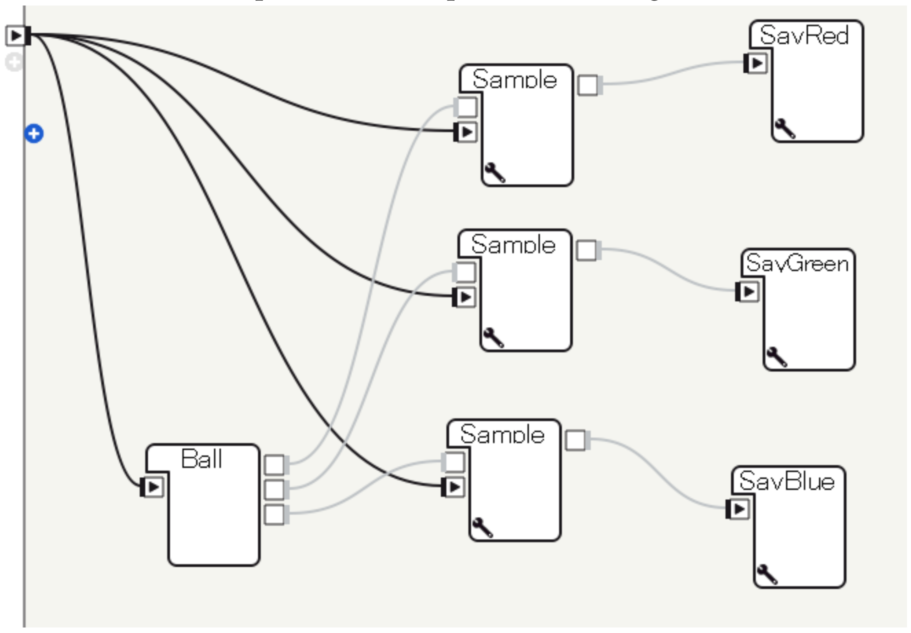


(B) Explain task A implemented using Choregraphe

FIGURE 7.1: RRP and Choregraphe implementations of Explain task A

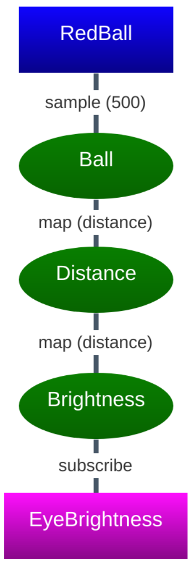


(A) Explain task B implemented using RRP

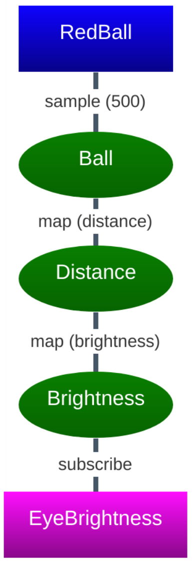


(B) Explain task B implemented using Choregraphe

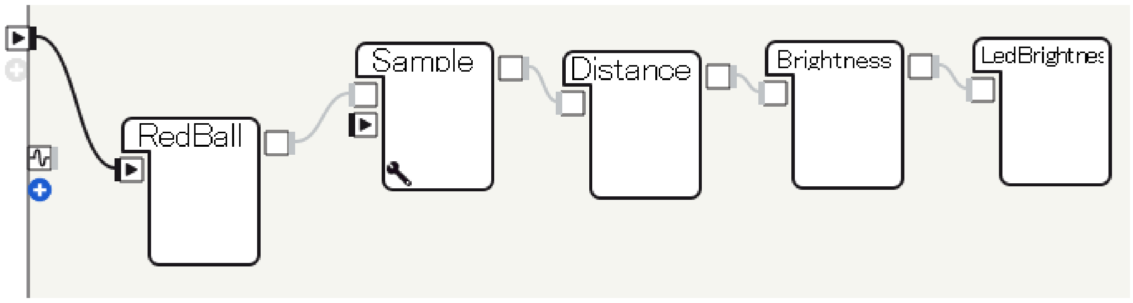
FIGURE 7.2: RRP and Choregraphe implementations of Explain task B



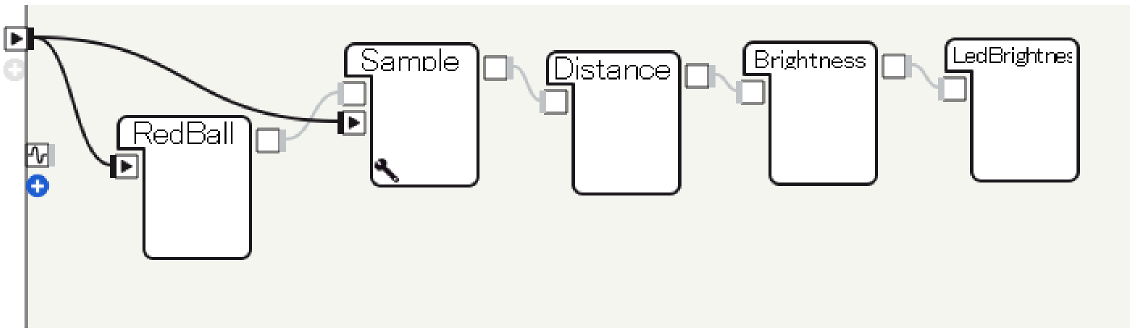
(A) Debug task in RRP (bug present)



(B) Debug task in RRP (bug fixed)

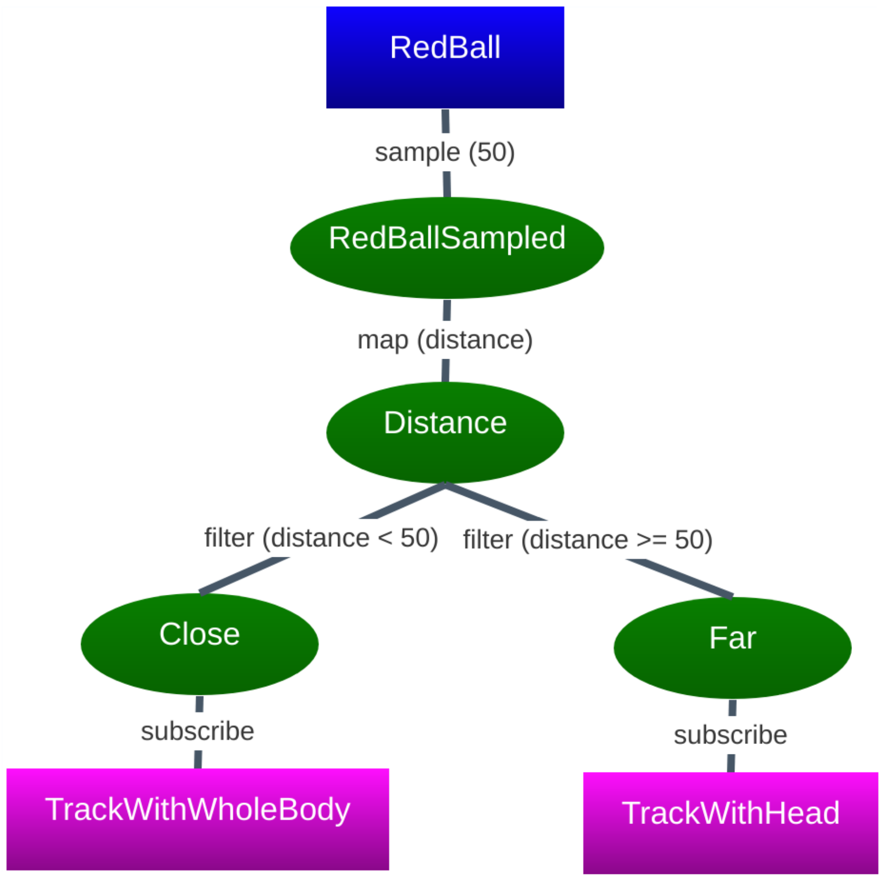


(C) Debug task in Choregraphe (bug present)

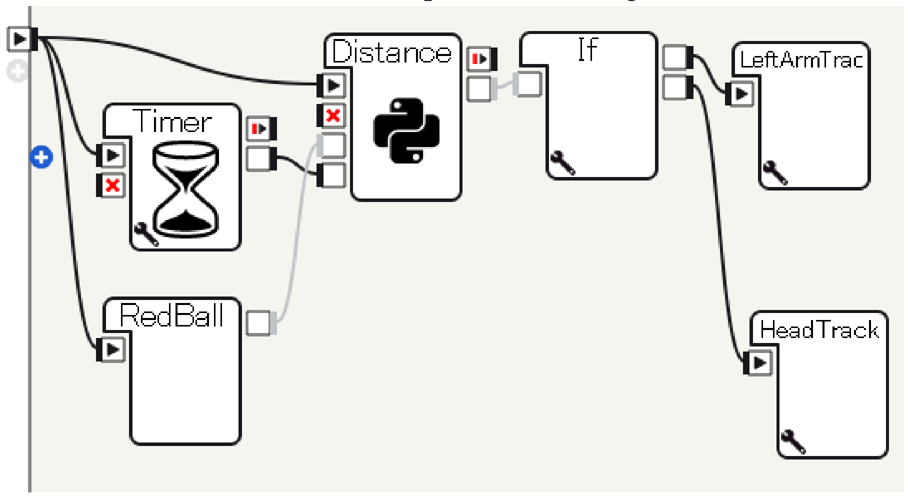


(D) Debug task in Choregraphe (bug fixed)

FIGURE 7.3: RRP and Choregraphe implementations of Debug task



(A) Create task implemented using RRP



(B) Create task implemented using Choregraphe

FIGURE 7.4: RRP and Choregraphe implementations of Create task

Table 7.3: Time taken per task. Mean/SD do not include partipant 9.

| Participant | Task | | | | | |
|---|---|---|---|---|---|---|
| | A | | B | | C | |
| (condition) | CHOR | RRP | CHOR | RRP | CHOR | RRP |
| 1 ($\alpha$) | 00:52 | 01:16 | 00:47 | 00:44 | DNC | 04:31 |
| 2 ($\beta$) | 02:19 | 01:14 | 01:08 | 01:04 | DNC | 02:42 |
| 3 ($\gamma$) | 02:22 | 01:15 | 01:19 | 02:55 | DNC | 05:10 |
| 4 ($\delta$) | 02:20 | 01:32 | 00:47 | 00:38 | DNC | 05:15 |
| 5 ($\delta$) | 00:35 | 00:31 | 02:44 | DNC | DNC | 06:49 |
| 6 ($\gamma$) | 00:57 | 00:28 | 02:00 | 01:51 | DNC | 07:46 |
| 7 ($\beta$) | 00:34 | 00:37 | 00:49 | 00:51 | DNC | 05:25 |
| 8 ($\alpha$) | 00:37 | 00:40 | 02:30 | 05:03 | DNC | 07:10 |
| 9* ($\alpha$) | 05:20 | 01:47 | DNC | 08:17 | DNC | 09:48 |
| Mean | 01:28 | 00:58 | 01:31 | 02:53 | 10:00 | 05:36 |
| SD | 00:49 | 00:26 | 00:48 | 03:14 | 0 | 01:37 |

there was a statistically significant difference between using RRP and Choregraphe. We can say that the creation task was completed faster using RRP than using Choregraphe.

## 7.3.2 Self-evaluation of workload using NASA-TLX

Table 7.4 shows the self-evaluation using NASA-TLX of each participant.

Figures 7.6, 7.7, and 7.8 graph the self evaluation mean and standard deviation per task.

The perceived workload increased as the tasks became harder.

The differences for the explanation task and debugging task are not statistically significant for any subscale. The difference for the creation task is statistically significant. We can say that participants felt a lower workload for the creation task using RRP versus Choregraphe.

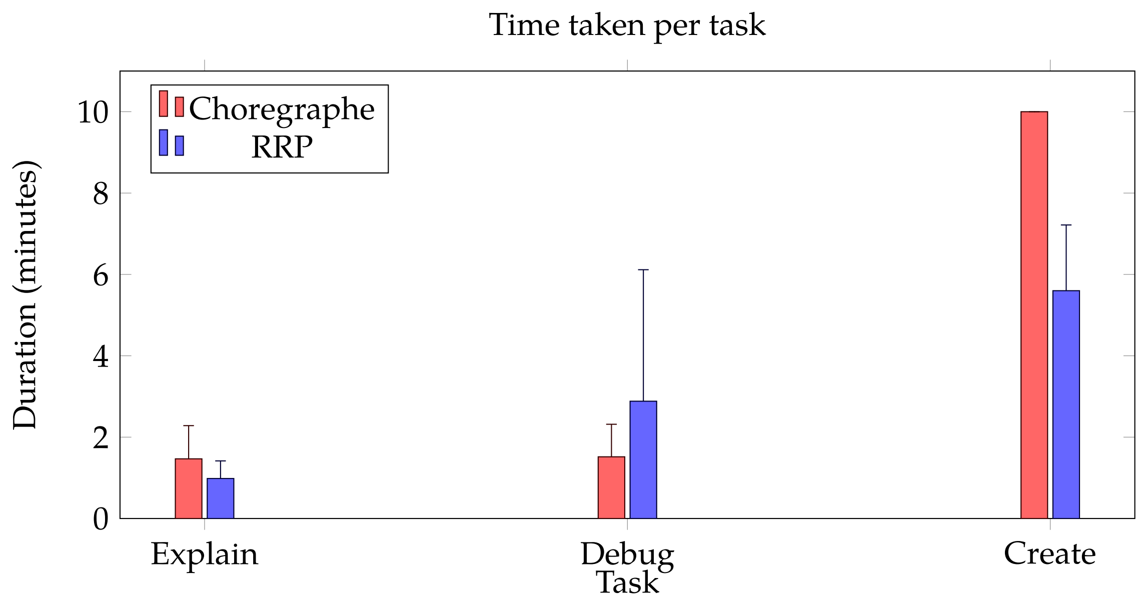


Figure 7.5: Graph of time taken per task

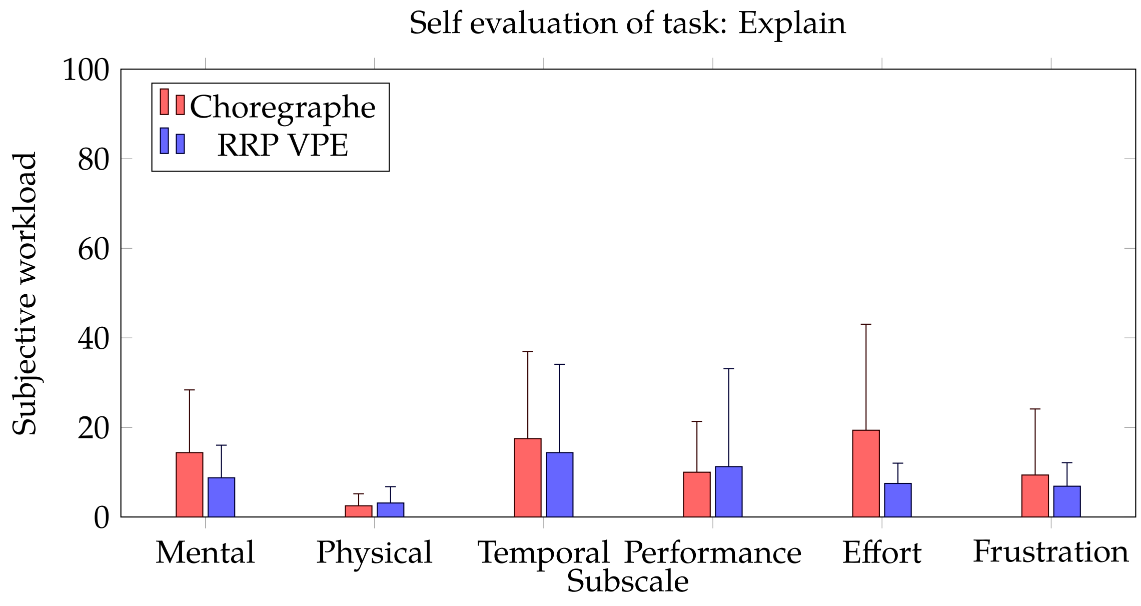


Figure 7.6: Self evaluation of task Explain

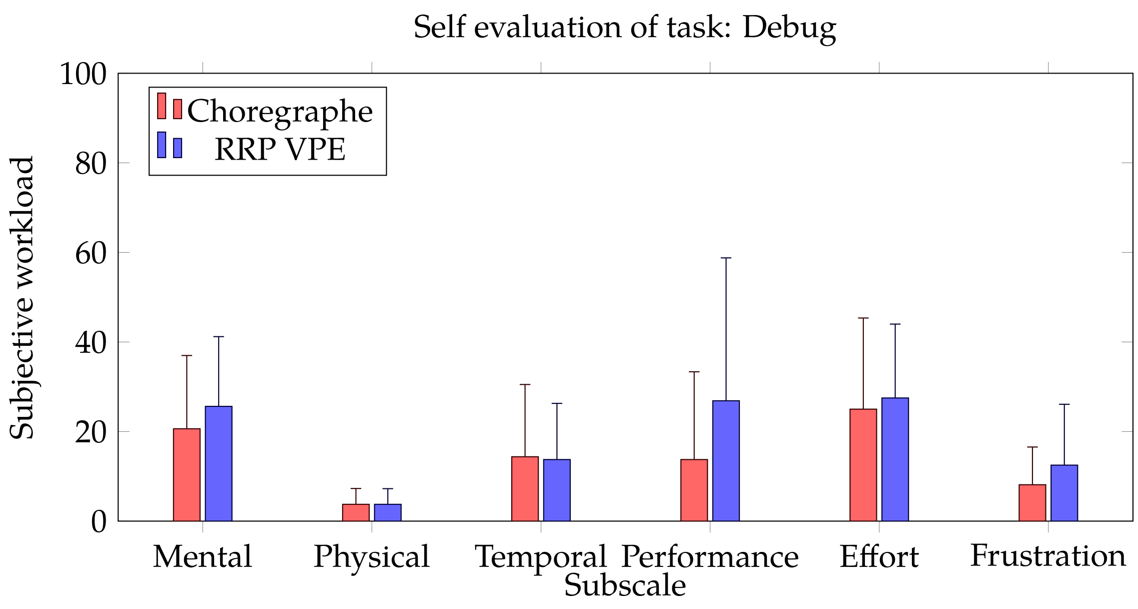


FIGURE 7.7: Self evaluation of task Debug

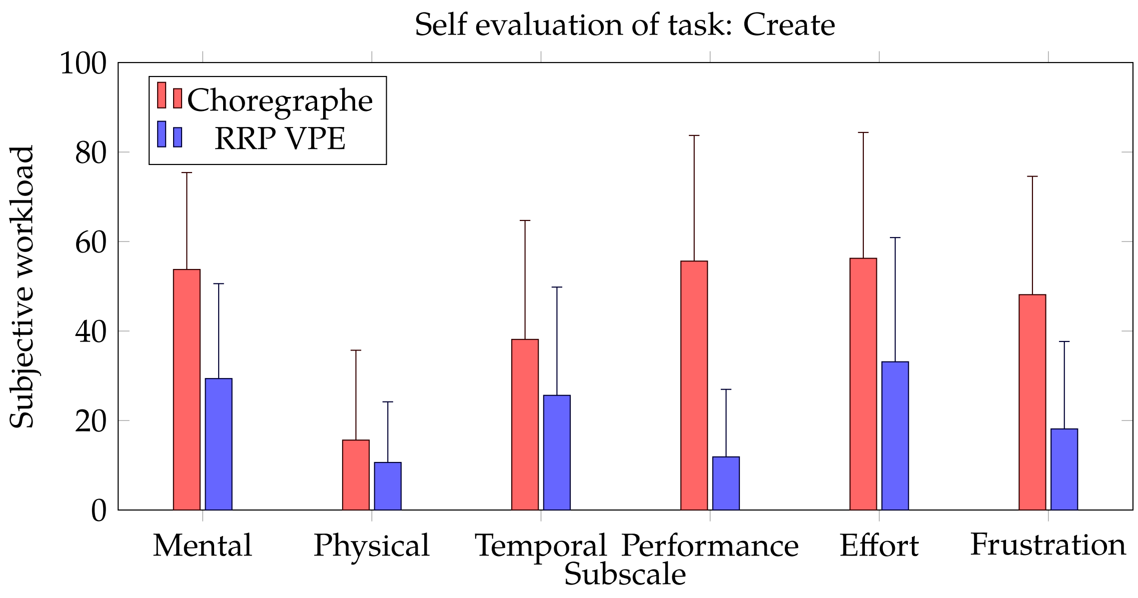


FIGURE 7.8: Self evaluation of task Create

Table 7.4: NASA-TLX self evaluation. Mean/SD do not include participant 9.

| Tool | Task | Subscale | Participant | | | | | | | | | Mean | SD |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9* | | |
| Choregraphe | 1 | Mental demand | 10 | 20 | 15 | 45 | 0 | 15 | 5 | 5 | 0 | 14.375 | 13.999 |
| | | Physical demand | 5 | 0 | 0 | 5 | 0 | 5 | 5 | 0 | 0 | 2.500 | 2.672 |
| | | Temporal demand | 5 | 20 | 50 | 45 | 0 | 5 | 5 | 10 | 25 | 17.500 | 19.456 |
| | | Performance | 5 | 10 | 0 | 25 | 0 | 30 | 5 | 5 | 0 | 10.000 | 11.338 |
| | | Effort | 15 | 15 | 25 | 75 | 0 | 10 | 5 | 10 | 35 | 19.375 | 23.669 |
| | | Frustration | 5 | 5 | 5 | 45 | 0 | 10 | 5 | 0 | 20 | 9.375 | 14.744 |
| | 2 | Mental demand | 5 | 10 | 25 | 50 | 40 | 10 | 10 | 15 | 60 | 20.625 | 16.352 |
| | | Physical demand | 5 | 0 | 5 | 5 | 0 | 10 | 5 | 0 | 0 | 3.750 | 3.535 |
| | | Temporal demand | 5 | 10 | 50 | 25 | 0 | 10 | 5 | 10 | 80 | 14.375 | 16.132 |
| | | Performance | 10 | 5 | 0 | 15 | 0 | 60 | 5 | 15 | 95 | 13.750 | 19.594 |
| | | Effort | 10 | 5 | 20 | 65 | 40 | 35 | 10 | 15 | 80 | 25.000 | 20.354 |
| | | Frustration | 5 | 0 | 25 | 10 | 0 | 5 | 5 | 15 | 90 | 8.125 | 8.425 |
| | 3 | Mental demand | 50 | 50 | 65 | 100 | 45 | 45 | 25 | 50 | 80 | 53.750 | 21.671 |
| | | Physical demand | 25 | 0 | 60 | 15 | 0 | 15 | 10 | 0 | 10 | 15.625 | 20.077 |
| | | Temporal demand | 60 | 50 | 25 | 75 | 0 | 15 | 20 | 60 | 95 | 38.125 | 26.583 |
| | | Performance | 70 | 5 | 75 | 65 | 45 | 85 | 25 | 75 | 10 | 55.625 | 28.086 |
| | | Effort | 70 | 75 | 65 | 75 | 0 | 75 | 25 | 65 | 70 | 56.250 | 28.125 |
| | | Frustration | 75 | 50 | 65 | 75 | 0 | 25 | 35 | 60 | 80 | 48.125 | 26.449 |
| RRP | 1 | Mental demand | 5 | 10 | 5 | 25 | 0 | 10 | 10 | 5 | 0 | 8.750 | 7.440 |
| | | Physical demand | 5 | 0 | 0 | 5 | 0 | 10 | 5 | 0 | 0 | 3.125 | 3.720 |
| | | Temporal demand | 5 | 0 | 60 | 25 | 0 | 10 | 10 | 5 | 0 | 14.375 | 20.077 |
| | | Performance | 5 | 0 | 0 | 10 | 0 | 65 | 5 | 5 | 5 | 11.250 | 21.998 |
| | | Effort | 5 | 5 | 10 | 10 | 0 | 15 | 10 | 5 | 5 | 7.500 | 4.629 |
| | | Frustration | 5 | 0 | 10 | 15 | 0 | 10 | 10 | 5 | 0 | 6.875 | 5.303 |
| | 2 | Mental demand | 10 | 10 | 40 | 40 | 45 | 10 | 10 | 40 | 55 | 25.625 | 16.783 |
| | | Physical demand | 5 | 0 | 5 | 5 | 0 | 10 | 5 | 0 | 0 | 3.750 | 3.535 |
| | | Temporal demand | 5 | 5 | 25 | 15 | 0 | 10 | 10 | 40 | 70 | 13.750 | 13.024 |
| | | Performance | 10 | 5 | 35 | 5 | 100 | 40 | 5 | 15 | 60 | 26.875 | 32.616 |
| | | Effort | 10 | 5 | 50 | 45 | 45 | 25 | 10 | 30 | 70 | 27.500 | 17.928 |
| | | Frustration | 10 | 0 | 40 | 10 | 0 | 10 | 5 | 25 | 50 | 12.500 | 13.627 |
| | 3 | Mental demand | 10 | 15 | 40 | 75 | 20 | 10 | 20 | 45 | 30 | 29.375 | 22.589 |
| | | Physical demand | 5 | 0 | 40 | 20 | 0 | 10 | 10 | 0 | 0 | 10.625 | 13.741 |
| | | Temporal demand | 10 | 10 | 60 | 45 | 0 | 10 | 10 | 60 | 65 | 25.625 | 24.991 |
| | | Performance | 5 | 5 | 5 | 25 | 0 | 45 | 5 | 5 | 45 | 11.875 | 15.338 |
| | | Effort | 10 | 10 | 60 | 85 | 20 | 10 | 15 | 55 | 60 | 33.125 | 29.269 |
| | | Frustration | 10 | 0 | 25 | 50 | 0 | 10 | 5 | 45 | 25 | 18.125 | 19.809 |

# Chapter 8

# Discussion

In this chapter we will discuss the benefits and limitations of the Reactive Robot Programming Paradigm as well as discuss specific characteristics of our developed interpreter and Visual Programming Environment.

## 8.1 Benefits of Reactive Robot Programming

Robots are a natural reactive system because of their embedding into a complex environment and their use of sensors and actuators. The transformational system paradigm is not a natural fit for the intelligent robotics domain because it requires a programmer to use constructs such as threads and loops to deal with concepts such as concurrency and a continuous life time.

We believe RRP is a beneficial robot programming paradigm for four reasons.

First, RRP is concurrent by default as sensors are active in parallel. A programmer hence does not have to manually construct threads or processes to create concurrent behaviour. This allows the programmer to focus on solving problems related to the behaviour that the robot should exhibit, and not dealing with technical characteristics of the programming environment.

Second, the separation between sensing, planning and acting is made explicit. This allows a programmer to look at a behaviour diagram and quickly understand how the robot is supposed to behave when the behaviour is active. Even though this separation between sensing, planning and acting is common in the robotics literature, we have been unable to show a statistically significant difference between our solution and a solution that does not separate between this (Choregraphe).

Third, RRP offers a small set of connectors which, thanks to procedural parameters, have many applications. Other environments for programming robots typically require the programmer to create *blocks* (e.g. nodes in ROS and boxes in Choregraphe). Tools such as Choregraphe offer a library of blocks that the programmer can use, however this library is typically quite restrictive. Increasing the size of the blocks library is not by definition beneficial for the programmer, as this increases the time required for studying about the library. High level connectors with procedural parameters separate what blocks should do and how they should do it. This allows a programmer to learn a small set of connectors and apply these in many situations.

Fourth, RRP offers a visualization of behaviours that can easily be used for learning, communication, programming and debugging. Many solutions for programming robots, for example ROS, are text-based and offer generation of diagrams. These diagrams are only useful for communication and debugging. Choregraphe offers a graphical notation that is useful for programming and debugging, but it is less useful for learning and communication as a lot of information is hidden behind property windows or mouse hovers. TDM offers a graphical notation that can be used for learning and communication and a library for programming in Python, but currently there is no tool that supports visually designing TDM diagrams. Hence the notation is at this point not useful for programming and debugging by novice programmers. RRP Graphs can be used for all four purposes.

## 8.2 Limitations of Reactive Robot Programming

This section will discuss some of the limitations of RRP. Most of the limitations are actually design decisions to keep RRP simple and easy to learn for novice programmers. For RRP to be more widely applicable, even by more experienced programmers for example, some of these limitations should be resolved.

### 8.2.1 Non determinism

RRP Graphs are non-deterministic, meaning that one cannot always predict the outcome of running a graph with only a static view of the graph. This is caused by the execution of the RRP Graph being dependent on previous inputs. Various behaviours can be modelled that are non-deterministic. For example, simply adding a sample-connector between a sensor and an actuator already adds a delay between the robot sensing an event and creating an actuator response. If in that scenario the event occurs multiple times within a single sample period, the robot will respond less often to the event occuring than if the sample-connector had not been added. Another example is when using the combine operator. In this case the behaviour of the RRP Graph is not determined by the most recent event that occured but also by cached values of earlier events. This lead to some interesting bugs, of which we demonstrated one case in the final sample application in Chapter 4.

While these examples are non deterministic when considering the static perspective of the RRP Graph, they are still deterministic when considering the graph from a dynamic perspective (i.e. taking into account the current state of connectors). More difficult cases of non-determinism are caused by race conditions. For example, a query such as "did event A occur before event B" in a combine-connector cannot simply be answered by considering the order of inputs. In RRP, each sensor runs in its own thread. When the sensor receives data, it will start processing that data on its respective thread. When a connector is encountered that has a dedicated thread (such as a sample) the processing will continue on that thread. Each connector adds some processing overhead, however the duration of this processing overhead can vary significantly. This means that an event occuring earlier on one sensor than an event on another sensor can arive later at a connector with multiple inputs. Worse, race conditions can occur on events that arrived on the same sensor. A solution to these problems is using timestamps when timing is important.

Non determinism makes graphs harder to interpret, can introduce bugs and complicates testing. Taking timestamps into account in the processing of events is not an ideal solution. Bugs can occur that are hard to understand, even by experienced programmers. Testing requires a programmer to repeatedly present combinations of inputs to the program. This is however a problem with software testing in general, as the famous Dutch Computer Scientist Edsger Dijkstra commented: "Testing shows the presence, not the absence of bugs" [28].

### 8.2.2 No ports

We made a conscious decision not to add ports to streams and connectors in RRP, to simplify their usage. As was mentioned before, streams either take no input (sensor streams) or they take a single input. At the same time streams have either no output (actuator streams) or they have one or multiple outputs, however the output is always broadcasted on all outputs. Connectors are more versatile, with connectors such as CombineLatest taking a homogeneous set of inputs.

Not having ports does not seem to be a problem for most behaviours, as we have shown in Chapter 4. The biggest problem with not having ports is that it becomes hard to create higher level connectors. For example consider a connector called *FilterCompare*, that takes three inputs: Two values (left side and right side) and an infix operator (e.g. smaller than, smaller than/equals, equals, larger than/equals, larger than). Currently this kind of behaviour requires the programmer to use the *CombineLatest* connector to pair the inputs together, and then the *Filter* connector to do the comparison.

One workaround could be to use the horizontal positioning of inputs to determine how the input should be used by the connector. In our experience however this leads to surprise and confusion by users.

Not having ports allows for simple diagrams to be constructed. To create more powerful applications in the future programmers using RRP would need to have the ability to create new connectors and use ports to disambiguate the inputs.

### 8.2.3 Actuator coordination

A limitation of RRP is that it currently does not have any logic for dealing with actuator coordination. For example, if one behaviour tries to make the robot point towards a ball and another behaviour tries to make the robot point towards a person, a race condition occurs. This is not a topic that we deeply explored in this research, however various existing robot programming systems offer solutions.

In the subsumption architecture [20, 21], behaviours are assigned to a unique layer. Behaviours at lower layers can be interrupted by behaviours at higher layers if both behaviours access a common resource. This however assumes that a total ordering of the behaviours exists. A robot might want to combine different behaviours, such as liveliness and interacting with a person. The total ordering hence forces some static constraints on the robot behaviour.

In Targets-Drives-Means [13–15] behaviours are prioritized using a score calculator. The score calculator can be used to make a robot behave more dynamically in response to its environment. Within a behaviour action units have a defined order. If two action units that use the same resource compete for that resource, TDM will give priority to the action unit in the behaviour with the best score. If both action units are part of the same behaviour then their relative ordering is used to determine which action unit should be allowed to use the resource. It is undefined what happens when action units in two behaviours with the same score try to access a resource. Additionally, organizing the score calculators to correctly determine the priority of behaviours is a complex task as characteristics about the physical world have to be mapped to score calculators, and these score calculators have to be matched.

For example, we can imagine a robot that is programmed to help people who are lost in a shopping mall. The robot can calculate the distance to people in the shopping mall as well as categorize people as needing help or not needing help. The robot now has to combine continuous data (distance to people) with categorical data (whether a person needs help or not). Score calculators however operate only on integers, hence the two pieces of data somehow have to be combined into an integer value.

Another approach for combining conflicting actuator commands is Motor Schemas [6]. Motor Schemas use vector fields for specifying behaviours, hence they are mostly applicable for geometric operations. As was shown in Chapter 2, some behaviours can be created by combining vector fields, e.g. the *move-to-goal-while-avoiding-static-obstacle* behaviour. Other behaviours are harder to combine, such as the *point-at-ball* and *point-at-person* behaviours. Summing both vector fields in this case makes the robot point somewhere in the middle of both targets, which is not a logical behaviour. In this case a partial order to decide the priority of both behaviours is necessary.

Boxes in Choregraphe [80] have start and stop ports. This means that it is up to the programmer to decide when to start and stop behaviours. Besides that boxes can have additional ports which can be used to convey state between boxes. Even though this is a widely applicable method for dealing with actuator coordination, it completely puts the burden on the programmer.

Another option is to broadcast signals which can be used for synchronizing different behaviours. This closely resembles the model of ROS [81], in which many nodes can publish to a topic and many nodes can subscribe to a topic. It is also a mechanism which is part of Harel statecharts [55]. Both ROS and Harel are however solutions aimed at experienced programmers and cannot easily be applied by novices. Besides, both solutions still require a programmer to think about synchronization between each combination of behaviours.

The problem of synchronization between each combination of behaviours is that this requires the programmer to model a maximum of $2^n$ scenarios, where $n$ is the number of actuators. For maximum flexibility, each scenario needs a "score calculator" with $n$ parameters, containing the state of each behaviour. If one behaviour is modified, this could lead to the programmer having to modify all the scenarios. This can quickly get troublesome and would be hard to present to the programmer in a user friendly manner.

## 8.3 Interpreter

We developed an interpreter to run RRP programs on a robot. The current implementation has some limitations which could be improved in the future.

Currently the only supported robot is NAO from Softbank Robotics. That said, there is a layer of abstraction which should make it easy to add robots with similar sensing and acting capabilities as NAO. One platform which should be especially easy to add support for is the robot Pepper, also from Softbank Robotics. The API for Pepper is written in the same language as the API for NAO. Also, the API for both robots is very similar. It would be more complicated to add robots that have an API not written in Python, as a conversion layer would need to be implemented. To be compatible with the API for NAO, we were also pushed to use Python 2.7, which is the latest version of Python supported by the API for NAO.

Currently the only supported data source is the Neo4j Graph Database. Just like in the case of robots, there is an abstraction layer which should make it easy to add new data sources. Of course, support for other graph databases would be easiest to add. Other types such as XML files should also be relatively easy to add support for.

One improvement which we are planning to implement in a next version of the interpreter is separation between the process that communicates with the robot and the process that performs computation of the RRP Graph. These processes can be connected using message queues (e.g. ZeroMQ [58] or Data Distribution Service [35]). If this is implemented, both processes could be developed independently from each other, for example using different Python versions and even different programming languages.

Some of the limitations of non determinism, which we discussed earlier in this chapter, could be solved by implementing new engines in the interpreter. A frame-based engine could for example ensure that the entire graph is executed at a certain frame rate. Within such a frame, all the inputs received on the sensors would be synchronized. This kind of engine would behave very differently from the currently implemented engine.

## 8.4 Visual Programming Environment

Some changes suggested in the discussion section on the Interpreter also require a change in the Visual Programming Environment. To support multiple robots an configuration field has to be added to the VPE to select the robot a behaviour is written for. A similar configuration field has to be added to support multiple engines.

The VPE currently has limited capabilities for working with multiple users in the same behaviour diagram. Actions taken by a user are automatically synchronized to other users, however a race condition can occur when multiple users modify the same connector at the same time. A locking mechanism could be implemented that locks a connector while another user is editing the connector.

Currently there is a one way connection between the robot and the VPE, i.e. the user can start behaviours on the robot using the VPE, but the VPE will not get any feedback from the behaviours. For example, it would be interesting to see in the VPE when certain connectors are being activated. Another example would be to see what information is inputted into a connector and outputted by the connector. If this data is numeric it could also be graphed over time.

The visualization of the RRP Graph could be improved by using a distinctive visual style for different connectors, by for example using graphical icons. Additionally users might want to customize the visualization of an RRP Graph.

These are software engineering problems for which a solution exists, however due to limited time and low research value we decided not to focus on them at this time.

## 8.5 When to use Reactive Robot Programming?

As we have shown, there are various benefits and limitations to Reactive Robot Programming, the interpreter and the visual programming environment. The famous Computer Scientist Fred Brooks once stated that there is no silver bullet in software engineering, i.e. that "there is no single development, in either technology of management technique, which by itself promises even one order of magnitude improvement within a decade in productivity, in reliability, in simplicity" [19]. The same applies to Reactive Robot Programming and Reactive Programming in general.

One probably wonders however if there is an easy set of criteria to decide whether to use Reactive Robot Programming or more traditional approaches (e.g. imperative programming of robot software). One simple criterium is the complexity of the events occuring in the domain and their mapping with actions by the system under development.

If there is a one-to-one mapping from events to actions and the action to be performed does not depend on any details about the event besides that the event occured, then probably an imperative programming approach is adequate. It is easy in this case to set up an event loop and call imperative code based on the events that occur.

However if there is a more complicated mapping from events to actions, for example if multiple events in concert determine an action, or the event contains some data which influences the action to be taken, then the Reactive Robot Programming approach is beneficial. Reactive Robot Programming offers a set of connectors that can be used to process the events, e.g. by extracting/transforming/filtering/combining data. Interactive robots are a technology in which problems frequently exhibit this more complicated mapping, and hence, except for the most basic applications, reactive robot programming is a beneficial approach.

# Chapter 9

# Conclusion

In this thesis we presented a new paradigm for the development and operations of software for robots, called Reactive Robot Programming. This approach is especially useful for end-users to program a robot that needs to interact with the outside world.

At the start of this thesis we states that our main research objective is to create a *reactive robot programming* environment so that *end-users* can develop and operate applications for *Socially Assistive Robotics (SAR)* experiments with minimal support from a robot programming expert. To this end we studied what type of applications therapists and researchers in the field of Human-Robot Interaction were currently creating. We also studied which approaches programmers of robots currently use to develop and operate software for robots. We then used this knowledge to develop a robot programming environment which allows programmers to create reactive applications for their robot. We tested the effectiveness of our developed reactive robot programming environment by studying the performance of both experienced and inexperienced programmers using our tool and a state of the art tool for novice robot programmers (Choregraphe). Finally we considered when it is beneficial to use our reactive robot programming environment instead of more traditional ways of programming robots.

This research has resulted in various contributions to academia and industry. We believe that the knowledge produced regarding the application of Reactive Programming to Robotics can be used by other researchers to improve robot software platforms in the future. All of the software that we developed as part of this research has been released as open source and can be used by anyone who is interested in using the software or to further improve it.

We hope that end-users and experienced programmers alike can benefit from our work.

# Bibliography

[1] Hal Abelson, Nat Goodman, and Lee Rudolph. *LOGO Manual*. A.I. Memo 313. MIT Artificial Intelligence Laboratory. Dec. 1974. URL: http://hdl.handle.net/1721.1/6226.

[2] Minoo Alemi et al. "Impact of a Social Humanoid Robot as a Therapy Assistant in Children Cancer Treatment". In: *Social Robotics*. Ed. by M. Beetz, B. Johnston, and M. A. Williams. Vol. 8755. Lecture Notes in Artificial Intelligence. 2014, pp. 11–22. ISBN: 978-3-319-11973-1. DOI: 10.1007/978-3-319-11973-1_2.

[3] Sonya Alexandrova, Zachary Tatlock, and Maya Cakmak. "RoboFlow: A flow-based visual programming language for mobile manipulation tasks". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. May 2015, pp. 5537–5544. DOI: 10.1109/ICRA.2015.7139973.

[4] Diego Alonso et al. "V3CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development". In: *Journal of Software Engineering for Robotics* 1.1 (2010), pp. 3–17. URL: https://joser.unibg.it/index.php?journal=joser&page=article&op=view&path%5B%5D=18.

[5] Noriaki Ando et al. "RT-Middleware: Distributed Component Middleware for RT (Robot Technology)". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2005. DOI: 10.1109/IROS.2005.1545521.

[6] Michael A. Arbib. "Schema Theory". In: *Encyclopedia of Artificial Intelligence*. Wiley-Interscience, 1992. ISBN: 0471503053.

[7] Ronald C. Arkin. *Behavior-based Robotics*. 1st. Intelligent Robots and Autonomous Agents. Cambridge, MA, USA: The MIT Press, 1998. ISBN: 978-0-262-01165-5. URL: https://mitpress.mit.edu/books/behavior-based-robotics.

[8] Ritta Baddoura and Gentiane Venture. "Human motion characteristics in relation to feeling familiar or frightened during an announced short interaction with a proactive humanoid". In: *Frontiers in Neurorobotics* 8 (2014). ISSN: 1662-5218. DOI: 10.3389/fnbot.2014.00012.

[9] Ritta Baddoura and Gentiane Venture. "This Robot is Sociable: Close-up on the Gestures and Measured Motion of a Human Responding to a Proactive Robot". In: *International Journal of Social Robotics* 7.4 (2015), pp. 489–496. ISSN: 1875-4791. DOI: 10.1007/s12369-015-0279-x.

[10] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Cambridge University Press, 2013. ISBN: 978-0521766142. DOI: 10.1017/CBO9781139032636.

[11] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015. ISBN: 978-0-13-404984-7.

[12] Esubalew T. Bekele et al. "A Step Towards Developing Adaptive Robot-Mediated Intervention Architecture (ARIA) for Children With Autism". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 21.2 (2013), pp. 289–299. ISSN: 1534-4320. DOI: `10.1109/tnsre.2012.2230188`.

[13] Vincent Berenz and Kenji Suzuki. "Usability Benchmarks of the Targets-Drives-Means Robotic Architecture". In: *Proceedings of the 12th IEEE-RAS International Conference on Humanoid Robots*. 2012, pp. 514–519. DOI: `10.1109/HUMANOIDS.2012.6651568`.

[14] Vincent Berenz and Kenji Suzuki. "Targets-Drives-Means: A declarative approach to dynamic behavior specification with higher usability". In: *Robotics and Autonomous Systems* 62.4 (2014), pp. 545–555. DOI: `10.1016/j.robot.2013.12.010`.

[15] Vincent Berenz et al. "TDM: A Software Framework for Elegant and Rapid Development of Autonomous Behaviors for Humanoid Robots". In: *Proceedings of the 11th IEEE-RAS International Conference on Humanoid Robots*. 2011, pp. 179–186. DOI: `10.1109/Humanoids.2011.6100887`.

[16] Stephen Blackheath and Anthony Jones. *Functional Reactive Programming*. Manning Publications, 2016. ISBN: 9781633430105. URL: `https://www.manning.com/books/functional-reactive-programming`.

[17] Xavier Blanc, Jérôme Delatour, and Tewfik Ziadi. "Benefits of the MDE approach for the development of embedded and robotic systems. Application to Aibo." In: *Proc. of the 3rd National Conference on Control Architectures of Robots*. 2007. URL: `http://www.lirmm.fr/gtcar/webcar/CAR2007/papers/blanc-delatour.pdf`.

[18] Valentino Braitenberg. *Vehicles: Experiments in synthetic psychology*. MIT Press, 1986. URL: `https://mitpress.mit.edu/books/vehicles`.

[19] Fred P. J. Brooks. "No Silver Bullet Essence and Accidents of Software Engineering". In: *Computer* 20.4 (Apr. 1987), pp. 10–19. ISSN: 0018-9162. DOI: `10.1109/MC.1987.1663532`.

[20] Rodney A. Brooks. "Elephants don't play chess." In: *Robotics and Autonomous Systems* 6.1-2 (1990), pp. 3–15. DOI: `10.1016/S0921-8890(05)80025-9`.

[21] Rodney A. Brooks. "Intelligence without representation". In: *Artificial Intelligence* 47.1–3 (1991), pp. 139–159. DOI: `10.1016/0004-3702(91)90053-M`.

[22] Davide Brugali. "Model-Driven Software Engineering in Robotics: Models Are Designed to Use the Relevant Things, Thereby Reducing the Complexity and Cost in the Field of Robotics". In: *IEEE Robotics Automation Magazine* 22.3 (Sept. 2015), pp. 155–166. ISSN: 1070-9932. DOI: `10.1109/MRA.2015.2452201`.

[23] Davide Brugali and Patrizia Scandurra. "Component-based robotic engineering (Part I) [Tutorial]". In: *IEEE Robotics Automation Magazine* 16.4 (Dec. 2009), pp. 84–96. ISSN: 1070-9932. DOI: `10.1109/MRA.2009.934837`.

[24] Davide Brugali and Azamat Shakhimardanov. "Component-Based Robotic Engineering (Part II)". In: *IEEE Robotics Automation Magazine* 17.1 (Mar. 2010), pp. 100–112. ISSN: 1070-9932. DOI: 10.1109/MRA.2010.935798.

[25] Herman Bruyninckx. "Open robot control software: the OROCOS project". In: *Proceedings 2001 IEEE International Conference on Robotics and Automation (ICRA 2001)*. Vol. 3. IEEE. 2001, pp. 2523–2528. DOI: 10.1109/ROBOT.2001.933002.

[26] Herman Bruyninckx et al. "The BRICS Component Model: A Model-based Development Paradigm for Complex Robotics Software Systems". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC '13. Coimbra, Portugal: ACM, 2013, pp. 1758–1764. ISBN: 978-1-4503-1656-9. DOI: 10.1145/2480362.2480693.

[27] *Business Process Model and Notation Specification*. Accessed on December 25th, 2017. Object Management Group, Jan. 2011. URL: http://www.omg.org/spec/BPMN/2.0/.

[28] John N. Buxton and Brian Randell, eds. *Software Engineering Techniques. Report on a conference sponsored by the NATO Science Committee*. Rome, Italy, 27th to 31th October 1969; http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF. NATO, Science Committee, Apr. 1970. URL: https://dl.acm.org/citation.cfm?id=1102021.

[29] Rafael Capilla et al. "An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry". In: *Journal of Systems and Software* 91 (2014), pp. 3–23. DOI: 10.1016/j.jss.2013.12.038.

[30] Alonso Church. "A set of postulates for the foundation of logic". In: *Annals of Mathematics* 33.2 (1932), pp. 346–366. DOI: 10.2307/1968337.

[31] Sandra Costa et al. "Using a Humanoid Robot to Elicit Body Awareness and Appropriate Physical Interaction in Children with Autism". In: *International Journal of Social Robotics* 7.2 (2015), pp. 265–278. ISSN: 1875-4791. DOI: 10.1007/s12369-014-0250-2.

[32] Antony Courtney and Conal Elliott. "Genuinely Functional User Interfaces". In: *Proceedings of the 2001 Haskell Workshop*. 2001. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.136.1413.

[33] Evan Czaplicki. "Elm: Concurrent FRP for Functional GUIs". MA thesis. Harvard University, 2012.

[34] Wanda P. Dann, Stephen Cooper, and Randy Pausch. *Learning to Program with Alice*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2011. ISBN: 9780132122474. URL: http://www.aliceprogramming.net/.

[35] *Data Distribution Service*. Accessed on January 2nd, 2018. Object Management Group, Mar. 2015. URL: http://www.omg.org/spec/DDS/1.4.

[36] Saadia Dhouib et al. "RobotML, a Domain-specific Language to Design, Simulate and Deploy Robotic Applications". In: *Proceedings of the Third International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. SIMPAR'12. Tsukuba, Japan: Springer-Verlag, 2012, pp. 149–160. ISBN: 978-3-642-34326-1. DOI: 10.1007/978-3-642-34327-8_16.

[37] Emil Eifrem, Jim Webber, and Ian Robinson. *Graph Databases*. O'Reilly, 2015. ISBN: 978-1449356262.

[38] Conal Elliott. *A Brief Introduction to ActiveVRML*. Tech. rep. MSR-TR-96-05. Microsoft Research, 1996. URL: `http://conal.net/papers/ActiveVRML/`.

[39] Conal Elliott and Paul Hudak. "Functional reactive animation". In: *Proceedings of the International Conference on Functional Programming*. 1997. URL: `http://conal.net/papers/icfp97/`.

[40] Floris Erich. "End-user Software Engineering of Cognitive Robot Applications Using Procedural Parameters and Complex Event Processing". In: *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. SPLASH Companion 2016. Amsterdam, Netherlands: ACM, 2016, pp. 47–48. ISBN: 978-1-4503-4437-1. DOI: `10.1145/2984043.2998538`.

[41] Floris Erich, Chintan Amrit, and Maya Daneva. "A Mapping Study on Cooperation between Information System Development and Operations". In: *Product-Focused Software Process Improvement*. Ed. by Andreas Jedlitschka et al. Vol. 8892. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 277–280. ISBN: 978-3-319-13834-3. DOI: `10.1007/978-3-319-13835-0_21`.

[42] Floris Erich, Chintan Amrit, and Maya Daneva. "Cooperation Between Information System Development and Operations: A Literature Review". In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '14. Torino, Italy: ACM, 2014. DOI: `10.1145/2652524.2652598`.

[43] Floris Erich, Chintan Amrit, and Maya Daneva. "A qualitative study of DevOps usage in practice". In: *Journal of Software: Evolution and Process* 29.6 (2017). DOI: `10.1002/smr.1885`.

[44] Floris Erich, Masakazu Hirokawa, and Kenji Suzuki. "A Visual Environment for Reactive Robot Programming of Macro-level Behaviors". In: *Social Robotics*. 2017. DOI: `10.1007/978-3-319-70022-9_57`.

[45] Paul Fitzpatrick et al. "A middle way for robotics middleware". In: *Journal of Software Engineering for Robotics* 5.2 (2014), pp. 42–49. URL: `https://joser.unibg.it/index.php?journal=joser&page=article&op=view&path%5B%5D=69`.

[46] N. Fraser. "Ten things we've learned from Blockly". In: *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. Oct. 2015, pp. 49–50. DOI: `10.1109/BLOCKS.2015.7369000`.

[47] Marina Fridin. "Kindergarten social assistive robot: First meeting and ethical issues". In: *Computers in Human Behavior* 30 (2014), pp. 262–272. ISSN: 0747-5632. DOI: `10.1016/j.chb.2013.09.005`.

[48] Marina Fridin and Mark Belokopytov. "Acceptance of socially assistive humanoid robot by preschool and elementary school teachers". In: *Computers in Human Behavior* 33 (2014), pp. 23–31. ISSN: 0747-5632. DOI: `10.1016/j.chb.2013.12.016`.

[49] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Nov. 1994. ISBN: 0201633612.

[50] Luca Gherardi. "Variability Modeling and Resolution in Component-based Robotics Systems". PhD thesis. University of Bergamo, 2013.

[51] Luca Gherardi and Davide Brugali. "Modeling and reusing robotic software architectures: The HyperFlex toolchain". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. May 2014, pp. 6414–6420. DOI: 10.1109/ICRA.2014.6907806.

[52] Luca Gherardi and Nico Hochgeschwender. "RRA: Models and tools for robotics run-time adaptation". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2015, pp. 1777–1784. DOI: 10.1109/IROS.2015.7353608.

[53] Barney G. Glaser and Anselm L. Strauss. *The discovery of grounded theory: strategies for qualitative research*. AldineTransaction, 1967. ISBN: 0-202-30260-1.

[54] Thomas R. G. Green and Marian Petre. "Usability analysis of visual programming environments: a 'cognitive dimensions' framework". In: *Journal of Visual Languages & Computing* 7.2 (1996), pp. 131–174. DOI: 10.1006/jvlc.1996.0009.

[55] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of computer programming* 8.3 (1987), pp. 231–274. DOI: 10.1016/0167-6423(87)90035-9.

[56] Sandra G. Hart. "Nasa-Task Load Index (NASA-TLX); 20 Years Later". In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 50.9 (2006), pp. 904–908. DOI: 10.1177/154193120605000909. eprint: https://doi.org/10.1177/154193120605000909.

[57] Brian Harvey and Jens Mönig. "Lambda in blocks languages: Lessons learned". In: *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. Oct. 2015, pp. 35–38. DOI: 10.1109/BLOCKS.2015.7368997.

[58] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. Ed. by Andy Oram and Maria Gulick. O'Reilly, Mar. 2013. ISBN: 978-1449334062.

[59] Paul Hudak et al. "Advanced Functional Programming: 4th International School". In: *Advanced Functional Programming: 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002. Revised Lectures*. Ed. by Johan Jeuring and Simon L. Peyton Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. Chap. Arrows, Robots, and Functional Reactive Programming, pp. 159–187. ISBN: 978-3-540-44833-4. DOI: 10.1007/978-3-540-44833-4_6.

[60] Sven Jörges et al. "Model Driven Design of Reliable Robot Control Programs Using the jABC". In: *Engineering of Autonomic and Autonomous Systems, 2007. EASe '07. Fourth IEEE International Workshop on*. Mar. 2007, pp. 137–148. DOI: 10.1109/EASE.2007.17.

[61] Alan Kay. *Squak etoys, children and learning*. Tech. rep. Viewpoint Research Institute, 2005.

[62] Barbera Kitchenham. “Guidelines for performing systematic literature reviews in software engineering”. In: *Technical report, Ver. 2.3 EBSE Technical Report. EBSE*. sn, 2007.

[63] Hatice Kose et al. “The Effect of Embodiment in Sign Language Tutoring with Assistive Humanoid Robots”. In: *International Journal of Social Robotics* 7.4 (2015), pp. 537–548. ISSN: 1875-4791. DOI: 10.1007/s12369-015-0311-1.

[64] Matthew Lasar. *25 years of HyperCard — the missing link to the Web*. https://arstechnica.com/gadgets/2012/05/25-years-of-hypercard-the-missing-link-to-the-web/. Accessed on December 25th, 2017. May 2012.

[65] Alex Lotz et al. “Managing Run-Time Variability in Robotics Software by Modeling Functional and Non-functional Behavior”. In: *Enterprise, Business-Process and Information Systems Modeling - 14th International Conference, BPMDS 2013, 18th International Conference, EMMSAD 2013, Held at CAiSE 2013, Valencia, Spain, June 17-18, 2013. Proceedings*. 2013, pp. 441–455. DOI: 10.1007/978-3-642-38484-4_31.

[66] Scott MacKenzie. *Human-Computer Interaction: An Empirical Research Perspective*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 9780124058651.

[67] Gergely Magyar and Maria Vircikova. “Socially-Assistive Emotional Robot that Learns from the Wizard During the Interaction for Preventing Low Back Pain in Children”. In: *Social Robotics*. Ed. by A. Tapus et al. Vol. 9388. Lecture Notes in Artificial Intelligence. 2015, pp. 411–420. ISBN: 978-3-319-25554-5. DOI: 10.1007/978-3-319-25554-5_41.

[68] John McCarthy. “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”. In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782. DOI: 10.1145/367177.367199.

[69] George H. Mealy. “A method for synthesizing sequential circuits”. In: *Bell System Technical Journal, The* 34.5 (Sept. 1955), pp. 1045–1079. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1955.tb03788.x.

[70] Erik Meijer. “Your Mouse is a Database”. In: *Communications of the ACM* 55.5 (2012), pp. 66–73. DOI: 10.1145/2168796.2169076.

[71] Erik Meijer. “The Curse of the Excluded Middle”. In: *Communications of the ACM* 57.6 (2014), pp. 50–55. DOI: 10.1145/2611429.2611829.

[72] Marvin Minsky. *The Society of Mind*. Simon and Schuster, 1986. ISBN: 9780671657130.

[73] Edward F. Moore. “Gedanken-Experiments on Sequential Machines”. In: *Automata Studies*. Ed. by C.E. Shannon and J. MacCarthy. Princeton University Press, 1956, pp. 129–153. DOI: 10.2307/2964500.

[74] Abid Mujtaba. *Lego Mindstorms EV3 Essentials*. Packt Publishing, 2014. ISBN: 9781783985029.

[75] Yadong Pan et al. “On the Reaction to Robot’s Speech in a Hotel Public Space”. In: *International Journal of Social Robotics* 7.5 (2015), pp. 911–920. ISSN: 1875-4791. DOI: 10.1007/s12369-015-0320-0.

[76] John Peterson, Gregory D. Hager, and Paul Hudak. "Proceedings - IEEE International Conference on Robotics and Automation". In: IEEE, 1999. Chap. Language for declarative robotic programming, pp. 1144–1151.

[77] John Peterson, Paul Hudak, and Conal Elliott. "Lambda in Motion: Controlling Robots with Haskell". In: *Practical Aspects of Declarative Languages: First International Workshop, PADL'99 San Antonio, Texas, USA, January 18–19, 1999 Proceedings*. Ed. by Gopal Gupta. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 91–105. ISBN: 978-3-540-49201-6. DOI: 10.1007/3-540-49201-1_7.

[78] John Peterson et al. "FVision: A Declarative Language for Visual Tracking". In: *Practical Aspects of Declarative Languages, Third International Symposium (PADL 2001)*. Ed. by I. V. Ramakrishnan. Vol. 1990. Lecture Notes in Computer Science. Las Vegas, Nevada, USA: Springer, Mar. 2001, pp. 304–321. DOI: 10.1007/3-540-45241-9_21.

[79] Klaus Pohl, Günther Böckle, and Frank van der Linde. *Software Product Line Engineering*. Springer-Verlag, 2004. DOI: 10.1007/3-540-28901-1.

[80] E. Pot et al. "Choregraphe: A Graphical Tool for Humanoid Robot Programming". In: *IEEE International Symposium on Robot and Human Interactive Communication*. 2009. DOI: 10.1109/ROMAN.2009.5326209.

[81] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. 2009. URL: http://www.willowgarage.com/papers/ros-open-source-robot-operating-system.

[82] Arunkumar Ramaswamy, Bruno Monsuez, and Adriana Tapus. "Architecture modeling and analysis language for designing robotic architectures". In: *2014 13th International Conference on Control Automation Robotics Vision (ICARCV)*. Dec. 2014, pp. 1911–1916. DOI: 10.1109/ICARCV.2014.7064608.

[83] Arunkumar Ramaswamy, Bruno Monsuez, and Adriana Tapus. "Model-driven Software Development Approaches in Robotics Research". In: *Proceedings of the 6th International Workshop on Modeling in Software Engineering*. MiSE 2014. Hyderabad, India: ACM, 2014, pp. 43–48. ISBN: 978-1-4503-2849-4. DOI: 10.1145/2593770.2593781.

[84] Alex Repenning. "Agentsheets: A Tool for Building Domain-oriented Visual Programming Environments". In: *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*. CHI '93. Amsterdam, The Netherlands: ACM, 1993, pp. 142–143. ISBN: 0-89791-575-5. DOI: 10.1145/169059.169119.

[85] Mitchel Resnick et al. "Scratch: programming for all". In: *Communications of the ACM* 52.11 (2009), pp. 60–67. DOI: 10.1145/1592761.1592779.

[86] Ben Robins and Kerstin Dautenhahn. "Developing Play Scenarios for Tactile Interaction with a Humanoid Robot: A Case Study Exploration with Children with Autism". In: *Social Robotics, Icsr 2010*. Ed. by S. S. Ge et al. Vol. 6414. Lecture Notes in Artificial Intelligence. 2010, pp. 243–252. ISBN: 978-3-642-17247-2. DOI: 10.1007/978-3-642-17248-9_25.

[87] Ben Robins, Kerstin Dautenhahn, and Janek Dubowski. "Does appearance matter in the interaction of children with autism with a humanoid robot?" In: *Interaction Studies* 7.3 (2006), pp. 479–512. ISSN: 1572-0373. DOI: 10.1075/is.7.3.16rob.

[88] Christian Schlegel et al. "Design Abstraction and Processes in Robotics: From Code-Driven to Model-Driven Engineering". In: *Simulation, Modeling, and Programming for Autonomous Robots: Second International Conference, SIMPAR 2010, Darmstadt, Germany, November 15-18, 2010. Proceedings*. Ed. by Noriaki Ando et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 324–335. ISBN: 978-3-642-17319-6. DOI: 10.1007/978-3-642-17319-6_31.

[89] Christian Schlegel et al. "Model-driven software systems engineering in robotics: Covering the complete life-cycle of a robot". In: *Information Technology* 57.2 (2015), pp. 85–98. DOI: 10.1515/itit-2014-1069.

[90] Aaron Steinfeld, Odest Chadwicke Jenkins, and Brian Scassellati. "The Oz of Wizard: Simulating the Human for Interaction Research". In: *Proceedings of the 4th ACM/IEEE International Conference on Human Robot Interaction*. Ed. by Matthias Schultz et al. 2009. ISBN: 978-1-60558-404-1. DOI: 10.1145/1514095.1514115.

[91] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2008. ISBN: 978-0470167748.

[92] Elena Torta et al. "Evaluation of a Small Socially-Assistive Humanoid Robot in Intelligent Homes for the Care of the Elderly". In: *Journal of Intelligent & Robotic Systems* 76.1 (2014), pp. 57–71. ISSN: 0921-0296. DOI: 10.1007/s10846-013-0019-0.

[93] Alan M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. ISSN: 1460-244X. DOI: 10.1112/plms/s2-42.1.230.

[94] Pinar Uluer, Neziha Akalin, and Hatice Koese. "A New Robotic Platform for Sign Language Tutoring Humanoid Robots as Assistive Game Companions for Teaching Sign Language". In: *International Journal of Social Robotics* 7.5 (2015), pp. 571–585. ISSN: 1875-4791. DOI: 10.1007/s12369-015-0307-x.

[95] Eric Wade, Avinash. R. Parnandi, and Maja J. Matarić. "Using socially assistive robotics to augment motor task performance in individuals post-stroke". In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sept. 2011, pp. 2403–2408. DOI: 10.1109/IROS.2011.6095107.

[96] Norbert Wiener. *Cybernetics or Control and Communication in the Animal and the Machine*. Vol. 25. MIT press, 1961. ISBN: 9780262230070.

[97] Roel J. Wieringa. *Design Methods for Reactive Systems*. Morgan Kaufmann, 2003. ISBN: 9781558607552.

[98] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014. ISBN: 978-3-662-43838-1. DOI: 10.1007/978-3-662-43839-8.

[99] David Wolber. "App Inventor and Real-world Motivation". In: *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*. SIGCSE '11. Dallas, TX, USA: ACM, 2011, pp. 601–606. ISBN: 978-1-4503-0500-6. DOI: 10.1145/1953163.1953329.

[100] Mingqi Zhao et al. "A Humanoid Robot Used as an Assistive Intervention Tool for Children with Autism Spectrum Disorder: A Preliminary Research". In: *Brain and Health Informatics: International Conference, BHI 2013, Maebashi, Japan, October 29-31, 2013. Proceedings*. Ed. by Kazayuki Imamura et al. Cham: Springer International Publishing, 2013, pp. 336–347. ISBN: 978-3-319-02753-1. DOI: 10.1007/978-3-319-02753-1_34.