UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

MAPPING STATE MACHINES TO DEVELOPERS' MENTAL MODEL: FAST UNDERSTANDING OF ROBOTIC BEHAVIORS IN THE REAL WORLD

TESIS PARA OPTAR AL GRADO DE
DOCTOR EN CIENCIAS, MENCIÓN COMPUTACIÓN

MIGUEL ENRIQUE CAMPUSANO ARAYA

PROFESOR GUÍA:
ALEXANDRE BERGEL
JOHAN FABRY

MIEMBROS DE LA COMISIÓN:
LUIS MATEU BRULE
SERGIO OCHOA DELORENZI
SEBASTIAN WREDE

SANTIAGO, CHILE
2020

Resumen

Típicamente, el desarrollo del comportamiento de un robot conlleva una serie de pasos: escribir el código fuente, cargar el código en un simulador o un robot, y ejecutar el código en el robot. Si el robot actúa de forma errónea, el programador debe mentalmente trazar el error, desde el comportamiento hacia el programa que lo produjo, para poder arreglarlo. Todo esto resulta en una dilatada distancia cognitiva entre el programador y el comportamiento del robot, ralentizando el desarrollo e implicando que las pruebas con diferentes comportamientos sean prohibitivamente costosas. Esto es aún más cierto cuando los robots se prueban en el mundo real, interactuando con objetos reales.

Para mejorar el desarrollo de la robótica basada en comportamientos, los desarrolladores deben recibir retroalimentación significativa de la ejecución del programa. Con esto, los desarrolladores pueden formular hipótesis precisas sobre lo que el robot está haciendo, permitiéndoles reparar comportamientos defectuosos más rápidamente. Si los desarrolladores, además, pueden cambiar y modificar sus programas mientras los ejecutan, también podrán escribir y probar programas más rápidamente, removiendo así la mencionada distancia cognitiva. La retroalimentación significativa con esta conexión inmediata es conocida como *Programación en Vivo*. La programación en vivo permite la creación y modificación extremadamente rápida de comportamientos robóticos, aumentando drásticamente la velocidad de depuración.

En esta tesis estudiamos retroalimentación significativa y programación en vivo, y lo aplicamos al desarrollo de robótica basada en comportamientos. Para lo primero, presentamos VizRob, una herramienta con visualizaciones interactivas de *logs* en máquinas de estado: que son la representación del comportamiento. Llevamos a cabo un pequeño caso de estudio que muestra que: (i) VizRob ayuda a resolver escenarios de depuración intrincados y (ii) VizRob llena un vacío importante que dejan las herramientas de desarrollo de comportamientos robóticos.

Para programación en vivo, presentamos LRP, un lenguaje de programación en vivo para comportamientos robóticos. Detallamos la importancia de LRP y presentamos unos experimentos para medir el impacto de LRP en la robótica basada en comportamientos. Descubrimos que LRP **no** supera significativamente un lenguaje clásico (*i.e.,* no en vivo), ni en compresión ni en escritura de programas. No obstante, los comentarios de los sujetos de prueba indican preferencia por LRP. Estos resultados parecen contradecir estudios de programación en vivo en otras áreas. Acá aprendimos que la compleja API escogida tiene una fuerte influencia negativa en los resultados. Hasta donde sabemos, este es el primer experimento en profundidad de programación en vivo en un dominio complejo.

Abstract

Typically, development of robot behavior entails writing the code, deploying it in a simulator or robot and running it in a test setting. In the case of a faulty behavior, the programmer needs to mentally map the error of the behavior back to the source code that caused it before being able to fix the error. This process results in a long cognitive distance between the programmer and the resulting behavior, which slows down development and can make experimentation with different behaviors prohibitively expensive. This is especially true when these robots are tested in the real world, interacting with real elements.

To improve the development of behavior-based robotics, developers should receive meaningful feedback of the program execution. With meaningful feedback, developers have the potential to formulate accurate hypotheses of how the robot is executing, allowing them to fix wrong behaviors faster. Even more, if developers have the ability to change and modify their programs on-the-fly, they also gain the ability to write and test programs even faster, potentially removing the long cognitive distance altogether. Meaningful feedback with immediate connection is known as Live Programming. Live programming allows for extremely rapid creation and variation of robot behaviors and for dramatically increased debugging speed.

In this dissertation we study both meaningful feedback and live programming and apply them to the development of behavior-based robotics. By meaningful feedback we first introduce VizRob, a tool with interactive visualizations built from log traces into a state machine model, that is, the visual representation of the behavior. We conduct a small case study that shows: (i) VizRob helps engineers solve intricate debugging scenarios and (ii) VizRob is perceived as filling a relevant gap within the current tools to build robotic behaviors.

For live programming, we introduce LRP, a live domain specific programming language to build robotic behaviors. While we detail the importance of LRP we also present a controlled experiment to measure the real impact of LRP in behavior-based robotics. We found that LRP does **not** significantly outperform a non-live programming language in program comprehension nor in program writing. However, the feedback of test subjects indicates their preference for the live programming system. The results of this work seem to contradict the studies of live programming in other areas. We learned that the complex API chosen in this work has a strong negative influence on the results. To the best of our knowledge, this is the first in-depth live programming experiment in a complex domain.

Acknowledgments

Primero quiero agradecer a mis amigos, a aquellos que no están presentes conmigo en la universidad. Gracias por ayudarme a distraerme y salirme de los ritmos agobiantes de este trabajo. A aquellos que están siempre presentes en la universidad, también quiero darles las gracias por acompañarme mientras estoy por la universidad. Gracias por las conversaciones y el entendimiento de como avanzar en algo que nadie te prepara.

Gracias también a mis compañeros, no solo del doctorado, sino también de pregrado. Conversar sobre temas de la academia y avanzar en proyectos, ya sea separado o en conjunto, me han ayudado a crecer. En especial quiero nombrar a Max y Seba por dejarme guiarlos en sus primeros años de ingeniería. También nombrar a Rodrigo por permitirme ayudarlo en sus últimos años de ingeniería y comienzo de su magíster.

Quiero agradecer a Johan, mi primer profesor guía y la persona que me impulso en este desafío. Sin su constante motivación y apoyo, el camino de este doctorado hubiese sido aun más tortuoso. E incluso cuando tuvo que retirarse, ha estado siempre presente.

Junto con Johan, quiero agradecer a Alex. Debe ser difícil guiar a alguien con el avance que ya tenía. Quiero agradecerte por confiar en mí y ayudarme en la segunda parte de mi carrera. Johan me dejó en buenas manos.

Paula, gracias por apoyarme al entrar en este desafío y ayudarme a enfrentarlo en sus primeros años. No todo resultó bien, pero sin tu apoyo no podría haber llegado hasta este momento. El primer viaje hubiese sido aun más duro sin tu presencia.

Lissete, gracias por darme una segunda oportunidad. Logré crecer y entenderme aun más contigo, apoyaste no solo mi trabajo sino también mi independencia. Motivaste un gran momento de mi vida y estuviste ahí en ese paso, gracias.

Dicen que la tercera es la vencida, gracias Vanessa por concederme esa oportunidad. Gracias por creer en mí, incluso más que yo mismo. Ya sea por coincidencia o por haberlo planeado, no olvidaré nuestros viajes.

Caro, muchas gracias por todo. A pesar de lo que he hecho, has estado ahí. Y queriéndolo o no, has sido un gran apoyo en todos mis procesos.

A continuación, quiero agradecer a todos los grupos de trabajo que me han permitido aprender y crecer en el transcurso de mi carrera. Primero, quiero agradecer a Inria Lille.

Mis primeros viajes me llevaron a Inria y me ayudaron a creer en mí y en mi trabajo. En particular, Marcus, Stéphane, Esteban, Martín, Guille, Santi y Pablo, agradezco su apoyo y compañía mientras estuve allá. También quiero agradecer a CAR team en Ecole de Mines, Douai. Luc, Noury y Jannik, gracias por ayudarme en la primera etapa de mi carrera y en la construcción de LRP.

Luego quiero agradecer a la comunidad de robótica que ayudé a crear. Mati, Gonzalo, Luz, Pablo, Cris y Joakin. No solo me han ayudado a crecer como profesional, sino también a entender este mundo tan complejo de la robótica.

Agradezco al equipo ISCLab. Son muchos para nombrarlos a todos, pero gracias por apoyarme en mis experimentos y darme consejos e ideas para mis proyectos.

Finalmente quiero agradecer al equipo de robótica de la universidad Hombreakers team. Gracias a su participación activa en mis experimentos, logré tener resultados importantes de las herramientas que he creado. Además, quiero agradecer a Davic Conner, Loy Vanbeek y Cristopher Gómez por su apoyo en la creación de otra de mis herramientas de este trabajo, VizRob

Para finalizar, quiero agradecer a Conicyt[1] por financiar mis estudios a través de la beca CONICYT-PCHA/Doctorado Nacional/2015-21151534. También agradecer a Object Profile por el apoyo y financiamiento de mis estudios.

[1] `http://www.conicyt.cl`

Contents

Chapter 1

Introduction

Automation is now part of any modern society. Industrial manufacturing of physical goods, such as packaging food, assembling cars and product inventory is carried out by physical computers that physically change the real world. These special kinds of computers are called robots.

Robots allow not only for automation of multiple, error prone, activities. They also allow humans to perform dangerous tasks in areas where it may be difficult or even impossible for a human to be, such as volcanoes or outer space. To perform the tasks, software plays a key role within robots. Software made for robotic behaviors requires specially designed data-structures and algorithms, not only for the different tasks that the robots perform, but also for the different robots involved in the tasks.

For software to work in the special embedded systems that are robots, robotic engineers focus primarily on performance of their software. This makes developers neglect other important attributes of a software system, such as maintainability and usability. When developers neglect these attributes, it makes the task of reproducibility, study or implementation for other developers prohibitively expensive [1].

One of these neglected tasks is the development of a software solution for a robot. Typically, the development of robotic software entails a number of steps: thinking about a solution to a problem, translating that solution into code, compiling the code into a program, deploying the program into a robot (or a simulator) and finally, testing the solution. When the robot does not behave as expected, these steps need to be repeated until the desired behavior is obtained.

In this dissertation we study the development of Behavior-Based Robotics. This kind of development focuses on building a behavior where a robot reacts (using its actuators) to external stimuli (captured by its sensors). For this kind of development a reactive paradigm is used to make the robot behave in an open, unpredictable world. A complete, pre-built model of the world lacks this unpredictable nature [2]. Behavior-Based Robotics relates to incremental building and testing of behaviors in a real world with a real robot.

Because of the several steps between thinking of a robotic behavior solution and testing the solution on a real robot, if the behavior fails, it is especially troublesome to understand why it failed. To address this problem, we find a practical solution to map the behavior of a robot to the developers' mental model. This is an important part of development, often neglected by robotic engineers.

To achieve this mapping, first, developers need to quickly understand how exactly the robot is behaving. A possible solution is to develop tools that help the running program on the robot give meaningful feedback to developers. For that purpose, we developed VizRob, a visual tool to give run-time information to developers, taking into account the specific domain of robotic behaviors.

With meaningful feedback we expect developers to better understand how the robot behaves, diminishing the need for repeating the mentioned steps as much as possible. However, the steps between their solution and the actual behavior of the robot still remains. To reduce the number of steps of building robotic behaviors, we implement an immediate connection with the developer and the robot. The combination of meaningful feedback and immediate connection achieves *Live Programming* in a robotic behavior domain. Live programming allows developers to have complete control of their programs, making changes in real time and so they can see what is happening via feedback given by the program. To achieve live programming in behavior-based robotics, we implemented Live Robot Programming (LRP), a live programming language to build robotic behaviors.

1.1 Behavior-Based Robotics

A behavior-based robot works by reacting to a stimulus of the environment, building complex behaviors from simple ones. Arguably, a behavior-based robot should work better in a 'real' world and not in an abstract model of it [3]. When working with behavior-based robots, it is important to quickly test the behaviors that are built into the robot.

Development of behavior-based programs consists of creating the overall behavior of a robot, taking into account inputs generated by external algorithms such as image recognition and map localization, and sending instructions to, *e.g.,* a robot navigation or object grasping algorithm. The robotics research community makes such external algorithms available online, relying on middleware to enable the joint operation of all these pieces of computation.

Considering the languages or APIs used to program the behavior of a robot, arguably the most successful are those based on nested state machines, *e.g.,* XABSL [4], the Kouretes Statechart Editor [5] or SMACH [6]. Such machines are said to naturally map to the problem domain, and multiple RoboCup teams have won several leagues in the RoboCup competition (such as robotic football and service robot) using these tools.

The steps for building a behavior-based program are the same as those for building any robotic software: defining the requirements of the behavior, translating the requirements to source code, compiling the code, deploying the program into the robot and testing the

behavior. This produces a *cognitive distance* between the developer and the final behavior, as shown in Figure 1.1.

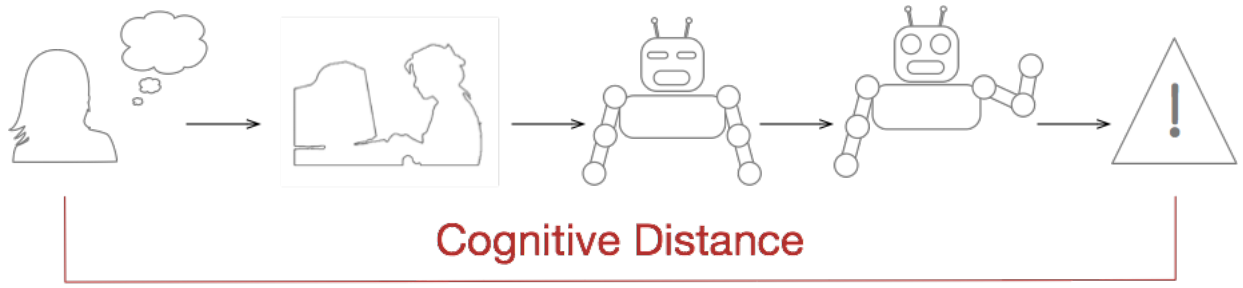

Figure 1.1: Cognitive distance in robotic development.

The intelligence of the robot in behavior-based robotics depends on the interaction of the robot with the real world [2]. A long cognitive distance in this domain considerably slows down development because the robot has to interact in the real world, and this interaction takes time. This makes experimentation with different behaviors prohibitively expensive.

1.2 Meaningful Feedback

Developers implement different ways in a program to receive feedback of the program itself. This feedback can give important information to developers, such as the state of the execution, response of algorithms and value of variables. One typical example is the use of logs in programs: developers normally see them in a terminal where the program is executed or in a textual file. The feedback received by developers in robotics is normally given by logs in a terminal and the behavior of the robot itself.

In the special domain of behavior-based robotics, we can achieve better feedback by taking this domain into account. For nested state machine development, there are visualizations of the program that show the specific state of the robot. However, there is a lot of information lost, *i.e.,* that is not present in these visualizations. Just by looking at the active state of the machine, developers are not sure if the robot is executing the expected behavior or not. Even when they are sure that the robot is failing, it is difficult to precisely know the cause the failure stems from. With meaningful feedback one should be able to contrast the observed robot behavior with the intent of developers with better speed and accuracy, in comparison to having no dedicated support.

1.3 Immediate Connection and Meaningful Feedback: Live Programming

By having meaningful feedback we can make the feedback loop more understandable for developers. However, the cognitive distance between developers and what the program is

doing is still wide. In robotic development this distance is even larger. Besides writing and compiling their programs (and do several other necessary software development practices), developers have to upload the code of the program into the robot and also they should wait for the robot to move and interact with the real world, a process that can take a considerable time.

To reduce the cognitive distance we can add an immediate connection between developers and their robots. With immediate connection developers can change their programs on-the-fly and see their changes in the robot, without going through all the classical steps. The combination of immediate connection and meaningful feedback is called *Live Programming* in software engineering.

Live programming has garnered attention thanks to the widely commented talk by Bret Victor at CUSEC'12 [7]. Its origins can however be traced back to the early work of Tanimoto on VIVA [8]. In this work, an argument is made for maximizing feedback to the programmer through a 'continuously active' system: every edit action triggers computation of the program and the display of computed values is updated live, as inputs vary.

In a nutshell, live programming postulates that developers benefit from an immediate connection with the program that they are making. Languages for live programming therefore provide for an immediate, 'live' and meaningful feedback on program behavior. With this feedback, developers are expected to build accurate mental models of the system when its execution is observed. This permits, on the one hand, for extremely rapid creation of program behavior as the effects of variations in the behavior are immediately visible. On the other hand, it immediately reveals bugs that are due to the programmers' mental model of the executing program differing from the actual behavior.

For an example, let us look at designing the layout of a web page. The layout is defined, but it needs to be implemented. A developer builds the source code and adds styling to make it look like it should. However, as most developing activities, it is likely to fail at first. Then, the developer should go to the source code, map why the layout it is not looking as it should, modify the source code and finally, re-open the page. The developer completes this loop several times until she or he reaches the desired aspect of the web page.

Thanks to modern web browsers, this process is significantly faster as it avoids the explicit loop. The developer can modify the style of the page while the page is open in the browser, seeing the changes of this modification right away. This shortens the feedback loop, empowering the developer to make explicit changes. The web page, thanks to the browser capabilities, gives explicit feedback of how it is looking. In theory, this shortened feedback make web page development faster.

1.4 Problem Statement

Even with the increase of tools for developing robotic behaviors, there is still a lack of adequate tools to comfortably understand and debug them. Developers mostly uses `print`

statements or log files for debugging [9], and even the current visualization for robotic behaviors in the case of nested state machines may be limited for debugging purposes, where most of them shows the static visualization of the machine and the current state being executed [6, 10].

This lack of debugging tools can be especially troublesome in the case of robotic behaviors using nested state machines, because of the long cognitive distance in this kind of development. To address this problem, we define the follow research questions:

RQ1: In the special context of behavior-based robotics, how the behavior of a program using nested state machines can offer meaningful feedback to developers?

Visualizations of programs often give feedback to developers on what the program is doing. When using nested state machines, there are several visualizations of the static model of the machine, including which state is running when the robot is performing an action. However, a behavior-based robotic program contains more information that may be useful for developers. It is important to identify which piece of information can be used without giving extra burden to developers on tagging this information in their programs.

RQ2: Is it possible to reduce (or remove) the cognitive distance using a practical solution in a behavior-based robotics context?

Live Programming postulates reducing the cognitive distance, however the behavior-based robotics development has yet to see practical live programming languages. By practical we mean a live programming language that operates with several types of robots and not only one robot specifically built to work with live programming.

There is a particular study in a general robotic context that does not take into account a behavior-based context [11]. Nevertheless, Hancock [12] developed a live programming language to program robotic behaviors: Flogo I. However, this language only works in one robotic system, making it impractical as a complete solution for several robotic systems.

RQ3: If the reduction (or removal) of the cognitive distance is possible, does this improve the developers productivity of behavior-based robots?

In a general context, live programming states that it improves developers productivity by reducing the cognitive gap via mapping the behavior of the robot to the developers' mental model. There are several studies of validation of live programming language, however not in a behavior-based development, nor even in complex scenarios [13, 14, 15]. Moreover, there are several more studies on how developers behave using a live programming language, but they do not measure productivity [12, 14, 15].

1.5 Goals & Hypotheses

Goals

The main goal of this dissertation is to improve developers productivity when working with behavior-based robotics. For this, we develop tools to map the developers' mental model to the behavior of their robots and study the impact of these tools in some development practices.

In particular, we study two development practices for behavior-based robotics: Program Understanding and Program Writing. The first refers to the ability of developers to understand program already being written. In the second practice developers write a complete behavior program for a robot.

Finally, we measure the productivity of developers by addressing the speed and accuracy of both development practices. The speed is given on how fast developers understand and/or write their programs. Accuracy measures the correctness to complete both development practices.

Hypotheses

Taking into account the previous goal and the research questions stated in Section 1.4, we propose the following hypotheses:

H1: It is possible to have a live programming language for nested state machines in a robotic context.

For studying behavior-based robotics, we use the popular paradigm of nested state machines. If this paradigm can support live programming, then it is a step closer to achieving live programming in behavior-based robotics.

Moreover, we need to connect these state machines to the robots themselves, not only creating a solution for one robotic system, but for several of them. Even when there are tools to develop nested state machines in the context of robotic behaviors, just a few of them support live programming. Moreover, those that support live programming are specially made for only one robot.

If we can achieve live programming for nested state machines in a robotic context, then we can have a convenient tool to map developers' mental models to the behavior of their robots.

H2: Live programming improves the speed and accuracy of program understanding and program writing in behavior-based robotics compared to a classical programming language paradigm (*i.e., *non-live).

When we review other live programming languages, we find that there are some studies of the advantages of live programming, but there are no studies in a robotic context, nor even in a practical, non-trivial setting to which we can compare. In particular, in a behavior-based robotics scenario, there are dependencies on complex APIs that connect with the actual robot and with complex algorithms. The previous experiments seem to validate this hypothesis, however we can not generalize those results in a non-trivial setting.

If we achieve a mapping between developers' mental models and the behavior of their robots, it is important to assess how useful this mapping is in the complex scenario of building robotic behaviors.

Roadmap

To answer the research question and validate the hypotheses previously mentioned, we first research the current development practices of robotic engineers when building robotic software in general, and behavior-based robotic software in particular.

After understanding how developers build their programs, we then research what the typical debugging practices of robotic engineers are and the complex scenarios they experience when building robotic behaviors. We noticed how the debugging may improve using software engineering techniques. In particular, we first create a tool to help robotic developers in understanding better the behavior of the robots that they built: VizRob. With VizRob we also understand the special characteristics that exist in the robotic development world, such as long development processes because the robot takes time executing their behavior in the real world and how robots actually interact and change the real world.

By having a better understanding of behavior-based robotics and achieving meaningful feedback in that context, we then focus on achieving live programming. With live programming in nested state machines in robotic behaviors, we can reduce the cognitive distance between developers and the behavior of their programs.

We then test our own live programming language by actually building behavior of robots in the real world. Finally, we test if this live programming language actually improves the developers productivity in the complex scenario of building robotic behaviors. With that, we can finally understand if live programming is a suitable technique on reducing the cognitive gap of developers' mental model and the behavior of their robots.

1.6 Results

Here we summarize our main results by answering our main research questions and hypotheses presented before. For more detail information on the tools and methodology used in our studies, we refer to their respective chapters.

1.6.1 Research Question 1: VizRob

For answering RQ1, we conducted an informal survey and reviewed several learning materials for robotic development. With all these taken into account, we developed VizRob, a set of high-level visualizations to understand robotic behaviors. These visualizations use more information than the previous debugger tools already presented in the robotic domain and the specific domain of robotic behaviors. VizRob is specifically made to give a meaningful feedback from the behavior of the robot to developers.

With the developed tool, we conducted a case study to gather the perception and usability of VizRob. Our case study shows: (i) VizRob helps engineers solve intricate debugging scenarios and (ii) VizRob is positively perceived as filling a relevant gap in the current tools for building robotic behaviors. A complete description of VizRob an its case study is in Chapter 4.

1.6.2 Research Question 2 & Testing Hypothesis 1: Live RobotProgramming

For answering RQ2 and testing H1 we developed Live Robot Programming, LRP. LRP is a live programming language for nested state machines with connection to several robotic systems. With liveness we do not only achieve a mapping of the mental model of the robotic behavior with the developer, but also developers can immediately change the behavior, improving this mapping on the fly. As VizRob, LRP is positively perceived by developers.

We accept H1: LRP is a complete solution for developing robotic behaviors using live programming that can be used with several types of robotic systems. For a complete description of LRP, we refer to Chapter 6.

1.6.3 Research Question 3 & Testing Hypothesis 2: Controlled Experiments

With LRP, we can focus on our last hypothesis and the third research question. We can use LRP to study the impact of live programming in a robotic behavior context. We designed and conducted two controlled experiments for program comprehension and program writing. In both experiments we measured the speed and accuracy of developers.

However, we found that **LRP does not improve** productivity in these areas against a classical approach of developing robotic behaviors. This novel result even contradicts previous experiments on live programming. Hence we reject H2. Nevertheless, the perception of the subjects of the experiments contradicts this result: LRP is perceived positively by developers.

In Chapter 7 we present a complete description of the experiment and give insights on why LRP does not increase the productivity of robotic behavior development.

1.7 Contributions

The main contributions of this dissertation are summarized below:

- **Live Robot Programming:** The first live programming language for nested state machines that is not coupled to any robotic system [16]. We developed several bridges to work with LRP, in particular: Robot Operating System (ROS) [17], EV3 Lego Mindstorm [18][1].
- **Experiments:** We tested LRP to see if live programming improves the productivity of developers. The results of our experiments told us that Live Robot Programming does **NOT** improve productivity. As far as we know, this is the first live programming experiment in complex scenario [19].
- **Discussion of live programming in a complex scenario:** The results we obtained in the LRP experiments differ from the rest of the literature, where live programming was study in other contexts. Because of our surprising results, we give several reasons on why we believe live programming does not work well in the complex domain of robotic behaviors. These reasons are: complexity of the robot API, missing opportunities and limited scope of live programming [19].
- **VizRob:** Improving the feedback of LRP is important as it gives developers more knowledge on how the robot is behaving. We implemented VizRob, a novel tool with several high-level visualizations of robotic behaviors [20]. We expect to improve the feedback of LRP in the future, which may improve the development of robotic behaviors.

1.8 Associated Publications

This work has produced the following conference and workshop publications:

- **Miguel Campusano**, Alexandre Bergel: *VizRob: Effective Visualizations to Debug Robotic Behaviors.* Third IEEE International Conference on Robotic Computing (IRC), 2019, pp. 86–93.

[1]The official website features the instructions for all previous systems and additional robots that support LRP, as well as information on LRP itself: `http://pleiad.cl/lrp`. Accessed: 04-07-2019.

- **Miguel Campusano**, Johan Fabry, Alexandre Bergel: *Live Programming in Practice: a Controlled Experiment on State Machines for Robotic Behaviors.* Information and Software Technology 108 (2019) 99–114.
- **Miguel Campusano**, Johan Fabry: *Live Robot Programming: The Language, its Implementation, and Robot API Independence.* Science of Computer Programming 133 (2017) 1–19.
- **Miguel Campusano**, Johan Fabry: *From Robots to Humans: Visualizations for Robot Sensor Data.* IEEE 3rd Working Conference on Software Visualization – VISSOFT, 2015, 135-139.
- Johan Fabry, **Miguel Campusano**: *Live Robot Programming.* Advances in Artificial Intelligence – IBERAMIA 2014, number 8864, pp.445-456, 2014, Springer-Verlag.
- Pablo Estefó, **Miguel Campusano**, Luc Fabresse, Johan Fabry, Jannik Laval, Noury Bouraqadi: *Towards Live Programming in ROS with PhaROS and LRP.* 5th International Workshop on Domain-Specific Languages and models for ROBotic systems – SIMPAR 2014.
- **Miguel Campusano**, Johan Fabry: *An Interpreter For Live Robot Programming.* 33rd International Conference of the Chilean Computer Science Society – JCC 2014, 2014.
- Johan Fabry, **Miguel Campusano**: *Live Programming the Lego Mindstorms.* International Workshop on Smalltalk Technologies – ESUG 2014.

1.9 Dissertation Outline

This dissertation is structured as follows:

- **Chapter 2: Behavior-Based Robotics**, describes what behavior-based robots are and how to implement them.
- **Chapter 3: Programming Robotic Behaviors**, introduces the systems and tools used in this dissertation to program robots and robotic behaviors.
- **Chapter 4: Meaningful Feedback: VizRob**, in which we introduce VizRob, a set of high-level visualizations to achieve meaningful feedback of robotic behaviors.
- **Chapter 5: Live Programming**, gives a complete description of the concept of live programming.
- **Chapter 6: Live Programming in Robotic Behaviors: LRP**, introduces and completely describes Live Robot Programming, a live programming language to develop robotic behaviors using nested state machines.
- **Chapter 7: LRP in Practice: Controlled Experiments**, describes the controlled experiment we used to answer Hypothesis 3. This chapter also contains a discussion on why LRP does not improve the productivity of robotic behavior development.
- **Chapter 8: Conclusions**, which concludes our dissertations with the main results of this work.

Chapter 2

Behavior-Based Robotics

This dissertation focuses on improving the efficiency of development of behavior-based robots. Developing behavior-based robots means building the overall behavior using external stimuli of the real world, captured by sensors. The robot reacts to these stimuli with its actuators (*e.g.,* motors, speakers, lights). The stimuli can be processed by dedicated algorithms such as computer vision, to know exactly how the robot should react. For example, a robot can follow a red ball while avoiding yellow balls in the way. In this example, the stimuli are the constant flow of images of the balls and the reaction is how the robot should move: follow or avoid.

To better understand the coming chapters of this dissertation, this chapter lays out the foundation of behavior-based robotics. Our work heavily focuses on developing tools to program behaviors for robots, which is why we need to understand what a behavior-based robot is and why it is important.

This chapter first introduces the commonly accepted primitives of robotics: Sense, Plan and Act. Section 2.2 defines the concept of behavior-based robotics. Finally, this chapter presents us with one way of developing behavior-based robotics: Nested State Machines.

2.1 Introduction to Robotics: Sense, Plan, Act

A number of definitions have been proposed by the robotic community as to what a robot is and what it means for a robot to be considered *intelligent*. In this dissertation, we consider a robot to be a mechanical creature that can interact with the world and possibly change it. An intelligent robot is the same mechanical creature which can function autonomously [21].

Murphy [21] gives an on point explanation about this: "'Function autonomously' indicates that the robot can operate, self-contained, under all reasonable conditions without requiring recourse to a human operator. Autonomy means that a robot can adapt to changes in its environment (*e.g.,* the lights of a room turn off) or itself (*e.g.,* a part [of the robot] breaks)

and continue to reach its goal".

For a robot to be autonomous, there is an inseparable, commonly accepted triad: Sense, Plan, Act [21]. Sense is when the robot takes information from the outside world using its sensors and parses the information for other functions to use it. Plan is when the robot takes the parsed information, processes it and figures out what it should do. Finally, Act is the phase where the robot does something with its actuators, based on the Plan phase.

Depending on how this triad acts, there are three different paradigms on how intelligence on robots works: Hierarchical, Reactive and Hybrid paradigms.

Hierarchical Paradigm. This paradigm strictly follows the triad sequentially. In the Sense phase, the robot senses its environment, making a complete model of the world. In the Plan phase, it then uses the model of the world built on the previous phase and plans an action to a desired goal. Finally, in the Act phase the actuators of the robot operate according to the plan designed in the previous phase [22].

This paradigm encounters two big problems [2, 21]. First, the *Frame Problem* is the inviability of representing the world into a viable model. This is due to the complexity of the world model and all possible cases that the world presents. For example, in a room with a door, there are two possibilities; the door is open or closed. If we keep adding elements to this room, the number of possibilities scale too quickly to be represented in a reasonable computer model.

Second, the *Closed World Assumption* assumes that the model of the world is complete (*i.e.,* the model has all the information of the world) and static (*i.e.,* the world does not change over time). The robot must have all the necessary information to elaborate an action plan to a desired goal. In the previous example, if we add a moving object in the room (*e.g.,* a walking person) the robot can not track this new information to plan a desired action.

The two problems can be resumed as: the robot can not track uncertainty in the world. The robot environments are unpredictable by nature, especially for robots working in the proximity of people. Even when everything works as expected, sensors are susceptible to noise and actuators can be unpredictable at some point (*e.g.,* mechanical failure) [23]. New paradigms address the problem of uncertainty [22].

Reactive Paradigm. The reactive paradigm was born as a consequence of the problems of the Hierarchical Paradigm. To address these problems, researches looked into biology and studied how animals behave in a complex world [2, 21].

In this paradigm, the Plan phase is removed. There is a popular belief that information processing and planning only diminishes the performance of the robot, even when they may be efficiently designed [22].

The Sensor and Act phases are linked together, creating a behavior. To create more complex behaviors, there are multiple instances of the Sensor-Act pairing together [21].

The reactive paradigm solves the problems of the Hierarchical Paradigm, the Sensor-Act

coupling allows for tracking uncertainty in the world. Moreover, the removing of the Plan phase allows for fast execution time.

Hybrid Paradigm. In the end, the Reactive Paradigm falls short in producing complex tasks that require some sort of cognitive features [22]. In the hybrid paradigm, the Plan phase is executed separately from the Sense-Act instance, decoupling the Plan state from the execution [22]. This allows for the combination of the rapid execution of the Reactive Paradigm within the cognitive capabilities that added the Plan phase.

2.2 What is Behavior-Based Robotics?

Behavior-Based robotics is not a new concept, as said in the survey of Silva and Ekanayake [22], one of the first robots with a reactive behavior was built in the 1950s.

The basic block of a behavior-based robot is a behavior. A behavior is a reaction to a stimulus of the environment, which a robot should interact with. Then, behavior-based paradigm can be seen solely as a reactive paradigm [2, 22]. As mentioned early, the reactive paradigm works by *Sense* and *Act* constantly, removing the *Plan* phase [22]. When building behaviors in this way, explicit representations and models of the world are not necessary [3].

To build robotic behaviors is to incrementally build the intelligence of the robots. At each step, the behavior allows the robot to 'live' in a real world. With more time and effort, the behavior of the robot in the real world gets more complex.

The key aspects for building behavior-based robotics are [3]:

- **Situatedness:** The robot 'lives' in a real world, not in an abstract representation of the reality.
- **Embodiment:** The robot has a physical presence and its part of reality.
- **Emergence:** Intelligence of the robot arises from the interaction within its environment. It is not a property of the environment nor the robot, but the results of the combination of both.

Because of these properties behavior-based robotics is based on building robots in the real world, with real and active interactions and testing them in the real world [3].

A behavior is an abstract concept, and to implement it we first need to give a concrete definition of what is a behavior. In particular, we need to define [2]:

- The right behavioral building blocks.
- What is a primitive behavior.
- How to coordinate different behaviors.
- How to encode behaviors with sensors and actuators.

The implementations of a behavior-based system need to define every one of these points.

For every implementation there can be many definitions and many ways to implement these definitions.

In this dissertation we study one particular implementation: Nested State Machines. While this may be implemented in many different ways with different definitions of the previous points, many of these implementations share common grounds.

2.3 Expressing Behaviors: Nested State Machines

There are several ways to express the behavior of a robot, such as Stimulus-Response Diagrams [2], Skills [21], or Nested State Machines (NSM). In this work we focus on NSM. An NSM is a computation model with mechanisms to explicitly specify what the program should be doing given certain conditions [21]. NSM describes the overall behavior of a robot, but the concrete implementation may vary [21].

Programming languages based on NSM are arguably the most successful for robotic behaviors: *e.g.,* XABSL [4], Kouretes Statechart Editor [5] and SMACH [6]. These machines are used to program robotic behaviors in the RoboCup competition, with notable results.

In a nutshell, a simple behavior is represented as a state in the NSM. When a stimulus of the world needs to be reflected in the behavior of the robot, the machine *transitions* from the running state to the desired state, changing the behavior of the robot.

NSM are useful when describing aggregations, sequences and concurrent behaviors [2, 21]. They are a natural mechanism to describe relationships between behaviors in a high level abstraction [2]. Also, NSM makes explicit to developers the active action the machine is doing in the active state of the machine [2]. However, they are not useful for expressing a single behavior, which is a trivial NSM with only one state [2].

In NSM, a stimulus can be the data of one or several sensors, or the processed information of these data. Moreover, the machines should have a way to encode the output of a resulting behavior to the actuators of the robot [2]. Making the robot "move" is different for different robots.

Because in this work we study behaviors on several implementations of NSM, we do not focus on how the coordination mechanisms work, though the implementations should have some kind of coordination mechanism. We also do not focus on how the programming language encodes behaviors, but they need an encoding to make the robot actuate in the real world.

In the following sections we explain the general concepts of the state machine model. We also give details on the concept of a *nested* state machine. We then explain how developers can achieve concurrency using NSM.

2.3.1 General State Machine Concepts

When we mention the concept of State Machines, we refer to a modification of the classical mathematical model of Finite State Automata. The classical model of finite automata consists of a tuple $(S, \Sigma, \delta, s_0, F)$, where:

- S: Finite set of states.
- Σ: Finite set of input symbols, called an alphabet.
- $\delta : S \times \Sigma \to S$: Transition function.
- $s_0 \in S$: Initial state.
- $F \subseteq S$: Subset of accept or finish states.

In a nutshell, the computation of the automata starts in the initial state s_0. The states are connected to each other indicated by the transition function δ. This function is a partial function, it is not necessary to define δ for every combination of states and symbols. These transitions trigger depending on an input defined in the domain of the alphabet Σ. Whenever a transition triggers, the automata changes its current state, as defined in the transition function δ. It is said that the input is accepted whenever the automata ends in a finish state $f \in F$.

In the context of robotic behaviors a state $s \in S$ represents a behavior of the robot. A transition represents a change of the behavior of the robot. The alphabet Σ in this context is a set of events that represents stimuli detected by the robot. Transitions are bounded to these events. The different behaviors of the robot are coordinated by the transitions and how they trigger, as defined in the transition function δ. In a visual way, the stimuli are represented by arrows from one behavior (state) to another, for which each arrow represents the *releaser* of the respective behavior [21]. How this coordination works in a state machine depends on its concrete implementation.

To represent a behavior using state machines, the states, transitions and events possess special characteristics. The states contain different types of *actions*. These actions normally contain pieces of source code that connect the program to the robot, depending on the specific programming language. The actions may be executed at different times, depending on the context of the execution of the state machine. For example, there are actions that are only executed once, when the state becomes active (*e.g.,* entry actions); other actions are executed once, but when the state becomes inactive (*e.g.,* exit actions); there are actions that are executed constantly when the state is active (*e.g.,* active actions). The different types of actions a state may have and how they work depend on the concrete implementation of a state machine for robotic behaviors.

The transitions are associated to certain events of the execution of the behavior. Normally, when a certain set of events associated to a transition are emitted, the transition triggers, allowing for the change of the behavior of the robot. The events may represent stimuli from the environment, a timeout in the execution or any other kind of event necessary for the robot to work. Transitions may also possess actions that may change or not the context of the execution (*i.e.,* actions with side effects). This also depends on the specific implementation of the state machine.

Common uses of state machines present us with the need to specify a *start* state and several *finish* states. While the start state is important in the context of representing robotic behaviors, the finish states are not necessary. There are behaviors that the robot should perform without finishing them, *e.g.,* the robot is monitoring the entrance of a building. In this example, the behavior ends when someone tells the robot to stop. This is easily done by turning off the robot. The specification of start and accept states depend on the concrete implementation of the state machine.

All these concepts are exemplified in the behavior presented in Figure 2.1. This behavior is the same presented at the beginning of this chapter: a robot follows red balls and avoids yellow balls. In this behavior there are three states: `follow`, `search` and `avoid`. The start state is the `search` state: first the robot searches for the red ball before following it. This state contains the active action `move_camera` that makes the robot move its camera, searching for a ball. When it encounters a red ball, the event `e_red_ball` emits, triggering the associated transition. This behavior is intended to run indefinitely, that is why there is no finish state.

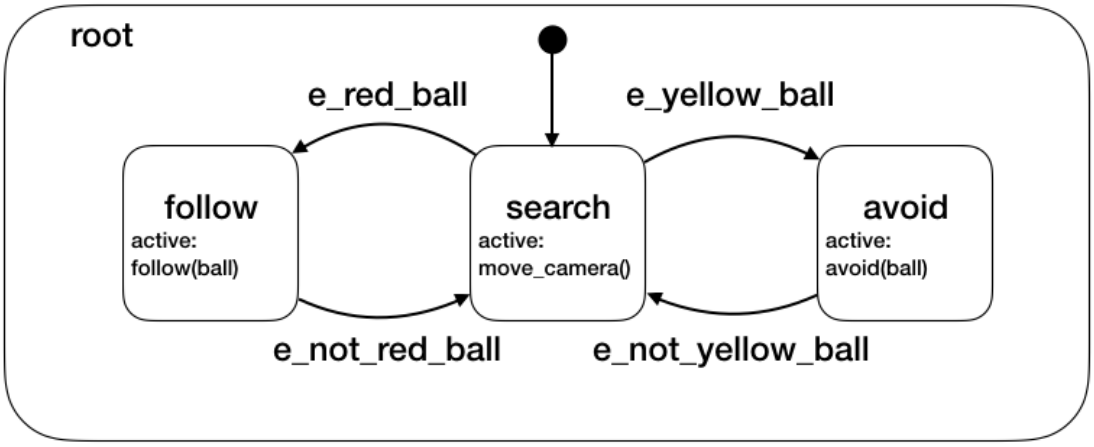

Figure 2.1: Example behavior using a state machine.

2.3.2 Nested Machines

An important property of behavior-based robotics is to incrementally build the behavior of a robot. A state machine can express a behavior using several nested machines. This is done by allowing a state to nest more states. Because of the intrinsic hierarchy of this operation (a state contains more states, those states can contain more states and so on) nested state machines are also called hierarchical state machines.

A way of implementing nested state machines is to allow a state to contain another machine. In this case, a state still could contain actions, but the active action would be the execution of the inner machine. The inner machines can go as deep as needed, depending on the developed behavior.

As we stated before, a state machine contains a start state and it may contain several finish states. When defining a nested machine, both concepts may have different interpretations. Using the start state to initialize the nested state machine seems logical. However, this can be redefined depending on the concrete implementation. For example, if the nested machine was executed previously and it should be executed again, the parent machine may decide to start the nested machine in its last executed state. This technique is commonly called *History States*.

In the case of finish states, it is important for the parent machine to manage what it should do. The parent machine may decide to transition to different states depending on the finish state that the nested machine ends. For example, a nested machine can have two different finish states, one when the behavior was successfully completed, and the other when the behavior failed. The parent machine then decides to transition normally if the nested behavior was successful, or to transition to a fall back state that executes an error recovery routine when the nested behavior failed.

Moreover, the parent state may have transitions that trigger independently of the execution of nested machine. The coordination mechanism of these nested machines depends on the concrete implementation of the state machine system. This implementation may decide to stop the execution of the nested machine, independently of the state being executed, or it may decide to wait for the nested machine to arrive to a finish state.

We can extend the previous example presented in Section 2.3.1 to use nested state machines. In the `search` state of the original behavior, the active action contains an instruction to move the camera of the robot. We can explicitly define this behavior by using a nested machine. This new machine contains 4 states that makes the camera go up, right, down, left and right again. Every movement is done for 1 second, until the corresponding timeout transition triggers. This extended behavior can be seen in Figure 2.2. Notice that we deliberately omitted actions inside some states to save space.

2.3.3 Concurrency

Another property that state machines may have is the ability to run several behaviors at the same time. Robots may execute two or more behaviors at the same time, especially if these behaviors are helping to achieve the same goal (*e.g.,* moving the body of the robot while moving the camera of the robot to find an object in the environment).

We can achieve concurrency in state machines mostly in two ways. First, by allowing a machine to execute more than one state at the same time. To do this, certain implementations incorporate at least two special states: one to split the execution, a fork state; and one to join the execution, a join state. The fork state can split the execution of a machine into several states, while the join state waits for all the paths to converge to one state, before continuing the execution of the behavior. There can be a special kind of join state where, instead of waiting for the execution of the different branches, it just waits for the first one to arrive to this state, then the join state interrupts the other executions.

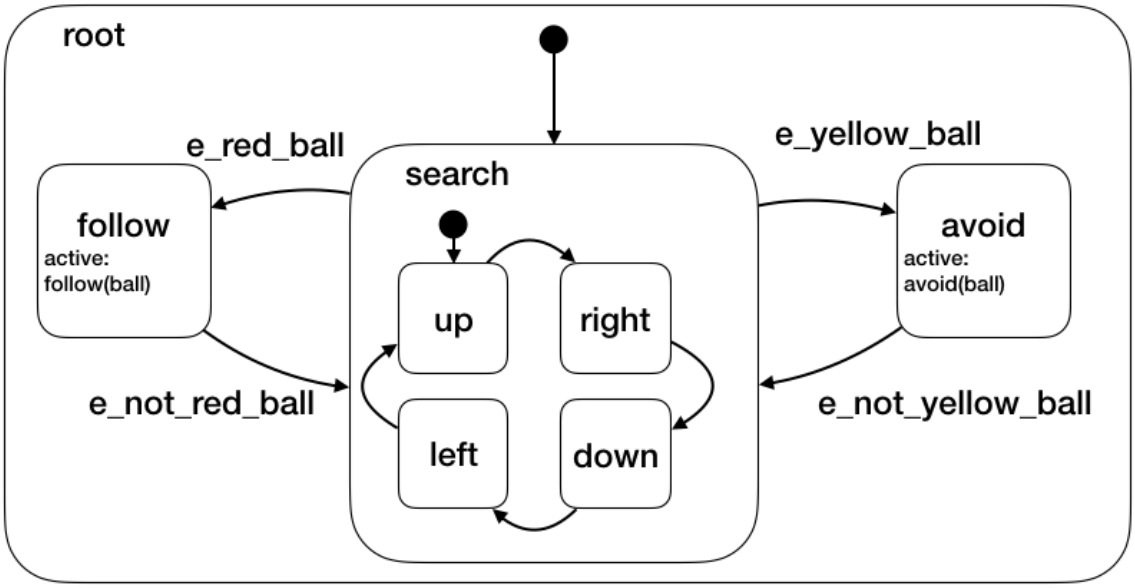

Figure 2.2: Example behavior using a nested state machine.

The second way to achieve concurrency is to use the concept of nested state machine explained earlier and to execute more than one machine per state. In this case, instead of fork and join states, the parent state should decide the coordination mechanism between the nested machines.

We again extend the behavior presented in Section 2.3.1 and Section 2.3.2. We extend the `search` state to support a more sophisticated searching routine. Now, instead of just moving the camera, the robot will turn at the same time. The turning behavior starts by turning the robot to its left side for 4 seconds, then to its right side for 4 seconds, before repeating its movement. With both behaviors running concurrently, the robot has a more wide searching area to look for the balls. For both behaviors to run concurrently, we use the concepts of two machines running at the same time, instead of using multiple states in a single machine. This new concurrent behavior can be seen in Figure 2.3

2.3.4 State Machines vs Statecharts

Statecharts is an extension of the concept of state machines defined first by Harel [24]. The Harel Statecharts is one of the first (if not the first) formalism to extend the concept of state machines with hierarchy and concurrency (and several other concepts). There are several implementations of this formalism, that take into accounts most of Harel concepts and rework another concepts depending on the specific implementation.

In our work, we could use the concept of Statecharts instead of State Machines. We decided to keep using the concept of state machine. The reason behind this is: if we use the concept of Statecharts, then the concrete implementation should take into account all (or most of) the concepts of Statecharts, in their classical definition. By using the concept of

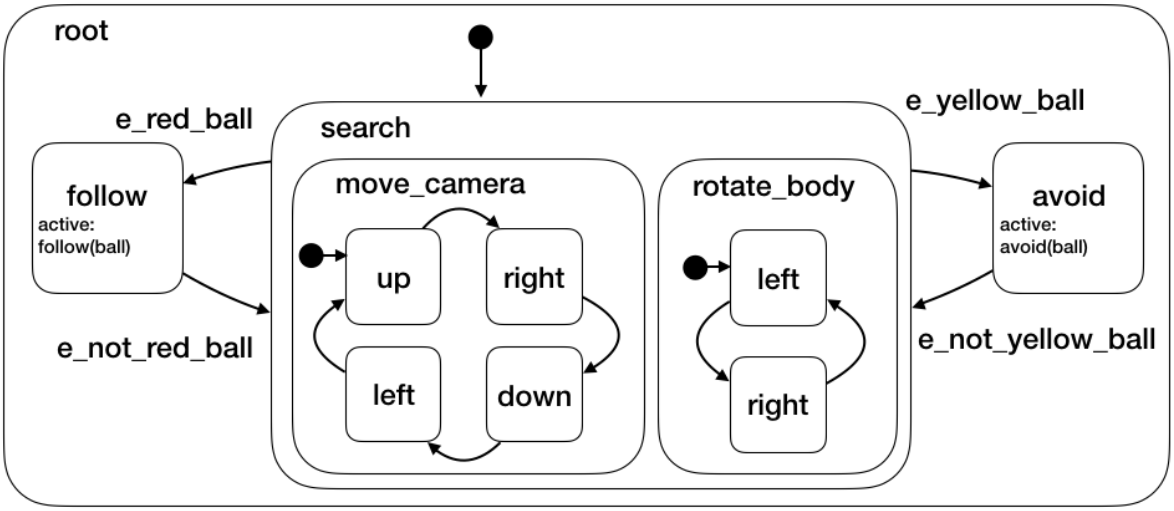

Figure 2.3: Example behavior using concurrent machines.

state machines, we can include more tools in our analysis without discarding some of them because they do not include all the concepts of Statecharts.

When the tools implement hierarchy, we called them Nested State Machines. If the implementation of state machines take into account concurrency, we explicitly state this. For example, a tool that implements hierarchy and concurrency, we say the tool implement concurrent nested state machines.

2.4 Conclusions

In this chapter we described the concept of behavior-based robotics. In a nutshell a behavior-based robot works as a reactive paradigm: it reacts to the stimuli of the environment, then act accordingly. We discussed Nested State Machines - a way to express behavior that is commonly used in robotics. We also discussed several properties that state machines may possess, in particular: nested machines and concurrency.

This dissertation focuses heavily on behavior-based robotics implemented with nested state machines. In this chapter we presented crucial concepts to understand the rest of this work.

Chapter 3

Programming Robotic Behaviors

In this dissertation we heavily use two tools to program robotic behaviors: ROS [17] and SMACH [6]. ROS is a popular middleware to work with robots that allows us to develop programs for robots while not worrying on the incidental complexity of working with robots. SMACH is the standard library to program behaviors using Nested State Machines in ROS. It is important to understand both SMACH and ROS to see their advantages and limitations and how they may affect the results of this work.

This chapter first introduces ROS and SMACH. Both of them are heavily used through this dissertation. It then presents several other tools to develop behavior-based robotics.

3.1 Programming Robots using ROS

Robots have a set of different sensors and actuators that communicate with the physical world. Because of this, development of robotic applications is a non-trivial process [25]. In this work, we develop and test tools for robotic behaviors without concern for the incidental complexity of developing software for robots.

Middleware services provide an approach that facilitates the development of robotic software [25, 26]. This is done by providing often needed functionalities that are well-structured and well-tested [25].

For this dissertation, we choose to work with the Robot Operating System (ROS), arguably, the most popular middleware for robots [17]. The popularity of ROS makes it the current de-facto middleware to use in robotics. ROS supports over 100 different robots and sensors, and an extensive software ecosystem providing a wide variety of possibilities for robotic development. ROS is a multi-language middleware, developers can choose among several programming languages like Python, C++, LISP.

ROS mostly works by using the *Publish/Subscribe* pattern. A sensor publishes its data

for any program that subscribes to the sensor. This program then processes the data and publishes it again to the actuators (motors). The motors are subscribed to a special channel to receive data that makes them move. Between this, any number of programs can subscribe and publish data to process even more complex data.

For example, let us take a robot with wheels and a bumper on the front and a behavior where the robot constantly moves forward while there is no obstacle, and if there is one, the robot should move backwards a bit, rotate and keep moving forward. There is a program that constantly *publishes* information to the motors to move forward. Also, this same program constantly receives the information of the bumper of the robot, because the program has *subscribed* to the bumper. When this program detects an obstacle on the bumper, the program *publishes* again to the motors to move backwards first and then rotate. Finally, the program goes back to *publish* to the robot to move forward again. Figure 3.1 illustrates this Publish/Subscribe program.

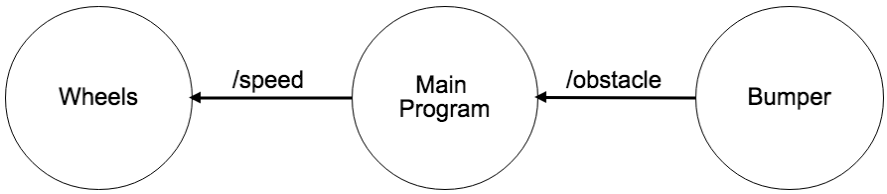

Figure 3.1: Example of a ROS system with three diferent programs communicating between each other.

3.1.1 Programs in ROS: Nodes & Topics

Programs in ROS are called *Nodes*. A Node is a modular process that performs a computation. A typical system in ROS is composed by many Nodes that communicate with each other by publishing or receiving data. In the previous example, every program that publishes and subscribes to a topic is a node as seen in Figure 3.1. The Main Program node processes information of the Bumper node and directs how the robot should move to the Wheels node.

Nodes do not publish or subscribe data directly to other nodes, they publish or subscribe to communication channels called *Topics*. Nodes can publish and/or subscribe to any number of topics. This means, a node can send or receive data to any number of nodes, including actuators and sensors. Figure 3.1 shows that the Main Program node is communicating with Wheels and Bumper node via two topics. The Main Program node subscribes to the same topic the Bumper node is publishing, the */obstacle* topic. Using the information of the */obstacle* topic, the Main Program publishes data to the */speed* topic, the same topic the Wheels node is subscribed, then the Wheels node uses this data to move the robot.

Topics send or receive any kind of typed data. The data is called *Message*. A message is a C-like struct filled with information. As any struct, messages can be recursive, *i.e.,* a message can store any other defined message. In our example, the structure of the message of the */speed* topic indicates the linear speed and the angular speed (*i.e.,* the struct contains two values). The message of the */obstacle* speed only contains one value, true if the robot is

touching something or false if it is not touching anything (*i.e.,* the struct contains a boolean value).

3.1.2 Systems in ROS: Packages

To support collaborative development, ROS is organized into packages. A package might contain anything that logically constitutes a useful module. For example, a package might contain several Nodes, Message definitions, configuration files, ROS-independent libraries, etc. In general, a package must have enough functionality to be useful, but not too much, to avoid that the package be heavyweight and difficult to use for another package.

One important component in a ROS system are *Launch Files*. A launch file is an XML file with enough information to run a system: nodes that need to be run, variables, dependencies to other launch files, etc.

3.1.3 Services and Parameters

Apart from the Publish/Subscribe type of calls in ROS, there is a classic Request/Response type of call: *ROS Services*. While the topics in Publish/Subscribe type of calls are asynchronous, ROS Services are used for synchronous calls.

Moreover, ROS provides a *Parameter Server*, where all the parameters defined in a robot are stored. The parameters can be obtained or changed while a ROS program is running. For example, one can assign the default speed of a robot to a parameter and change it on the fly with the parameter server. This variables can be retrieved and changed via ROS Service calls.

3.2 Programming Behaviors in ROS: SMACH

In the course of this work, we heavily use a library to program robotic behaviors, SMACH [6]. First, we use it as a research platform to understand how developers work with robot behaviors and analyze how we can improve this development. Then, we build state-of-the-art visualizations based on SMACH to help developers understand the behavior of their robots (see Chapter 4). Lastly, for this dissertation we develop our own behavior-based robotics programming language (see Chapter 6) and compare our language with SMACH itself (see Chapter 7).

We use SMACH because it is a popular library for building robot behaviors when developers use ROS. SMACH is used to explicitly describe well-defined tasks, removing the burden to developers to write well encapsulated code.

With SMACH, developers can build task-level applications with several types of machines

and states for a large range of problems. SMACH has been successfully used in several projects, including by the UChile Hombreakers team from the University of Chile[1].

The core of SMACH is independent of ROS. Nevertheless, it provides a ROS implementation with more features and it implements ready-to-go states for robots using ROS.

For information not presented in this dissertation, we refer to the published literature [6] and the official website: `http://wiki.ros.org/smach`.

3.2.1 SMACH Core

In SMACH, machines and states are implemented using Python classes. The execution of a behavior is defined in a method of the state class called `execute`. After the execution of the behavior, this method returns an outcome. An outcome is a string that is used to connect this state with another state, defining a transition. This transition is defined when the state is added in a machine.

Because machines and states are classes, developers can extend them to build new elements that better fit their own needs.

Every type of machine can include any number and type of states. SMACH supports nested state machines by allowing every type of machine to include any other machine (even machines of the same type). There are several types of machines included in SMACH:

- **State Machine:** Simple machine with no other special feature.
- **Concurrence:** Allows for multiple states to be running at the same time.
- **Sequence:** Allows for a sequential execution of states. The states inside this machine have only one outgoing transition that is automatically connected to the next state that is added in the machine.
- **Iterator:** Loop over states until a condition is met.

Also, in SMACH there are several types of states ready to use:

- **Generic State:** This is the base state with a general interface to define simple behaviors.
- **CBState:** This state allows developers to execute a callback without the need of extend from the Generic State.

SMACH allows developers to pass variables between states, this mechanism is called User Data. To work with user data, a developer should define a communication layer between states. This communication layer is done by defining input and output keys. The input keys enumerates all the values that the state has access to, and the output keys enumerates all the values that a state provides to another state.

[1] `http://robotica-uchile.amtc.cl/about.html`. Accessed: 04-07-2019.

3.2.2 SMACH and ROS

SMACH supports a ready-to-go integration with ROS. This integration allows developers to build applications with ROS using new states and features that are not accessible in plain SMACH. These new states are:

- **Service State:** Represents a service call from ROS. A service call is a request/response type of call (more on service calls in Section 3.1.3).
- **Monitor State:** This state allows developers to wait for a value from a ROS topic and execute a callback.
- **Simple Action State:** Represents an *actionlib* action from ROS. An actionlib action is an asynchronous request/response type of call.

3.2.3 Visualization

In this dissertation we focus on fast understanding of behavior-based robotics. A way to quickly understand the behaviors of a robot is to provide meaningful feedback of those behaviors to developers. A visualization of the program itself may provide meaningful feedback to developers. SMACH possesses its own visualization in the form of *SMACH Viewer*.

The visualization only works with ROS, where developers have access to the *Introspection Server*. The Introspection Server passes on information of the state machine at run-time by using the ROS publish/subscribe mechanism with the introspection server's own topics and messages.

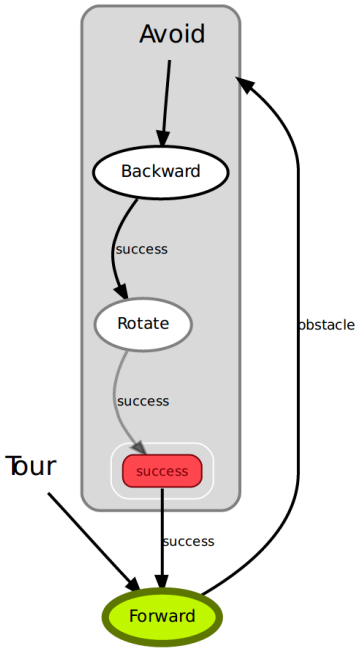

Figure 3.2: Smach Viewer visualization of the Main Program node presented in Section 3.1.

This raw information is passed to SMACH Viewer to visualize what is happening inside the

state machine. The visualization shows all the states in the state machine, marks the current state being executed and shows all the user data that are passed between states. Figure 3.2 show the visualization of the Main Program node presented in Section 3.1. The behavior of the Main Program node is represented with two states: Forward and Avoid. When there is an obstacle (represented with the *obstacle* outcome), the Avoid state is executed. This state has a nested machine with two more states: Backward and Rotate. Both states make the robot behave as the names suggest: the robot moves backward and rotates. After the robot rotates, it moves forward again until another obstacle is encountered. In this example, the robot is moving forward, as shown in the visualization with the state Forward colored green

In our work, we heavily use and compare not only SMACH, but also its visualization against our own developed tools. We focus on maximizing the meaningful feedback for behavior-based robotics.

3.3 More Behavior-Based Tools

In this dissertation we work closely with SMACH to program robotic behaviors. This is mostly because of its features, but also for its integration with ROS. However, there are more tools to program behavior-based robots. Section 2.3 presented several characteristics that concrete implementation of state machines may have. In Table 3.1 we summarize every related work presented in this section, including SMACH, with the features presented in Section 2.3. Moreover, we add columns expressing features not exclusive to state machine development, such as: visual programming, run-time visualization and the base language of the concrete implementation.

Tool	VP	V	BL	SA (En/Ac/Ex)	OS	FS	NM	C
KSE	Y	N	C++ (IDE)	Ac	N	Y	Y	Y
XABSL	N	Y	Prop. (DSL)	Ac	N	Y	Y	N
CABSL	N	Y	C++ (DSL)	Ac	N	Y	Y	N
NBE	N	Y	Lua (API)	En	N	N	Y	N
Gostai	Y	Y	Urbiscript (IDE)	Ac	N	N	Y	N
rFSM	N	N	Lua (DSL)	En/Ex	Y	Y	Y	Y
RAFCON	Y	Y	Python (IDE)	En	N	Y	Y	Y
SMACH	N	Y	Python (API)	En	N	Y	Y	Y
FlexBe	Y	Y	Python (IDE)	En/Ac/Ex	N	Y	Y	Y

Table 3.1: Features of the related work presented in this chapter. Columns represent if the tool supports: Visual Programming (VP), Run-time Visualization (V), Base Language (BL), State Actions (SA), Override Start State (OS), Finish State (FS), Nested Machine (NM) and Concurrency (C). The state actions could be: entry (En), active (Ac) and/or exit (Ex) actions.

Considering the use of state machines in a robotics context, the Kouretes Statechart Editor (KSE) [27] is a visual tool that forms part of a model-driven process for robot behavior development. In it, state machines are graphically edited, optionally starting from a text-

based description, and the model-driven process then generates the executable code for these machines. There is however no visualization of execution of the state machine, only of the static machine, as presented in the editor of the program itself. KSE supports concurrent behaviors by allowing machines to execute multiple states at the same time.

XABSL [28] is a text-based approach to define nested state machines that features a variety of support tools. For example, it allows for the automatic creation of diagrams of the state machine, though does not include any simulation support or simulator integration. Even when it does not include a static visualization of the state machines, it compensates it with a run-time visualization of the machines and the arguments of the machines. XABSL supports its own DSL to build state machines (represented as *Prop.* in Table 3.1)

CABSL [29] born as an alternative to XABSL. Basically, CABSL has the same features of XABSL, however CABSL is written from the ground-up using C++ macros. This allows for a better integration with C++, using not only the datatypes supported by XABSL, but supporting any datatypes, API, extensions, etc. supported by C++.

Niemüller *et al.* [30] propose the implementation of behaviors in a general purpose scripting language, called *behaviour engine* (for Table 3.1, we called it Niemüller Behaviour Engine, NBE). It is however unclear from the text what the concrete syntax for state machine description is, nor what features the system supports. Their tools do provide support for visualization of the state machines, including showing the current state and last taken transitions as the program runs.

Another example of hierarchical state machine programs in robotics is Gostai Studio [31]. Gostai Studio is an IDE with visual programming properties that allows developers to build state machines using an editor, without the need for writing actual code to build the machine. States in Gostai Studio represent behaviors. Transitions are associated directly with their guard code. When developers click on a state they can code the behavior of the robot on that state, and when they click on a transition they can change the event that allows the change of behavior on the robot. Gostai Studio also shows in real time what is happening inside the robot by highlighting the current state of the machine and the triggered transition. To make changes in the program, developers should first stop the program and, when the changes of the program are made, the program is restarted.

Restricted Finiste State Machine (rFSM) [32] born as a minimal variant of Harel Statecharts, instead of extending state machines. It takes the concepts of Harel Statecharts and removes complex features that, arguably, are not necessary to develop robotic behaviors. This tool allows for nested machines to execute not only in the defined start state, but also in the special *history* pseudo state. When a nested machine is executed twice in the same execution, when this pseudo state is defined, the second time will start its executing in the last executed state. It does not support parallel states as expressed in Harel Statecharts, but supports executions of multiple machines at the same time.

The RMC advanced flow control (RAFCON) [33] presents as with another IDE to develop state machines. One of the most important contribution of this works is its own IDE to develop robotic behaviors using visual programming. This IDE also supports several debugging features, such as giving developers the option to execute the machine one step forward

or even one step backwards. For the latest case, states may define a special action called `execute_backwards`, which will be executed whenever developers decide to step back an execution. RAFCON supports concurrency similar to SMACH. It provides a special machine where, when executed, all states defined inside this machine will be executed at the same time.

Within ROS, there are more tools to build behavior-based robots. FlexBe [10] is based on SMACH, with extended features. In particular, FlexBe is an IDE to create complex behaviors using visual programming and already built states in a more interactive way, especially for non-developers who need to make adjustments to the behavior of robots. It also monitors the behavior at run-time and the behavior can be changed on-line too. Because it uses visual programming, it also provides a visualization of the states and transitions of the built machines. As SMACH, it also allows for concurrent behaviors using the same concept: a special machine will execute all its states when this machine is active.

Chapter 4

Meaningful Feedback: VizRob

Robotic behaviors are commonly implemented using large and complex pieces of software. Although the robotic community has produced efficient APIs and middlewares to *build* robotic systems, there is a still a need for adequate tools to comfortably *debug* those systems. A survey taken by RoboCup participants [34] shows that state machines are one of the most affected algorithms when the robot has problems.

Before answering the first research question defined in Section 1.4, we first need to understand the limitations of current practices when developing robotic behaviors. For this, we first reviewed prominent learning materials of ROS [17], the de-facto standard of robotic development. We then surveyed three robotic software engineers. This step is key to identifying patterns and tools commonly used in robotic software debugging. Subsequently, we employed these patterns to design a set of visualizations of ROS logs. We produced three visualizations, each showing a particular aspect of the logs, including logs' severity and frequency. These visualizations are packaged in a tool called *VizRob*. We also provided an implementation of VizRob for SMACH [6], a widely used framework to build robotic behaviors.

We received initial feedback of VizRob from six robotic software engineers in a small scale case study. Our results show that VizRob is key to solving complex debugging situations and it is positively perceived by the experiment participants. In particular, all of them acknowledge that our approach is relevant, highly promising and significantly reduces the effort to process logs compared to textual tools.

With VizRob and its preliminar use case study, we answer the first research question, we can know how the behavior of a program can give a meaningful feedback to developers in a nested state machine program for robotic behaviors.

This chapter first reviews learning material of ROS and SMACH and surveys three robotic software engineers. It then formulates a set of research questions considered for VizRob. Section 4.2 visits tools that present some kind of feedback to robotic behavior developers. Section 4.3 describes the visualizations embedded into VizRob, followed by Section 4.4 that presents design notes of VizRob. The small scale case study is detailed in Section 4.5, then we follow with results and threats to validity in Section 4.6 and Section 4.7 respectively.

Finally, Section 4.8 concludes this chapter.

4.1 Current Debugging Practices and Research Questions

To see what developers need to understand about their programs, we conducted an informal survey of debugging and logging techniques for robotic behavior. We reviewed learning and teaching material related to Robot Operating System (ROS, explained in Section 3.1) [17] and SMACH (explained in Section 3.2) [6] as well as surveyed three professional programmers (Section 4.1.1).

Results of these efforts are used to formulate a set of research questions (Section 4.1.2). These questions are considered to motivate and design VizRob as a way to improve the feedback of behavior-based robotics to developers.

4.1.1 Looking at common logging and debugging problems

To understand how robotic developers use logs as a feedback of what the program is doing, we followed the next steps:

1. We analyzed several robotic programs. All of them are programs used in tutorial materials of ROS and SMACH.
2. Once we collected information, we surveyed three robotic software engineers working in robotic behaviors.
3. We contrasted engineers' answers to what we found in the tutorial materials. We found recurrent situations related to debugging robotic behaviors using logs.

The analysis of ROS learning material and programs indicates that the most important logging practices are:

- Marking critical events on the robot: *e.g.,* low battery.
- Value of variables: *e.g.,* battery level, distances.
- Value of published and received variables: *e.g.,* text heard, pose of the robot.
- Marking part of the program: *e.g.,* starting to move.
- Marking part where the program failed: *e.g.,* the program did not receive a value, it did not plan a path.

These situations are recurrently employed as "typical situations" for which a logging facility should be employed.

The results of our analysis is similar as the results stated by Valdman [35] for generic log files. He stated that logs often show values of variables (*e.g.,* battery level) and that logs

present information that developers consider important (*e.g.,* low battery, marking part of the behavior of the robot).

We contrasted what we found in learning material with the survey of three software robotic engineers involved in robotic behavior from different robotic teams. We contrasted our result with the engineers' answers and found the following:

- The three engineers use the ROS logs system.
- They only use `print` when they debug. After finishing the debugging process, those `print` instructions are removed.
- They use the ROS logging facility in a similar fashion as provided in the ROS learning materials.
- Logs are the most common way to understand what the robot is doing, but it is not the only way. We found other ways to introspect the robot behavior: using the lights of the robot or making it speak to indicate some aspect of its behavior. Another way is by using the `ipdb` debugging tool and analyzing saved executions a posteriori.
- Logs are complemented with dedicated tools (`rqt_console`, `swri_console`, ANSI colors on terminal).

Participants emphasized that logs are helpful but rarely sufficient alone, and some errors are very difficult to reproduce. The tools *rqt_ console* [36] and *swri_ console* [37] are sophisticated, general purpose GUI to filter and navigate ROS logs.

The survey of Wienke and Wrede [9] shows developers also use special purpose visualization tools to monitor systems. They also state that developers mostly use `printf` or log file as debugging tools for robotic systems, while they sometimes use debuggers such as `gdb`.

All the information obtained in this section, including the data, the programs we analyzed and the survey is published for reproducibility [38].

4.1.2 Research Questions

Based on the debugging practices we observed, we designed the VizRob debugger for robotic behavior. We formulated a set of research questions we wish to answer:

- **Q1:** Can developers locate the causes of faulty behaviors?
- **Q2:** Can developers locate the value of the variables when the program is running? Can developers locate the possible variables that produce faulty behaviors?
- **Q3:** Is the program more error-prone in their critical parts?
- **Q4:** Can developers understand the causes of faulty behaviors?
- **Q5:** Is VizRob faster at locating faulty behaviors than using conventional debugging methods?
- **Q6:** Is VizRob faster at locating the value of the variables that produce a faulty behavior than using conventional methods?

- **Q7:** Is VizRob faster at locating the critical parts of the program than using conventional methods?
- **Q8:** Is VizRob faster at understanding faulty behaviors than using conventional methods?

Our first four research questions focus on understanding errors of faulty behaviors, while the last four focus on the time of understanding the errors.

We define the *critical parts* of a program as the part of the program where, in its execution, produces the most logs and expends most of the execution time. Also, in these parts the program produces logs of a level of severity that developers should be concerned about, such as warning and error logs.

For this first small scale case study, we do not expect to measure the time of our tool with a baseline, instead we are just going to give some qualitative insights for the design of a follow up experiment.

4.2 Current Tools to Debug Robotic Programs

There are several tools to give feedback to behavior-based robotic developers. However, all of them lack some capabilities that VizRob combine.

4.2.1 Log Listings

There are well known tools that list logs in a ROS application: *rqt_console* [36] and *swri_console* [37]. Both are sophisticated log listings for ROS systems. The *swri_console* has several log filters to focus on the important logs. However, both tools are for generic purposes, not being able to use the domain of robotic behaviors to provide more feedback.

4.2.2 Nested State Machine Visualizations

In Section 3.2 and Section 3.3 we mentioned SMACH [6] and FlexBe [10]. We also have LRP [16], a live programming language for behavior-based robots (more on LRP in Chapter 6). All these tools offer high level, run-time visualizations of state machines. However, these visualizations do not allow visualizing state machines produced by other tools. On the other hand, VizRob has an open API for developers to use other tools.

Moreover, the mentioned tools only visualize the machines without additional information. This reduces the feedback the tool gives to developers. VizRob integrates more than the machine visualization, *i.e.,* logs and execution time, providing more feedback than the previous tools.

4.2.3 Augmented Reality

Augmented Reality (AR) has been used to debug the data captured by the robot, comparing this data against the real world [39, 40]. AR puts the robot in the real world, and enhances the 'real world' with virtual data coming from the robot's sensors. This helps developers to understand if the captured data is precise.

While these tools analyze the data of the sensors, VizRob uses log information on the domain of robotic behaviors to show developers where in the program the behavior may be failing. Both techniques are complementary, and using both may offer even more feedback to developers.

4.3 VizRob

We use a graph metaphor to represent state machines and use polymetric views to represent metrics [41]. Section 4.3.1 presents the common features of all VizRob visualizations. Sections 4.3.2 – 4.3.5 describe our visualizations. Section 4.3.6 discusses the navigations between visualizations. VizRob is an artifact available online[1].

As far as we know, there are no available visualizations that help developers understand how logs are being produced in a nested state machine type of program. We used VizRob for that purpose.

The core of VizRob is independent of the robotic behavior API. VizRob is able to work with any API if a suitable bridge between VizRob and the API is built. Also, the API itself should follow the nested state machine paradigm. VizRob generates the visualizations automatically with the provided information passed by this bridge. VizRob is bridged with SMACH, a popular API to build robotic behaviors in ROS.

In this chapter we present the visualizations provided by VizRob. We have a routine of grabbing an object using an artificial robot arm as our running example throughout this chapter.

4.3.1 Common Features

VizRob presents three distinct visualizations with common threads. Each visualization represents a state machine with rectangular nodes that represent states and directed edges are their transitions. A green border around a state indicates the presence of a nested machine. The machine name is located at the top-left corner. The names of the state are displayed with a pop-up when the cursor is over one of the states. The polymetric information of each visualization is presented in a label.

[1] `http://mcamp.github.io/VizRob`. Accessed: 04-07-2019.

In all visualizations, states are represented as a box to which metrics are vertically and horizontally mapped: the *height* of a box represents the execution time (*i.e.,* the time of execution inside the state), and the *width* represents the number of logs (*i.e.,* the number of logs produced while executing this state).

An example of one of these visualizations for grabbing an object can be seen in Figure 4.1 (more information on this particular visualization is shown in Section 4.3.2). We see that the state machine contains three nested machines and a seemingly important state that has 5 incoming transitions.

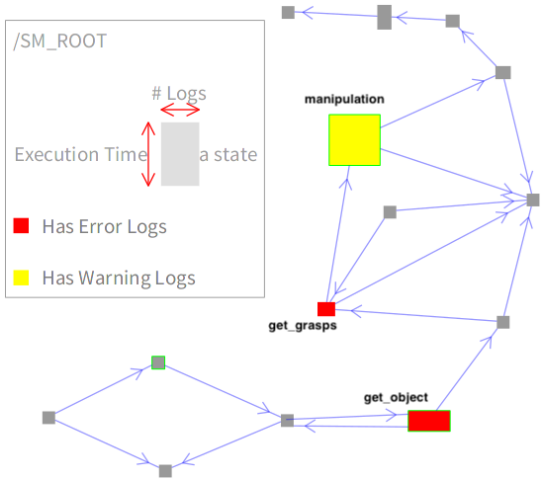

Figure 4.1: Type of Logs Visualization.

The transitions represent static relations: they are shown if they connect the executed states, even if the transition is not triggered. Every visualization therefore has the option of removing and showing the states that are not being executed during the robot execution. This is crucial for indicating the coverage of the robotic behavior during its execution.

In Figure 4.2 we see the same execution presented in Figure 4.1, but with the option to only show the coverage of the behavior execution. While in the whole visualization we identified a state that seems important because it has 5 incoming transitions, we can see that this state is not even executed.

4.3.2 Types of Logs

The *types of logs visualization* associates states to the severity of anomalies found in the logs. States are colored in yellow, red or dark gray. When there is at least one warning log in a state, this state is yellow. A state with at least one error log is red. When there are warning and error logs in the state, red has priority. Dark gray is the default state color, without any warning or error log produced by the state.

The visualization of our running example can be seen in Figure 4.1. In the visualization there are two prominent and large states. The large yellow state, named *manipulation* in our example, triggers at least one warning log and takes a significant amount of time to execute.

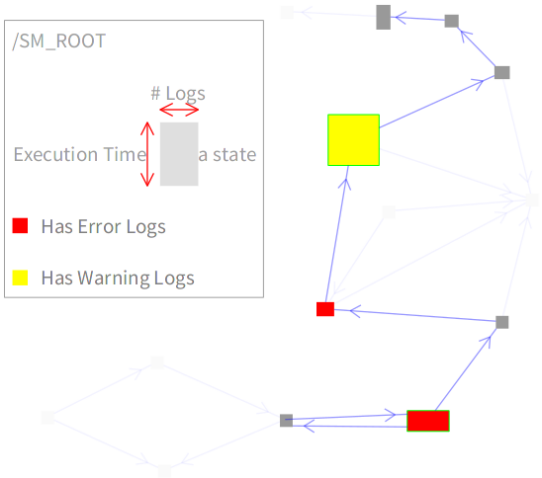

Figure 4.2: Figure 4.1 while turning on the coverage option. All the states that do not execute in this example disappear.

The wide red state, named *get_object* triggers at least one error log while it is relatively fast to execute, since it has a small height. This state has a green border, indicating that it contains a nested machine. Both states have a considerable amount of logs of any severity. The state *get_grasps* also produces error logs since, it is red, takes very little time to execute and does not produce a significant amount of logs, since it is very small.

4.3.3 Error Logs

It is relevant to highlight logging data indicating erroneous behavior. The *error logs visualization* uses a dedicated color mapping. The number of error logs emitted from the execution of a state is linearly mapped to a gray-to-red fading. In this visualization, a red box indicates the state that emitted the largest number of error logs, while a gray state is the state with the least number of error logs.

Figure 4.3 illustrates the error logs visualization. We see that *get_object* is not the state that triggers the most error logs, even if it triggers the highest number of logs of any severity. The small red square is named *get_grasps* and deserves to be carefully considered by the robotic software developers.

4.3.4 Frequency

Determining the number of times a state is executed is relevant when debugging. The *frequency visualization* represents the number of times a state is executed using a white-to-black fading.

In this visualization, a state is colored black if it has many executions and it is light-gray with very few executions. The number of executions of a state may vary from the time of the execution. For example, while a state can be executed for a long time, but only once,

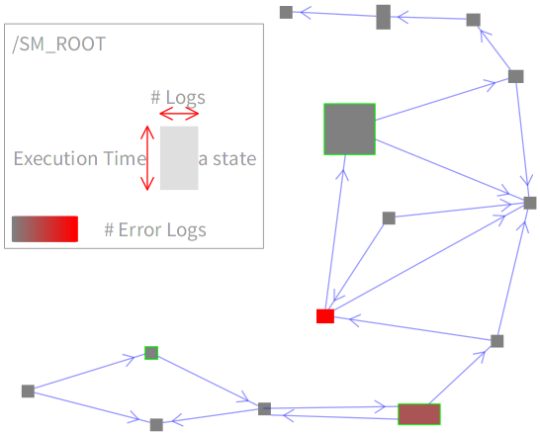

Figure 4.3: Error Logs Visualization.

another state can be executed a short time, but many times. Figure 4.4 has an example of this visualization.

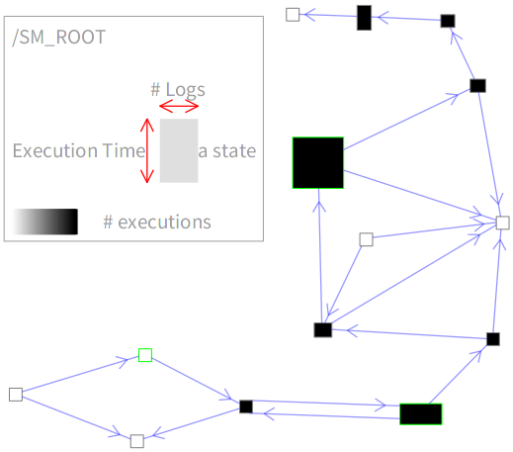

Figure 4.4: Frequency Visualization.

This visualization clearly distinguishes the states that are executed from the ones that are not executed. The white ones are not executed. The black ones are executed the same number of times. Note that in this execution, there are no gray states that are executed an intermediary number of times.

4.3.5 Logs Listing

Each of the previous visualizations are interactive. Clicking on a state lists all the logs associated with the state (Figure 4.5). Moreover, the exact location in the source file that triggered the log is also given. This list of logs offers the classical navigation and filtering options found in common debugging tools. The listing provides two filters: one for the severity and other for the name of the program where the log is produced[2]. Also, an icon appears on the left of the text of the log representing the severity of the log.

[2]In the case of ROS, the name of the program is the name of the Node producing the log

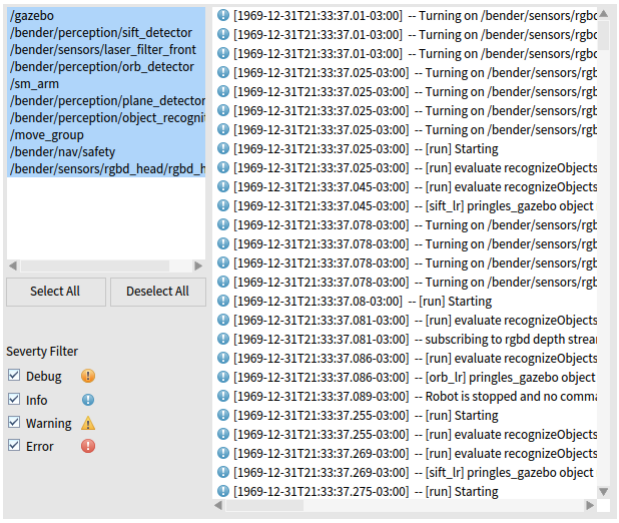

Figure 4.5: Logs Listing. At the left we see the filter options. At the right we see the list of all the logs produced.

4.3.6 Navigation

Detail of the represented elements and log information in all visual elements may be obtained on demand. Clicking on a state opens new visualizations, leading to a chain of visualizations: the previous visualization remains accessible and interactive.

Visualizations are recursive as exemplified in Figure 4.6. Clicking on a state with a nested machine reveals the nested structure, with the option to use any of the previous visualizations. When a log is clicked, it is possible to see the attributes of the logs, such as the name of the program, severity, message. In Figure 4.6, left side of the visualization we have the root machine. The right side of the visualization represents the nested machine of the state represented with the big yellow square of the left visualization, the *MANIPULATION* state.

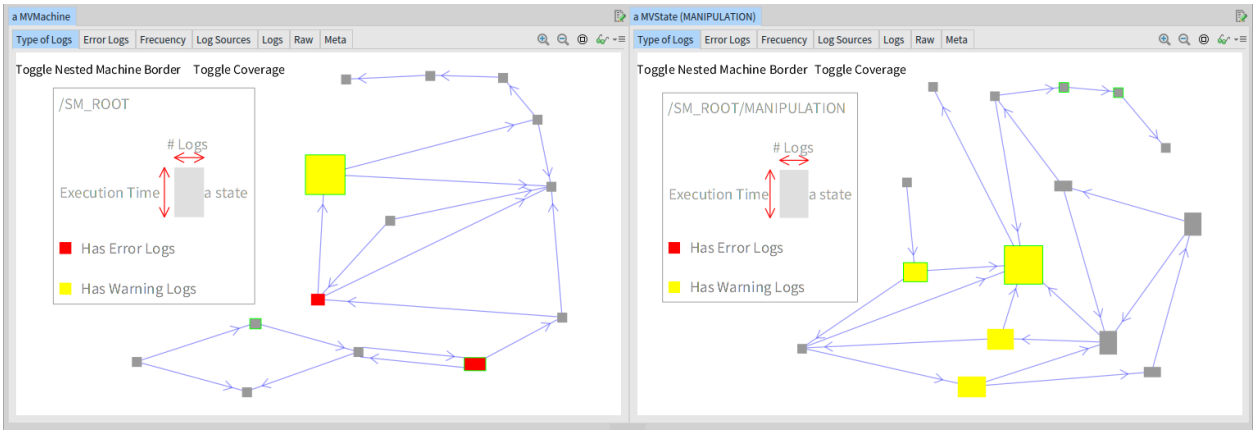

Figure 4.6: Navigation of the Visualizations.

4.4 VizRob Design Notes

VizRob's core is designed around the concept of nested state machines, this allows VizRob to be independent of the robotic behavior APIs. It shows run-time information of a nested machine program: logs and execution time. It is important to simulate an execution of the behavior of the program in VizRob to get the right data to show in the visualizations.

VizRob generates its own machines, states and transitions by taking into account the information of the robotic behavior API. A *Machine* object contains all the *State* and *Transition* objects of that machine. For nested state machines, a state may have a unique machine instance.

Because of the run-time nature of VizRob, one of the most important aspects is how it saves the run-time information. VizRob saves the state executing on the machine, *i.e.,* the current state. Whenever a state is executing, VizRob creates a single *Status*. A Status is the representation of the run-time information of a state. Here, VizRob saves all the logs that are produced in that point of the execution. VizRob also saves the time when the state stops being active. With this, VizRob gets the execution time of that status.

Moreover, a state can be active any number of times in the same execution of the program (*i.e.,* with a loop of states). For this reason, every state has a collection of statuses. The sum time of all status times is the total execution time that is shown in the parametric views of VizRob (the parametric views explained in Section 4.3). The number of executions visualized in the frequency visualization (Section 4.3.4) is the amount of statuses that a state has.

VizRob allows for a machine to have more than one active state at the same time, this allows for concurrent states on the execution of a machine. In this case, the logs produced by the program are assigned to all the active states. We cannot be sure in which particular state the log is being produced.

For a machine, it groups the information already presented in states, for example, the logs of a machine are all the logs of all the states of that machine.

VizRob can be fed with data automatically using a bridge that connects it with the desired robotic behavior API. The API should offer a way to expose relevant information about the static and dynamic configuration of the state machine. In particular, we designed a bridge that connects VizRob with SMACH [6], a popular ROS API to develop robotic behaviors. By using an extra interface shown in Figure 4.7, VizRob connects with the SMACH program, receiving the data while the program is running. Developers need to give the name of the three communication channels for VizRob to work with SMACH: structure, status update and logs. With these channels, VizRob collects the static structure of the machine, the current states of the running machine and the logs of the program. When developers click the start button of the interface, VizRob starts collecting data of the SMACH program. This happens while the SMACH program is running, allowing for assigning the logs of the program to the running states and setting the execution time and the number of executions of the states. This data is then processed by VizRob to automatically build the visualizations after the program stops. The program needs to stop for VizRob to collect the final information of the

programs (*i.e.,* the execution time of the last active states). When stopping the recollection of data, the open button becomes available. Clicking on the open button will make the visualizations of VizRob available.

VizRob only shows one execution of a behavior at a time. If VizRob is not stopped at the end of the behavior, it may clash with the information of a new behavior. For VizRob SMACH, to allow the recollection of data for a new behavior, a new connection with SMACH should be created, using a new instance of the VizRob SMACH UI.

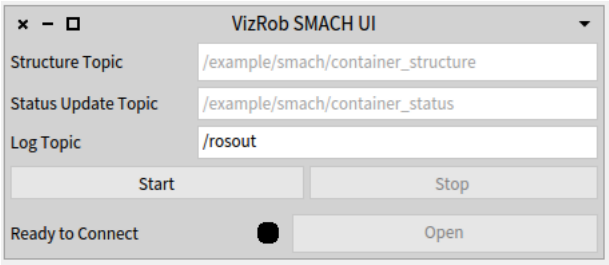

Figure 4.7: SMACH Bridge UI for VizRob.

4.5 Initial Feedback: Case Study

We received initial feedback of VizRob through a small scale case study involving six robotic engineers (who we refer to as *participants* in the remainder of this chapter). The participants are members of the UChile Homebreakers team[3]. All the participants of this team have extensive experience in participating at the RoboCup competition (RoboCup@Home league) and all are experienced with nested state machines. The team built and designed the software aspects of two robots: *Bender*[4] and *Pepper*[5].

We also have as a participant a former member of the UChile Robotic Team. This participant has an extensive experience in participating at the RoboCup competition, in the RoboCupSoccer league. The participant is an expert in building behavior and handing all software aspects of the NAO robot[6].

For the sake of reproducibility, the data produced in this case study is available online [38]. It includes videos, audios and the answers from the pre and post questionnaire.

4.5.1 Robotic Behaviors

We visited the UChile Homebreakers team on site and the participants agreed to use VizRob to debug robotic executions on 2 different robotic behaviors. We asked the participants to use

[3] `http://robotica-uchile.amtc.cl/about.html`. Accessed: 04-07-2019.
[4] `http://robotica-uchile.amtc.cl/bender-robocup.html`. Accessed: 04-07-2019.
[5] `http://robotica-uchile.amtc.cl/pepper-robocup.html`. Accessed: 04-07-2019.
[6] `https://www.softbankrobotics.com/emea/en/nao`. Accessed: 04-07-2019.

VizRob on faulty behaviors with difficult-to-debug anomalies, but also in seemingly correct behaviors. An apparently correct behavior may still have errors during its execution.

The first behavior is *Speech and People Recognition (SPR)*, which consists of 3 routines: categorizing people in a crowd (*e.g.,* determining gender and age); then, an operator asks questions about the crowd; finally, the robot is surrounded by people and they take turns asking the robot questions. The second behavior is *Grasping*. In this task, Bender positions itself in front of a table. It recognizes an object located on the table and grabs it. In our case study, participants employed VizRob on 3 different executions of these 2 different behaviors.

4.5.2 Problems found

In this case study, all participants should find problems of a particular robotic behavior in 3 executions of the behavior. We label the participants according the behavior they debug and the team they belong to. For the behavior we use S for SPR and G for Grasping. For the team we use H for UChile Homebreakers team and R for UChile Robotic team.

For the SPR behavior we had 3 participants labeled SH1, SH2 and SH3. Two of them have knowledgeable experience using SMACH and one of them has advanced experience (within the scale: None, Beginner, Knowledgeable, Advanced and Expert). They are all undergraduate students with experience ranging from 1 to 3 years using SMACH.

For the Grasping behavior we had 3 participants labeled GH1, GH2 and GR3. GH1 and GH2 considered themselves as having knowledgeable experience using SMACH (within the same scale as before), while GR3 does not have experience in SMACH. However, GR3 is an advanced robotic developer with more than 5 years of experience and 3 to 5 years of experience using ROS. GH2 is a Master student and GR3 is a professional with a Master degree.

Depending on the execution of the behavior, participants found different types of problems in the executions, thanks to VizRob. We list some of the techniques using VizRob that the participants employed to find these problems:

Error logs are important: The red states on the visualization are important and show problems for all the five executions that they appeared.

Warning logs (in big states) may be important: There are several problems that became apparent to participants via the yellow color on states for warning logs, especially when there were a lot of logs produced on that state. This is shown in the Type of Logs visualization with big yellow squares and they were relevant in the 2 executions where they appeared. The problems that became apparent at least need to be checked to see if they are important or not, as stated by GR3.

Similar states group errors: In one of the executions of the grasping behavior, GH2 notices a problem that was visible, thanks to a group of yellow states that were connected to each other. This group of yellow states shows several warning logs that are connected within

the same problem.

Frequency visualization group several states: Connected with the previous problem, after finding all the yellow states, changing to the frequency visualization, GH2 notices that the errors not only were connected, but happened multiple times. The frequency visualization shows states with a lot of executions, indicating that the robot tried several times to grab an object, but finally gave up.

Problems with variables: In 2 of 3 executions of the SPR behavior, there were problems of not filling or wrongly filling variables for the SMACH API. This is found by the three participants of the SPR behavior, thanks to a red square presented in the type of logs visualization. In one of the executions of SPR, SH3 believes this error is responsible for making the robot incorrectly answer questions, even when the execution of the routine to answer questions is several states beyond the state where the error appeared.

Seemingly correct behaviors can have problems: One of grasping behavior executions performed correctly, resulting in the robot grabbing the object. However, the three participants notice several problems that may affect the robot in the future. These problems are apparent in the type of logs visualizations with yellow and red states.

4.5.3 VizRob vs participants' debugging tool

In robots like these, it is common to have several open terminals that execute different aspects of the robot. All participants stated they only use the terminal where they executed the behaviors to debug and not the other terminals that executed other aspects of the robot. They look at the logs in that terminal to see what is happening.

When the error of a behavior crashes the system, it is easy for the participants to find it because, usually, the last log of the terminal points to the crash. However, SH3 mentioned that VizRob helps to better localize the cause of the problem, because an error may be triggered way before it manifests in the execution. With VizRob, the participant can see that the program seemed fine until the error is triggered, reducing the debugging time for these kinds of errors. SH1's conclusion is similar: any other tool to debug is for generic purposes and they spend too much time looking for the right information.

The participants believe they can find almost all the errors they found with VizRob by using the terminal. However, all of them believe it is difficult and time consuming using the terminal compared with our tool. Within the terminal it is difficult to find the right log because of the information overload in it. These errors can be found by tools like *rqt_console*. However, with this tool they still may have a problem locating the root of the error if they do not see where the error is in the execution, as we previously mentioned.

Some errors can be found just by being present in the task at hand. If the robot is incorrectly answering a question, then it is easier to hear that the robot is performing poorly instead of looking at logs in any kind of sophisticated tool. However, with VizRob SH3 found the cause of why the robot was answering wrongly. The participant believes he could not

reach the same conclusion by solely using the terminal.

In seemingly correct behaviors, participants stated they mostly ignore the problems of the behaviors because they believed the robot behaves correctly. As we mentioned in the previous section, VizRob helps developers find problems even in seemingly correct behaviors.

4.5.4 Use of VizRob

Five of the six participants first used the coverage option to find what the last state of the execution of the behavior was, believing they would find the error there.

After finding the last executed state, all the participants used the Type of Logs and the Error Logs visualizations to have a general view of the problem and click on the state that may have a problem. GH1 told us that these visualizations help him see if the machine is having other problems or if the current error stems from an earlier problem in the execution of the machine.

Three of the participants found VizRob extremely useful (on a scale of "not useful", "somewhat useful", "useful", "extremely useful"), while the other three found the tool useful. All of them wanted to adopt VizRob on the debugging of robotic behaviors. Two of them would use VizRob to solve issues in complex robotic behaviors.

GH1 told us that many events occurred while executing a robotic behavior. However, most of these events are not noticeable in the same terminal where the behavior of the robot run. For example, it is not possible to see low-level hardware logs in the terminal of the behavior of the robot. VizRob is perceived to be a significant change in that respect.

Moreover, robots are known to have problems of uncertainty and these problems may affect the robot in future executions of the same behavior, as stated by GR3. GH2 also stated that if there are problems that do not affect the execution of the machine in an isolate context, it may affect the execution of more complex behaviors when using this machine with others. Both participants agree that VizRob may reduce these issues because, with VizRob, they found problems even in a seemingly correct execution.

4.6 Answering our Research Questions

In Section 4.1.2 we listed eight research questions. Taking into account the previous experiment, we answer the questions:

Q1: Can developers locate the causes of faulty behaviors? Positive answer. The participants found several problems in all the executions, even the execution that work seemingly fine. VizRob helps them find faulty behaviors.

Q2: Can developers locate the value of the variables when the program is run-

ning? Can developers locate the possible variables that produce faulty behaviors? Negative answer. VizRob does not help developers in finding the value of variables and how they change in time, nor in correct and faulty behaviors. Our best guess is that in the executions at hand, the developers do not use or only use a small number of logs to show the value of the variables.

However, participants still see the value of some variables that were useful. Nevertheless, these kinds of logs do not seem prominent in executions of the robot in the case study.

Q3: Is the program more error-prone in their critical parts? No answer. Even when the participants found more errors in the critical parts of the robot, we are not sure if in these parts there are more errors than other parts. Nevertheless, we show that critical parts are important places in the behavior of the robot and they need to be analyzed.

Q4: Can developers understand the causes of faulty behaviors? Positive answer. The participants not only found the place where the faulty parts of the behaviors were, but also they gave several insights about why the robot may be failing. One participant stated this when he told us that the tool helped them to better pin down the possible causes of the faulty behavior.

Q5, Q6, Q7, Q8: Is VizRob faster? Positive answer, but only based on participant perception. In all the questions related to time, all participants considered this tool much faster to find bugs and therefore, to fix them rather than use their conventional methods. Though this is an important statement, it cannot be verified without a rigorous comparison.

4.7 Threats to Validity

Here we present the most important threats to validity for our case study.

Ready to use VizRob. All participants found VizRob more comfortable to use rather than using their normal way of debugging. However, one of them explicitly told us that if VizRob is not as easy to install as if it is to use, he may not find it as comfortable as he stated.

Nevertheless, we do not want to measure that part of the development (even as important as it is), we only want to measure the effectiveness of VizRob to find errors in the behavior. With that in mind, the participants found a significant number of errors using VizRob and they stated that VizRob helps them find the errors in an accurate and faster fashion.

Lack of comparable baseline. Because we conducted a small scale case study, we do not have a satisfactory baseline to compare to VizRob. The participants do not use sophisticated tools to debug programs, they only use the terminal. We therefore expect better results using VizRob.

Nevertheless, our case study is not about comparing VizRob against the terminal, or any

other debugging tool for now. In this work we want to show how participants behave when using VizRob. Also, we want to show that the participants can find errors using VizRob without a long previous training.

4.8 Conclusions

In this chapter we presented VizRob, a debugging tool for robotic behaviors designed to give meaningful feedback of the behavior of robots to developers. This tool incorporates metrics based on logs and execution time into the view of nested state machines - a classical paradigm to represent robotic behaviors. Developers can navigate into the machines and states to see focused information of that particular element. Developers can look at the overall information or immerse into the behavior and focus their attention on a particular part of the system.

VizRob is fed on the fly by developers. In particular, it can be fed by running programs with a proper bridge between the program and VizRob. Using our bridge between SMACH and VizRob, developers can feed VizRob with behavior-based programs using SMACH.

We received initial feedback on VizRob using a small scale case study among six software engineers from the UChile Homebreakers team and the UChile Robotics team. Our preliminary findings show that participants find several errors in robotic behavior programs and they suggest solutions for these errors, even without looking at the source code of the programs. VizRob is positively perceived by the participants, in particular, all of them want to adopt this tool in the future when they program robotic behaviors.

However, VizRob intensively uses the logs of the system. If the logs of the system are insufficient, VizRob may be insufficient for developers too. Nevertheless, VizRob still uses information of execution time of the program, showing developers where programs spend most of the time. This may be useful for developers, even for them to notice where the program needs more logs. This is a compelling approach and worthy of future work for our tool. Maybe VizRob helps developers understand where to use more logs and what kinds of logs.

With our preliminary findings we need to conduct more experiments to measure the real value of the tool. In this work we have positive findings based on the qualitative information that participants gave us. It is important to measure the value of VizRob in an experiment with quantitative value.

Nevertheless, VizRob answers our first research question defined in Section 1.4: VizRob shows how a behavior of a nested state machine program for robotic behaviors can give meaningful feedback to developers. Even with several tools and visualizations that give feedback, they all lack the capabilities of VizRob. VizRob gives more, aggregate feedback using information that is almost always used contextually, like logs and execution time.

Chapter 5

Live Programming

Live Programming has gained attention thanks to the widely commented talk by Bret Victor at CUSEC'12 [7]. Its origins can however be traced back to the early work of Tanimoto on VIVA [8].

Live programming is now a popular concept in several kinds of programming languages and applications. While they all declare to be 'live', they all use the concept of live programming indistinctly though they may use and/or achieve totally different ways of liveness.

This dissertation specifically focuses on Live Robot Programming (LRP), our own live programming language. A full introduction of LRP can be found in Chapter 6. Nevertheless, before talking about LRP, we considered it necessary to understand the bases and the depth of live programming to understand how LRP and others' live programming languages achieve liveness and how much 'liveness' they achieve. It is also important to concretely define liveness and what it means that LRP is 'live'.

We characterize live programming in this chapter, basing it on Taminoto's first [8], his revisited definition [42] and also on Hancock's definition [12].

5.1 Overview

In a nutshell, live programming postulates that developers benefit from an immediate connection with the program that they are making. Languages for live programming therefore provide for an immediate, 'live' and meaningful feedback on program behavior. With this feedback, developers are expected to build accurate mental models of the system when its execution is observed. This permits, on the one hand, for extremely rapid creation of program behavior as the effects of variations in the behavior are immediately visible. On the other hand, it immediately reveals bugs that are due to the programmers' mental model of the executing program differing from the actual behavior.

To put this in context, we postulate an example of live programming in a classical example, a program that makes the logo turtle move in a square circuit using a 'turtle' robot. The turtle robot does not know how much it has advanced and how much does it has turned. The turtle robot can only move forward, backwards or turn at a certain speed for a certain time. This is similar to how classical robots work.

First, we want to move the robot 1 meter forward. To do that we make the robot move forward with a certain speed for a certain time. We check how much the robot has moved. Let us say the robot does not reach 1 meter, then we can increase the speed of the robot or the time the robot move. In this example, let us increase the speed. The robot passes the 1 meter mark and now, we decrease the time the robot move. We do this a couple of times until we reach the 1 meter mark. For every iteration, we have to change the speed or the time in our source code, upload that to the robot and, finally, wait for the robot to move.

If we use live programming, the development time of the turtle robot decreases quite a bit. We still have to iterate adjusting the speed and/or the time of movement. However, we do not have to upload this to the robot itself every time. When we make a change in the source code, this change is reflected right away in the robot. Then, we make the change and wait for the robot to move. In the first iteration, if the robot does not reach the 1 meter mark, we can even increment the time right away and the robot will move right away. Using live programming, it may even reduce the number of iterations of the development cycle.

Continuing with the example, now we want to make the turtle robot turn 90 degrees. This is exactly the same example as the 1 meter mark, but using the rotation speed instead of the linear speed of the robot. Then again, with live programming we can decrease the development time of the turning behavior.

In summary, using live programming, we can decrease a lot the development of the behavior of the turtle robot by decreasing the development time of every simple behavior that composes the overall behavior of the robot.

The decrease in development time is due by decreasing or removing the *cognitive distance* between developers and the resulting program. Cognitive distance is the number of steps between thinking about a problem and finally seeing its solution in a program. We can see this in Figure 5.1. Moreover, whenever the program behaves incorrectly -*i.e.,* a bug in the program or a problem with the solution-, this process is repeated, increasing the cognitive distance even further. When this happens, developers need to mentally map the error of the robot into their way of thinking and the source code.

Figure 5.1: Cognitive distance in software development.

Live programming postulates to improve development by removing this cognitive distance. This is done through having a constant connection with the program, while the program gives constant and meaningful feedback to developers.

5.2 Characterization

Live programming has been used indistinctly to define several techniques where the only constant is to have a program that be modified while it is running.

In this work, we focus on two categorizations of live programming: levels of liveness [8] and immediate and meaningful feedback [12]. Both of them are important for understanding what the live programming languages really mean when they say they are 'live'. We also look at other definitions of live programming while, not being as deep as the previous two, are important to notice.

5.2.1 Level of Liveness

Live programming was first introduced by Tanimoto in VIVA [8]. VIVA is a visual language for image processing where programs are graphically represented as electronic circuits. Tanimoto defined different level of liveness for programs, classified according the degree of live feedback that is present. These 4 different levels of liveness are:

- **Level 1 - Informative:** A supplemental visual representation of programs is given, which is only used by developers to understand them.
- **Level 2 - Informative and significant:** The visual representation of the program (of Level 1) has enough information to be executable.
- **Level 3 - Informative, significant and responsive:** Programs wait until developers trigger computation directly, such as interaction with buttons. The execution environment will then attempt to execute these updates.
- **Level 4 - Informative, significant, responsive and live:** Programs are continually active; their behaviors are modified immediately when changed by the developers.

In a level 4 live programming language the amount and rate of feedback to the programmer is maximized [8]. Hence, in this work, when we talk about live programming, we partly refer to level 4 of liveness as defined by Tanimoto.

At the moment Tanimoto defined these levels, he assumed only visual languages for live programming. However, over time numerous work were published about liveness in programming languages, which is no longer restricted to visual languages, such as Flogo II [12] or Swift [43]. Moreover, the live programming concept has been expanded over the years to include many more programming languages, also considering level 3 of Tanimoto's definition to be live programming, *e.g.,* Self [44].

Then, several years layer, Tanimoto revisited his definition of liveness, including 2 new levels [42]. These new levels are designed to integrate more intelligence into the programming languages. The new levels proposed by Tanimoto are:

- **Level 5 - Tactically Predictive:** The program itself plans ahead slightly, predicting the next programmer action.
- **Level 6 - Strategically Predictive:** It is basically level 5, but the program can create itself new behaviors that developers intend to program.

5.2.2 Immediate and Meaningful Feedback

While it is true that the amount and rate of feedback is maximized in Tanimoto's level 4 liveness, for Hancock it is not enough for a live programming environment to have only continuous feedback [12]. Hancock claims there are two more things to consider: the feedback needs to be meaningful and there should be a continuous action over the continuous feedback.

The first issue can be seen in the next example: in a Java program there are lines of code that do not make sense without taking into account the context of the program. The meaning of liveness depends on how programs are represented. For example, Hancock presents us with a question: what does a line of code like `a = a + 1` mean? This line does not make much sense in isolation. In other words, even when a system is live, the code is not, the meaning of liveness depends on how the program is presented [12].

Hancock also presents us an example for the second issue of continuous feedback. Continuous feedback is meaningless when there is no continuous action over that feedback. It is much easier to aim a hose rather than a bow and arrow because the watering hose gives continuous feedback. We can easily adjust our aim because we have a continuous stream of water (information) coming out the hose. With the hose the only movement we need to make a correction is to aim, while with the bow and arrow we need to take a new arrow, aim again using the previous information and shoot. For the water hose, there is no need to "reload" the hose, while the archer should reload every time to shoot. This is important because it allows us to have a continuous aim with the hose, in other words, with a continuous feedback we can manage to aim at our target at all times because we have continuous action [12].

To formalize these two issues, Hancock proposes the concept of *steady frame*. A steady frame is a two-part concept where:

- **framing part:** The relevant variables can be seen and/or manipulated
- **steady part:** The variables are constantly present and meaningful

Part of the work of Hancock is the live programming language for robots Flogo I [12], which uses continuous feedback and steady frames. Flogo I is a live dataflow programming language that focuses on improving teaching of programming to children by using Lego Mindstorms [18] robots. Flogo I supports steady frames at two levels:

1. **Program Execution:** Developers can see input and derived values changing in the

program. These values are presented in the data flow visualization itself. Developers can experiment with these values, for example, moving the robot to see a blue color instead of a red one and then observe the effect in the program.

2. **Program Development and Testing:** Because of the immediate response to edits of the system, developers tend to write smaller parts of the program, see if it works correctly, then continue with other parts while finishing their code as they test it at the same time. This is possible because of the live nature of Flogo I, specifically, the continuous feedback the program gives to developers via its visualization.

Steady frames facilitates developers when taking continuous actions over a continuous feedback loop on their programs. In this dissertation, we do not only consider live programming to be level 4 of Tanimoto, but also about continuous actions to build programs and meaningful feedback to expose these actions.

5.2.3 Related Concepts

Swift *et al.* in their work on Visual Code Annotations [45] group concepts related to live programming of different works and classified them by:

- **Just-in-time (JIT):** Implementation of algorithms when the user is trying to accomplish the task. This is the broadest definition of the list.
- **Livecoding:** Artist-programmers performing an audiovisual show while developing the performance live in an audiovisual system.
- **Live Programming:** Direct construction, manipulation and visualization of programs at run-time.
- **Cyberphysical Programming:** Extending the concept of livecoding to any real-time domain with real-world interaction.

These concepts and the level of liveness of Tanimoto presented before are orthogonal in nature. For example, while Swift *et al.* classify Self into Live Programming, we consider that Self is at level 3 of liveness.

Cyberphysical programming also can be classified as Tanimoto's live programming. For example, if the live program is connected to a robot, it can also be classified in any of the level of liveness of Tanimoto and not to only be cyberphysical.

5.3 Live Programming in Robotics

Tackling live programming in robotics entails its own difficulties. Here we divided these difficulties between immediate connection and meaningful feedback.

5.3.1 Immediate Connection

When using live programming for robotic development, the programs should be connected to the robots. For this, it is not enough to have tools that connect to the robot only when the program is loaded to the robot itself. There should be a way to access the features of the robot, modifying the behavior of the robot itself, while the robot is behaving at the same time. For a complete solution of a robotic live programming language, this connection should not be limited to only one robot but for multiple robotic systems.

Another interesting feature of this connection is to reconstruct the same environment the robot is accessing. For some behaviors, this is not important. For example, when building a behavior that makes the robot move to a certain point in the environment, it is not important where the robot starts, it is only important where the robot should go. But for other examples, this becomes a difficult problem. An example of this is a pick-and-place routine. Imagine that the robot grabs an object and it moves the object to another location, then we change the program using liveness features, now we want the robot to do something with the object before reaching the desired location. But while we change the program, the robot drops the object. For the new program to work correctly, it assumes that the robot is still moving the object to the desired location. In this case, we could put the object manually in the gripper of the robot, or we could put the object in its original position and, restart the program for the robot to grab the object again. These kind of problems are not unique to live programming, but they also exist in other contexts, for example, if we want to automatically test the behavior of the robot in the real world (*i.e.,* how to test automatically independent behaviors of the robot, while the robot interacts with the real world).

If we use a simulation, in theory, we could solve this problem programmatically. We can save any moment of the simulation and attach these moments to parts of the program. When developers then want to restart from a certain point of the execution or jump to another part of the execution, the program may use the saved moment of the environment to run the new execution.

5.3.2 Meaningful Feedback

Another important aspect of programming language in robotics is how the robot itself should give feedback to developers. The robot possesses several actuators and sensors. The actuators change the configuration of the robot in the environment, by moving the robot itself or by moving certain parts of the robot (*e.g.,* a robotic arm).

The sensors show how the robot understand the real world. By making the robot move in the environment, or by changing the environment, the data of the sensors changes too. Moreover, most of the time the raw data of the sensors is unintelligible.

Robotic engineers already tackle this problem, they understand how important is to quickly see what the robot is sensing, and see how the actual configuration of the robot is being translated in their programs. For example, RVIZ, the standard 3D robotic visualization integrated with ROS, helps developers with this problem [46].

If we understand the configuration and the sensor of the robots as variables of the robotic program, with this visualization we can have the *steady part* of the *steady frame* properties mentioned in Section 5.2.2. If we have a way to modify the configuration of the robot manually in the real world (*e.g.,* moving a robotic arm manually), we also have the *framing part* in the configuration of the robot. However, we miss the *framing part* in the information of the sensors, as we can not change easily the information of the sensors of the robot.

In the case of behavior-based robotics, we encounter even more challenges. When we use nested state machines as a model to develop robotic behaviors, we have several ways to achieve meaningful feedback. We shown one in the form of VizRob in Chapter 4. However, with this we only have the *steady part* component. To achieve the *framing part* of *steady frame* we need to achieve also immediate connection, to change the relevant variables of the nested state machine program. The variables of a nested state machine programs include (but are not limited): the nested state machine program itself (machine, states, transitions, etc) and the variables of the program (as the classical variables used in any common programming language to represent values of the program). These parts should be changeable while the program is running, without the need of restarting the program altogether (framing part); and should be constantly present and meaningful (steady part).

5.4 Live Programming Languages

In this section we summarize several programming languages with some kind of liveness features. First we detail live programming languages outside the world of state machines and robotics behavior. We then describe live programming languages in the context of robotics and behavior-based robots.

5.4.1 General Live Programming Languages

Live programming was first proposed by Tanimoto [8], with the goal of providing maximum feedback to the programmer while a program is being constructed. That work states that "A live system begins the active feedback at editing time, and then continues it through the remainder of the session or until explicitly disabled by the user". The language presented in that work is VIVA, a visual programming language for image manipulation. Another, well-known, example of a live visual programming language is VVVV [47], where code is edited by connecting dataflow patches.

Outside of visual programming, the SuperGlue language [48] is a textual language that is also based on dataflow programming and extended with object-oriented constructs. McDirmid has other work on live programming, notably YinYang [49]. YinYang focuses on debugging, combining debugging with editing, then making changes while debugging that also change program execution dynamically. Explicit data flow eases live programming in that there is a clear order in the tree of computation that can be used for triggering required re-computation.

Burckhard *et al.* add live programming features for UI construction to an existing live

programming language [50]. The separation of UI state allows the UI to be reconstructed from scratch on each code change.

The work of Victor [7] showcases various examples of live programming in Javascript that produce pictures, animations and games. This work can be credited for sparking wide-scale interest in live programming, which arguably helped the crowdfunding of Light Table [51]: an editor that adds live programming features to a number of general-purpose languages.

Apple released Swift [43], a multi-purpose programming language with live features in 2014. Swift became the new standard to program software for iOS and OS X. Swift also presents a visual representation of the execution of its programs into a tool called *Playground*.

5.4.2 Robotics and State Machines Live Programming Languages

We offered a glimpse of related work in robotic behaviors in Section 3.3. However, most of the related works given there do not work with liveness.

Nevertheless, one of the behavior-based tools mentioned before, Gostai Studio [31], has several features that can be compared with live programming. For example, Gostai Studio also shows in real time what is happening inside the robot by highlighting the current state of the machine and the triggered transition. However, to make changes in the program, developers should first stop the program and, when the changes of the program are made, the program is restarted, contrary to the capabilities of what a live programming language should have. Gostai Studio likely builds behaviors for robots that are compatible with its own specific architecture, not allowing behaviors to be expressed to any other kind of robotic system.

InterState [13] is a live programming language that uses state machines as its computational model. But InterState is used to program user interfaces like Web interfaces using JavaScript, and not used in a robotics context. In InterState, state machines and constraints are a fundamental part of the language and it also provides behavior reuse by state machine inheritance. In experiments, InterState proved to be more efficient for writing user interface behaviors than using JavaScript.

We are aware of three other live programming languages used in robotics. First, Lim [11] extended a live programming music synthesis software with robot programming features. This language is a standard imperative programming language, and not specifically built to develop robotic behaviors. The language was used to program a robotic arm. The author remarks on how fast it was to implement prototypes of robotic actions using an exploratory approach. However, Lim also noticed some drawbacks of using a live programming approach, for example, a careless editing of a program could make the robot move inadvertently, possible damaging the robot itself or the environment.

Secondly, Hancock, in his Ph.D. thesis, proposes a live programming language for robotics using a dataflow representation, Flogo I, and a textual based language, Flogo II [12]. Flogo I uses a dataflow representation and the metaphor of circuits, such as VIVA, to build behaviors:

receiving data in circuits and passing the results to other circuits, finally reaching circuits that represent motors. Flogo II is a textual live programming language whose goal is to give live programming features to a textual programming language. While the dataflow representation of Flogo I is similar to nested state machines programs, Flogo II differs from this representation. However, both execution models, dataflow and nested state machines are still different. Dataflow representation uses a continuous stream of data that goes into circuits that can represent motors, change data, and is used to codify behaviors. On the contrary, nested state machine representation uses events and transitions to change the state of the machine that represents the behavior of the program. Lastly, the focus of Hancock's research is to improve teaching of programming to people, especially children. He tested his languages with kids to see if they understood how to program better, using robots as a motivation and live programming as an important feature. The work of Hancock does not entail a complete solution of using live programming for behavior-based robotics for multiple robotic systems.

Thirdly, the authors of RAFCON [33], first presented in Section 3.3, do not mention live programming within their work, however, RAFCON allows for modifications of the structure of the machine and the action of the states at runtime, seemingly enabling live programming. However, there is no specification on how they achieve live programming and how they deal with program changes. Moreover, even when RAFCON is independent of the robotic system, there is no specification on how to use it in a robotic system or how to extend RAFCON for other robotic systems.

5.5 Conclusions

Summarizing, there are different interpretations of what 'liveness' is in a programing language/environment, but the initial goal is the same: maximizing the feedback of a program for developers, reducing the time and effort of coding a correct program. With these properties, theoretically there will be a better mapping between the developers' mental model and their programs.

The concept of live programming that we use in this dissertation is the level 4 of Tanimoto, where modifying the behaviors of programs is directly related to a continuous action over a program, thanks to a meaningful feedback of the program, as Hancock states. We also use other categorizations when they are needed, such as Cyberphysical Programming, but they are not that important in understanding the concept of level of liveness, continuous action over a program and meaningful feedback of a program to developers.

We presented several live programming language, some of them used in robotics. However, all of them fall short in some aspects that are important for a complete solution to develop behavior-based robotics using live programming. In the next chapter we describe our own live programming language that addresses the aspects other programming languages miss.

Chapter 6

Live Programming in Robotic Behaviors: LRP

To map the developers' mental model to the behavior of their robots we built Live Robot Programming, LRP. LRP is a live DSL for behavior-based robots. As stated in Chapter 5, live programming should help developers better map between what they are thinking and the final behavior of their programs. We applied live programming to behavior-based robotics to allow developers to map their mental models to the actual behavior of the robot.

However, current programming of robot behaviors is far from 'live'. Section 5.1 explains the cognitive distance of a typical software development. Section 1.1 has an introduction of the cognitive distance in the particular case of robotic behaviors. Summarizing, the robot behavior is typically written, compiled and then deployed on a simulator (or the robot itself) for testing. Each step in this cycle widens the cognitive distance between the program and the robot's behavior, Figure 6.1 shows the cognitive distance in robotic behavior development. When testing reveals errors in behavior, the programmer needs to map this error back to the code before making changes there and repeating the development cycle. Hence, the feedback loop between writing code and seeing the results is not tight at all. This wide gap between writing and observing slows down development and can make experimentation with different behaviors prohibitively expensive.

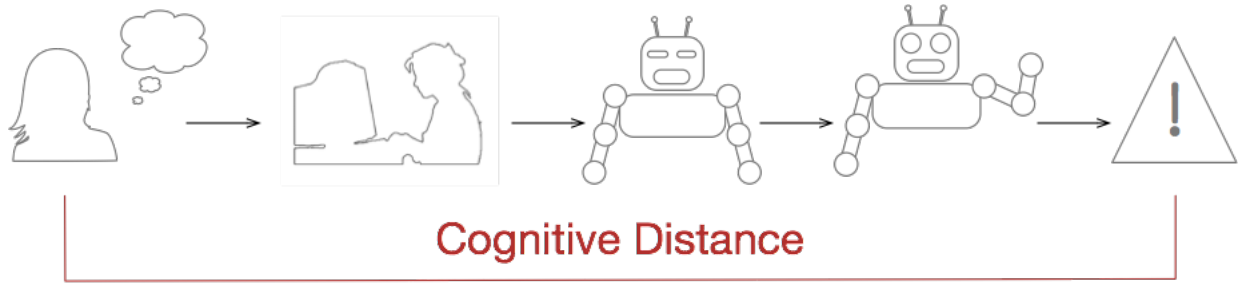

Figure 6.1: Cognitive distance in robotic development. Notice the replaced and added steps that differ from those in Figure 5.1.

LRP allows live programming for robotic behaviors. Because live programming can shorten

and even remove this cognitive distance, with LRP we can answer our second research question presented in Section 1.4: It is possible to reduce the cognitive distance in behavior-based development. Moreover, thanks to LRP, we accept our first hypothesis *H1* presented in Section 1.5: We have a live programming language for robotic behaviors using nested state machines, that can be used in several robotic systems.

In LRP, the programming language environment includes a visualization that acts as meaningful feedback of what the program is doing. This helps map the developers' mental model with their robotic behaviors. LRP also allows for immediate connection of the source code with the running robot. By reflecting the changes of the program right away in the visualization and the running robot, the cognitive distance is reduced.

LRP combines interpreted with compiled code by embedding a complete Object Oriented language whose statements are compiled. We discuss the interpreter of LRP, outlining the main interpretation loop as well as talking about how performance is taken into account, *e.g.,* through the use of compiled code. The liveness aspects of the interpreter are separately presented. Most importantly, we show how nested state machines can be modified while they are running, which almost never disrupts program execution.

Lastly, LRP is independent from the software that is used for lower-level control of the robot and hence can be used with multiple robot APIs or middleware. As an illustration of its independence we offer two instances of such bridges: one to the Robot Operating System (ROS) [17] and one to the Lego Mindstorms Ev3 [18].

We present LRP with several example videos, in particular: `http://bit.ly/LRPVideo`. We placed information and installation instructions on the PLEIAD webpage: `http://pleiad.cl/lrp`.

6.1 The LRP Language

LRP (Live Robot Programming) is a domain specific language (DSL) for nested state machines, which is arguably a natural paradigm for designing and programming robot behaviors. A DSL is designed to take advantage of the problem domain in their syntax and semantic to improve the developers productivity [52]. The development of software for robotics may be improved by using a DSL for a specific robotic domain [53, 54, 55].

LRP is inspired by existing languages for robot behavior programming based on nested state machines, *e.g.,* XABSL [28], Kouretes State Charts [27], the Lua behavior engine [30] and SMACH [6] (more on behavior-based programming in Chapter 2). We decided to base LRP on this paradigm because of its success in the RoboCup competition, for example, the typically highly ranked German team programs with a modified version of XABSL called CABSL [29] and several teams adapted the German team code for their own robots [56]. Note that LRP, like the languages it is inspired on, is not intended for computationally intensive applications such as image recognition. Instead the goal of the language is to enable the straightforward expression of complex behaviors based on already processed sensor inputs.

The core difference of LRP with respect to previous work is the focus on live programming, more precisely the level 4 of liveness of Tanimoto and including Hancock's concepts of continuous feedback and continuous action (as discussed in Chapter 5). The editor, shown in Figure 6.2, includes the integration of an always-on state machine interpreter whose machines change in sync with the code as it is being edited, and is coupled to a custom interactive visualization. In the middle pane, the tree of nested machines is shown. The programmer selects which machine to visualize in the right pane by clicking on a node of the tree. The machine visualization shows active states and last triggered transitions as well as the values of variables in scope, all of this updated as the interpreter runs. Furthermore, the editor also allows for the interpreter to be paused and stepped as well as the current values of variables to be modified by the programmer.

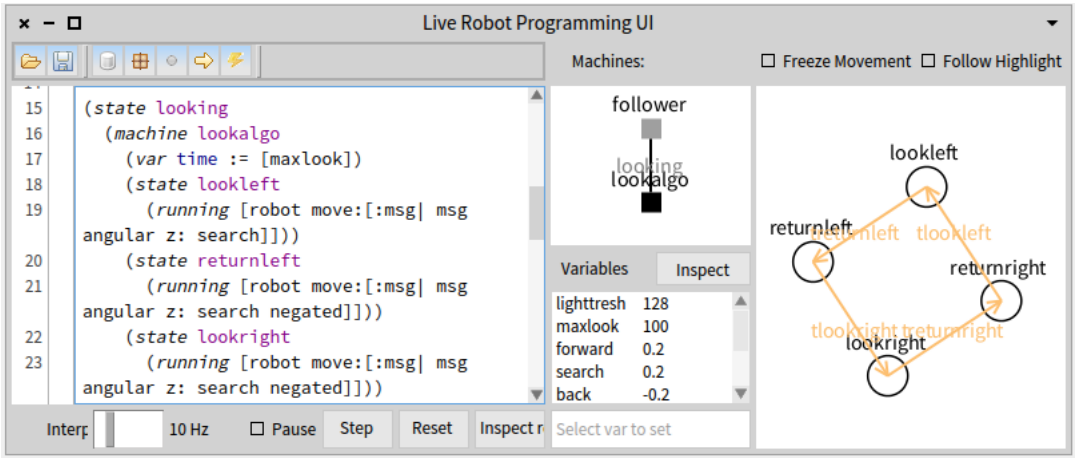

Figure 6.2: The LRP editor showing part of the running example of this chapter.

The UI editor is an important part of LRP, since the UI supports an important part of live programming: steady frames. With the visualization of variables and running states we support steady frames at "Program Execution" level. More specifically, the variables in LRP are constantly present, they can be manipulated at any moment and the variables are meaningful, because they were defined by developers themselves. The change of the variables in the execution of the program can be seen in the changes of the current states in the program itself. Moreover, the visualization of the program with state machines also allows us to support steady frames at "Program Development and Testing" level. The UI responds immediately to edits, updating the visualization as developers write their code. This makes it possible for developers to build and test programs from the very beginning, naturally. Section 5.2.2 offers more details on steady frames.

Considering the UI, there is no inherent requirement for the visualization to be present, nor even the UI for code editing. Indeed, LRP programs can be deployed on a 'bare' interpreter, which has no user interface and hence consumes fewer resources. Also, it is not the goal of the LRP UI to give all possible visualizations for robot sensors, *e.g.,* also showing an image that is being captured by a camera. Instead we expect that during development the LRP UI is complemented with other, existing, visualizations that show camera images or a visualization of a robot's world model, for example.

LRP is not coupled to any robotic systems in particular and could be used to develop

any behavioral solution. Hence, we claim that LRP is a live programming language when it could be classified as a cyberphysical programming language (more on live programming classification in Chapter 5).

6.1.1 Language Design

LRP is a language that allows for the description of *nested state machines*. Nesting in LRP means that a given *state* may contain a complete state machine (whose states may again contain a machine, and so on). The following example describes a nested machine inside a state:

```
(machine root
    (state a
        (machine nested)
    )
)
```

The LRP language is tightly coupled with its interpreter, due to its live programming nature and the need to interface with a robot API. A state can define more than one nested state machine, with this LRP supports the execution of concurrent machines (more in Section 6.3.3), as seen in the following example:

```
(machine root
    (state concurrent
        (machine m1)
        (machine m2)
    )
)
```

Next to nested machines, states may also have associated *actions*: an *onentry* action, an *onexit* action and a *running* action. Actions are snippets of code in the imperative object-oriented dynamically typed programming language Smalltalk[1]. States can declare any of the three actions multiple times, *e.g.,* a state can define 2 onentry actions and 1 running action. When the state becomes the active state, its onentry actions are executed, once. The execution of all the actions are atomic, *i.e.,* they cannot be interrupted by a state change. When the state stops being the active state, its onexit actions are executed once, also atomically. While the state is the active state, its running actions are atomically executed, once per interpretation loop. If a state with nested machines stops being active, the nested machines are stopped and discarded. As part of this process, the onexit actions in every active state of this nested machine is performed after any machines nested in that state are stopped and discarded. A developer can define actions inside a state as:

[1]This is because the interpreter and its connection to robot middleware is written in Smalltalk. Fundamentally this may be any other imperative programming language.

```
(machine root
    (state actions
        (onentry [Transcript crLog: 'initialize'])
        (running [Transcript crLog: 'executing'])
        (onexit [Transcript crLog: 'finished'])
    )
)
```

A machine in LRP may define variables and all actions may read, invoke methods on, and set all variables in scope. The scope of an action is its enclosing machine plus the machine that encloses it, and this up to the root of the hierarchy. Global variables may also be defined, outside of the root machine. Variables and actions act as the link from the interpreter to the outside world, allowing it to be connected, *e.g.,* to ROS or to any piece of software for which a suitable API is available. The variables are defined with the *var* keyword:

```
(var outsideRoot := [1])
(machine root
    (var insideRoot := [42])
)
```

Machines may also define *events*: named actions that are guards for transitions. If the result of the evaluation of the action is the boolean `true`, the event is said to *occur*. Transitions are also declared inside a machine with the keyword `on` and they connect two states of this machine. Transitions can be associated to any named event. The following example represents a transition associated to the `alwaysTrue` event:

```
(machine root
    (state a)
    (state b)
    (event alwaysTrue := [true])
    (on alwaysTrue a -> b)
)
```

In LRP, when an event occurs, transitions outgoing from the currently active state are inspected to see if they are stated to *trigger* on this event. This inspection happens from the root machine down the tree to the most deeply nested active machine, as executing a nested machine implies that its enclosing state is active. The direction of inspection from the root down to the leaves of the tree prioritizes leaving states that are at a higher level in the tree of machines. Note that this means that the interpretation of a currently executing machine stops when its enclosing state has become inactive due to one of its transitions being triggered. Also, in case multiple transitions may trigger in one machine, the first transition in lexical program order is triggered.

There are four kinds of transitions: *normal transitions*, *epsilon transitions*, *timeout transitions* and *wildcard transitions*. Respectively, they used the keywords: `on`, `eps`, `ontime` and `on` again. Normal transitions are the usual transitions we described above and they are represented with a blue arrowed line in the visualization. Epsilon transitions trigger automatically when the state is active (*i.e.,* after their onentry action is executed), that is why they are not associated to any event, they are represented as a green arrowed line. Timeout transitions

specify a timeout in milliseconds, either as a literal or a variable reference, and trigger after this timeout, they are seen as a orange arrowed line. Wildcard transitions have no specific source state, instead they consider all of the states in their machine as the source state, these transitions are represented as a purple arrowed line that goes between a fictional state named '*' and the destination state. This fictional state represents all possible states in the machine. The visualization of all of the different transitions provided by LRP is shown in Figure 6.3. The following example shows the syntax of the four transition types by connecting the same two states:

```
(machine root
    (state a)
    (state b)
    (event alwaysTrue := [true])
    (on alwaysTrue a -> b) ; normal transition
    (eps a -> b) ; epsilon transition
    (ontime 1000 a -> b) ; timeout transition
    (on alwaysTrue *-> b) ; wildcard transition
)
```

(a) Normal (b) Epsilon (c) Timeout (d) Wildcard

Figure 6.3: Types of Transitions in LRP. Note artificial wildcard source state in (d).

To identify the program starting point, LRP has a bootstrapping construct that specifies the machine to *spawn* and which is the start state of that machine. If the spawn construct is placed at top-level, the root machine is initialized:

```
(machine root
    (state a)
)
(spawn root a)
```

The spawn construct can also be placed inside onentry actions of states. These onentry actions will spawn the indicated state machines, enabling for nested state machine and concurrency in LRP:

```
(machine root
    (state a
        (machine nested1
            (state n1)
        )
        (machine nested2
            (state n2)
        )
        (onentry (spawn nested1 n1)) ; initialize nested1
        (onentry (spawn nested2 n2)) ; initialize nested2
    )
)
(spawn root a)
```

Last but not least, LRP is able to bridge the language and the robot API. Exposure of the robot API in the language is done via a pseudovariable named `robot`. It contains a Smalltalk object that acts as a facade to the robot API. LRP by itself does not send commands remotely to robots, but it depends on the bridges that LRP uses to work with robots, if the bridge provides a way to work remotely, so does LRP. Moreover, when the behavior is completed, we can work without the UI and load the program into the robot, even if it does not have a GUI display. This improves the communication time between the program and the robot. LRP currently provides bridges to four different robotics systems: ROS, the Lego Mindstorms EV3, Nao Robot and the AR Drone 2.0. Section 6.5 presents two of these bridges: the ROS and the Lego Mindstorms EV3 bridges. Moreover, the same chapter explains how to create custom bridges.

6.1.2 LRP By Example

To show the concrete syntax of the language and illustrate how it can be used to provide the behavior logic for a robot, we now present the program for a line following robot. This program illustrates almost all of the language features while remaining a conceptually simple task. The robot is a differential drive robot with a front mounted ground pointing light sensor and a front bumper, and uses ROS as its middleware (more on ROS in Section 3.1). Its task is to first follow a black line painted on the ground until it bumps into an obstacle. It then needs to turn around and follow the line back, until it again bumps an obstacle.

The code for this behavior is given below, and we will discuss it step by step.

```
1 (var lightlim := [128])(var maxlook := [100])(var forward := [0.2])
2 (var search := [0.2])(var back := [-0.2])(var turn := [1])
```

The first two lines of code show the declaration of six different variables: `lightlim`, `maxlook`, `forward`, `search`, `back`, `turn`. Each is initialized with their specific value, given between square brackets. These variables are defined at top-level and are hence global variables. In this code they are mainly used as calibration constants, *e.g.,* `lightlim` establishes the threshold between black and white for the light sensor.

For clarity we do not include here the code that creates the connection to ROS, nor the

full code of how commands are published, as this is not key to the example.

As mentioned above, the language provides for a bridge to connect to the robot, which is discussed more in detail in Section 6.5. To access the bridge, the interpreter provides for a pseudovariable named `robot`. In this example we use the bridge that works with ROS, as this is the middleware running on the robot. To access the data of the bumper and the light sensor, the developer specifies a subscription to a ROS topic in the bridge UI (discussed in more detail in Section 6.5.1). As a result, the developer can access the robot data via the `robot` pseudovariable. In this example, we subscribe to the bumper and the light sensor of the robot. Thus, we can access the bumper data with `robot bumper` and the light sensor data with `robot light`.

For sending data to the robot, and consequently moving the robot, the developer should specify a new publisher via the ROS bridge, using the `robot` pseudovariable (as explained in more detail in Section 6.5.1). For this example, we publish data that cause the robot to move. Typically in ROS, the type of a message for a topic that makes the robot move is `geometry_msgs/Twist`. Messages of this type contain speeds for linear movement on the `x`, `y` and `z` axes as well as angular speeds for these axes. In our example, the developer moves the robot with: `robot move: [:msg| msg linear x: aSpeed]`, moving the robot in the `x` dimension with a speed of `aSpeed`. Note that the block of code inside the square brackets only sets the `linear x` component of the message, leaving the rest of the components to their default value of 0.

Throughout this text, code between square brackets is in effect Smalltalk code. The evaluation result of this code is the result of the evaluation of the last statement. For example in the variable initialization cases shown in lines 1 and 2, these numbers simply evaluate to themselves. As the code that will be presented in this text is quite simple and can be read more or less like natural language, we do not discuss the syntax of Smalltalk more in detail.

```
3 (machine follower
4   (state moving (running [robot move: [:msg|msg linear x:forward]]))
5   (on outofline moving -> looking tlooking)
6   (on intheline looking -> moving tmoving)
7   (event outofline [robot light data > lightlim + 10])
8   (event intheline [robot light data < lightlim - 10])
```

Line three starts with the definition of a state machine, named `follower`. On line four, a state is specified, named `moving`. This state represents the machine moving straight, as it is located on top of the line. While this state is active, the interpreter will send commands on the `move` topic, instructing to move forward with a velocity of `forward`. Sending the message will happen once, per interpretation loop. Note that between each iteration of this loop a user-specified delay takes place (which may be set to zero).

Lines 5 and 6 define two transitions. The first triggers upon occurrence of the event `outofline`, defined in line 7, and causes a transition from the `moving` to the `looking` state. All kinds of transitions can be optionally given a name, and this transition is called: `tlooking`. The second transition is responsible for transiting back from the `looking` to the `moving` state.

Lines 7 and 8 define the events of interest for the above two transitions. Both use the light sensor, whose messages contain an integer value that is higher as the measured luminosity is higher. The `outofline` event on line 7 is triggered when the light sensor's last value should be read. It obtains the `data` field from the last message, held in `robot light`, adds ten to the light threshold, and tests the given inequality. This results in a boolean value. The `intheline` event lowers the threshold by ten and verifies the inverse.

```
 9    (state looking
10      (machine lookalgo
11        (var time := [maxlook])
12        (state lookleft
13          (running [robot move: [:msg| msg angular z: search]]))
14        (state returnleft
15          (running [robot move: [:msg| msg angular z: search * -1]]))
16        (state lookright
17          (running [robot move: [:msg| msg angular z: search * -1]]))
18        (state returnright
19          (running [robot move: [:msg| msg angular z: search]])
20          (onexit [time := time * 2 ]))
21        (ontime time lookleft -> returnleft treturnleft)
22        (ontime time returnleft -> lookright tlookright)
23        (ontime time lookright -> returnright treturnright)
24        (ontime time returnright -> lookleft tlookleft))
25      (onentry (spawn lookalgo lookleft)))
```

The `looking` state defines a nested state machine, and its definition spans lines 10 through 24 above. Note that the rightmost visualization in Figure 6.2 shows a diagram of this nested machine. The looking algorithm is an iterative left to right sweeping motion. Line 10 specifies the name of the nested machine: `lookalgo`. The four states (lines 12 through 20) respectively represent looking to the left, returning back to center from the left, looking to the right, and returning from the right. The sweeping behavior is orchestrated by the four timeout transitions of lines 21 to 24. Each times out after the time contained in the `time` variable. This initially contains the value of `maxlook`, but is multiplied by two at the end of each sweep, when leaving the `returnright` state (line 20). As a result, the size of the sweeping motion doubles at the end of each sweep.

Line 25 declares that on entering the `looking` state, the algorithm is spawned and the machine starts in the `lookleft` state. Note that exiting this state happens whenever the `intheline` event of line 8 occurs, causing the `tmoving` transition of line 6 to trigger. When this happens the `lookalgo` machine is discarded.

```
26    (var nobump := [true])
27    (event bumping [robot bumper data == 1 & nobump])
28    (event ending [robot bumper data == 1 & nobump not])
29    (on bumping *-> bumpback)
30    (on ending *-> end tend)
31    (state bumpback
32      (onentry [nobump := false])
33      (running [robot move:[:msg|msg linear x: back]]))
34    (state bumpturn
35      (running [robot move:[:msg|msg angular z: search]]))
36    (ontime 1000 bumpback -> bumpturn)
37    (ontime 3000 bumpturn -> looking)
38    (state end)
39  )
40  (spawn follower looking)
```

The last piece of code is responsible for the bumping and stopping behavior. Recall that on the first bump the robot turns around and follows the line back, and on second bump, it should stop. The `nobump` variable of line 26 records if the robot has not yet bumped. The events at lines 27 and 28 occur on a bump sensor press combined with the logical operator `and` on the variable, in line 27, and the negation of this variable in line 28.

Lines 29 and 30 are wildcard transitions that trigger on these events. Note that these have no origin state and the arrow notation is different: including the asterisk to highlight the 'wildcard' nature of these transitions. For example, the `bumping` transition takes the machine from all `moving`, `looking`, `bumpturn` and `end` states to the `bumpback` state.

The `bumpback` state makes the robot move backwards. The speed variable `back` is negative as declared in Line 2, allowing the robot to move backwards instead of moving forwards. The timeout transition on Line 36 triggers when the robot has moved on for 1,000 milliseconds (1 second). After that, the program enters the `bumpturn` state, where the robot turns. The robot will turn for 3,000 milliseconds (3 second) as shown on the timeout transition on Line 37. After the turning, the robot goes back to the `looking` state, where it starts its behavior again. On Line 32, when the robot begins the turning motion, the bump is recorded. This is important because if the bump sensor is touched again during the motion, the `tend` transition on Line 30 is triggered, allowing the program to go to the `end` state. If this were not the case and something touched the bump sensor, the robot would re-enter the `bumpback` state, restarting the turning motion.

The state in line 38 does nothing. Note that the state name `end` has no special status within the language, *i.e.,* the language has no concept of end (nor of start) states. In this program we consider this state truly as an end state as it does nothing and has no transitions that go out to another state.

The last line of the program: 40, instructs the interpreter to run the program by spawning the `follower` machine and making the `looking` state the active state. This makes the robot start by looking for the line.

6.2 Using LRP: Live Programming of the Looking Behavior

It is difficult to capture the experience of live programming in a piece of text. The most convincing argument for how it dramatically shortens development time is seeing it in action, which is arguably why Bret Victor's keynote [7] sparked wide interest, and pioneering work [8] has received little attention. As a textual attempt to convey how live programming with LRP enables rapid development of a behavior, we now present one scenario of use: developing the nested machine for the looking algorithm in Section 6.1.2 (lines 10 to 24). Recall that the interpreter and visualization of LRP are updated as each character of code is being edited, *e.g.,* showing new states immediately when their definition is complete, and highlighting them as soon as they become active. This is important because LRP supports steady frames at the "Program Development and Testing" level (more on steady frames in Section 5.2.2). Some video examples of LRP code being edited are available on the LRP website[2].

Consider the setting where the LRP development environment is deployed on a simulator of the robot or even the robot itself. The states and values in the environment reflect the states and values of the robot. The robot starts on the line, and goes forward until it leaves the line. In this example, let us suppose the robot leaves the black line and ends up on the right of the line, *i.e.,* the line is on the left hand side of the robot. This triggers a transition to the `looking` state. As the state has no behavior defined, the robot is frozen and the LRP interpreter visualization shows that no further state changes occur. Development of the looking algorithm may now start.

First, an empty nested machine `lookalgo` is added. In `lookalgo`, the states `lookleft` and `returnleft` and their timeout transitions are added (lines 12, 14, 15, 21, 22). In the `looking` state the onentry spawn statement is added. This last edit changes the currently active state, which requires the interpreter to be reset (as detailed in Section 6.4.1). Interpretation starts in the `moving` state, however the robot will not move because the running action will not be executed. Since the robot is still out of the line the interpreter immediately goes to the `looking` state. The robot makes a left sweep, detects the line and goes back to `moving`, again following the line.

All goes well whenever the robot keeps the line to its left hand side. When it leaves the line to the other side, it however starts looking left, and does not find it before the timeout occurs. This causes it to move back to the center. The visualization shows that first the `lookleft` state is active and after the timeout a transition is made to the `returnleft` state. There is however no outgoing transition from that state shown, because the code in line 22 refers to a state that does not exist yet! (Details on handling of incorrect code is given in Section 6.4.2). This means that the robot will never stop its returning motion. Seeing this, the programmer pauses the interpreter, which stops the robot as motor commands are no longer published. The programmer then adds the `lookright` and `returnright` states, let us suppose without the code of line 20. The interpreter is unpaused, the robot looks to the right and finds the line.

[2] `http://pleiad.cl/lrp`. Accessed: 04-07-2019.

It is important to remark that the program is built using tiny, correct steps, *i.e.,* developers build this program while testing it at the same time. This shows, previously explained, how liveness and visualization allow LRP to support steady frames at the "Program Development and Testing" level (as explained in Section 5.2.2).

All goes well until the line curves so much that it cannot be found by looking left or right in the specified timeframe. The robot endlessly sweeps left to right. The programmer can then, *e.g.,* increase the value of `maxlook`, immediately increasing the breadth of the sweeps. Experimenting with this value will in time, yield the right timeout for this specific case. Note that this use of immediate feedback to tweak the value of a parameter is an important feature of live programming languages, as explained in Section 5.2.2 when talking about steady frames at the "Program Execution" level. Alternatively, the programmer may provide a general solution in the form of the `time := time * 2` onexit action of line 20 omitted above. In that case the sweeps of the robot will progressively get larger, which is observable immediately after exiting the `returnright` state.

6.3 The LRP Interpreter

In this section we introduce the interpreter of LRP and how interpretation of LRP programs is performed. We discuss the main interpretation loop, how action blocks are compiled and end with a note on performance. While liveness is the main focus of LRP, we do not focus on the liveness features of the interpreter here, this is detailed in Section 6.4.

6.3.1 Executing a Program: The Main Loop

The interpreter simulates a continuous space of execution inside an infinite loop. The rate of this loop is defined in the interpreter and can be changed from the UI through a slider. The default rate value is 10 cycles per second. We choose this value because is the default rate value of ROS when publishing data over a terminal [57]. The loop starts when the interpreter finds a *spawn* statement, which specifies a machine and a state to start the computation. The loop consists of the next six steps (starting from the root of the machine hierarchy):

1. If no state is currently active, the start state indicated in the machine's spawn statement is selected to be the active state.
2. The selected state to be the active state is made active, updating the UI. The timers for the timeout transitions are restarted and its onentry actions are executed. If there is any spawn statement for nested state machines, the nested state machines are interpreted, starting from step 1.
3. For the currently active state all outgoing transitions are selected, and all wildcard transitions of the machines are added to this group.
4. The events of the selected transitions are extracted. Then, the actions associated to those events are run. Events for timeout transitions use their timers to determine

whether they occur; events of epsilon transitions always occur. When there is an event occurrence the destination state of the triggering transition is set to be the next state.

5. If no next state is set, the running actions of the active state are run. If the active state has any active nested machine, the nested machines are interpreted, starting from step 3. The interpretation loop ends here, the next loop starts in step 3.
6. A next state is set, so the currently active state is left.
   - if the active state has any nested machine, the active state of every machine is left, *i.e.,* step 6 is executed for that machine,
   - the onexit actions of the state are run,
   - the nested machines, if any, are discarded,
7. The next state is selected to be an active state. The interpretation loop ends here, the next loop starts in step 2.

Between each iteration of the interpreter loop a user-specified delay takes place (which may be zero). Modifying this delay allows tweaking CPU usage of the interpreter, in order to achieve optimal overall robot performance. Note that a discussion on performance of the interpreter is given in Section 6.3.4.

Optimizing Checking of Events

A program in LRP implements the behavior of a robot, therefor slowly executing the program translates into a slow behavior of the robot. It is therefore important to optimize the interpreter where possible. An important case of this is the evaluation of events, whereby evaluating an event in LRP may trigger the reading of a sensor. This can be slow, which produces delays in program execution.

To minimize this possible slowdown, the interpreter only checks the events of the outgoing transitions of the active state, not the events of all transitions defined in the program. This optimization is possible because the interpreter only needs to check the events that actually can make a change in the program, and not every event that is defined in the program.

Also, if there are two or more outgoing transitions from the active state, the interpreter checks the transitions in the order they were defined in the program. It changes the active state on encountering the first triggered transition, discarding the rest of the transitions without checking their events.

Change of the Active State

When a transition triggers, the interpreter changes the active state of the machine with the state defined in the transition. For a robot, this means there will be a change of its behaviors. Whenever there is a change of the active state, the timer for the timeout transitions is restarted.

Run Pertinent Actions

The state may defined actions to execute inside the loop when there is a change of active state or when that state is an active state. When there is a change of state, the interpreter executes the *onexit* action of the former active state, then executes the *onentry* action of the new active state. When there is no change of active state, the interpreter executes the *running* action of that state.

Nested Machines

In LRP a program can define a machine that is nested inside a state. To execute this machine, the developer places a spawn statement inside the onentry action of this state. When the interpreter encounters this statement, it reuses the interpretation main loop to execute the new machine. In other words, it does not create new loops or new threads for every nested machine in the system.

As a result, when there are nested state machines, the activity of checking a triggered transition is slightly more complex than in the normal case. The implementation of this check allows us to ensure that the transitions highest in the tree of machines get checked first as well as executing every onexit action of all currently active states in the correct order, *i.e.,* starting from the most nested state machine and then walking back up the tree of nested machines. As a result, nested machines may be exited from thanks to the taking of a transition that goes out of the state that contains them, recursively to the root machine, and all active states are able to perform cleanup operations in their onexit actions.

The activity proceeds as follows: If there is no change of state in the root machine, the interpreter executes step 5 of Section 6.3.1. If the state has a nested machine, after the state executes its running action, its nested machine is processed by the loop. This process is repeated for every nested machine until the state that has no nested machine. This activity can however be stopped by a transition that may be triggered, *i.e.,* step 4 of the loop sets a next state. After step 4 selects a next state, step 6 of the loop is executed on the nested state machine, if any. Hence no transitions are checked deeper down in the hierarchy and no running actions are performed. Instead, the onexit action of the most inner active state is executed first, and the interpreter goes up the hierarchy and executes onexit actions until it reaches the machine where the transition triggered. There, the interpreter goes to step 7 of the loop.

6.3.2 Variables, Scope and Blocks of Code

As stated in Section 6.1.2, variables are initialized when defined, using a Smalltalk block of code. Similarly, all action blocks are written as Smalltalk blocks of code, and these action blocks have access to all variables in lexical scope. We chose to embed a general purpose language in LRP for variable initialization and action blocks so as to have its full expressivity to the disposal of the LRP programmer. As a result, there is maximum flexibility of what

values variables can have and what computation is performed in action blocks. A direct consequence of this is that LRP is not bound to a specific robot API, as the code in blocks just needs to use the API as required and variables can hold whatever value worthwhile to the API.

Yet the use of Smalltalk variables and code should not come at the price of having a high overhead on the execution of action blocks or the lookup of values of variables. LRP is not intended for highly computationally intensive tasks, but will typically run as one of many processes on the robot, and so its execution should be as efficient as possible (a small benchmark is discussed in Section 6.3.4). Hence, in order to have minimal overhead, Smalltalk blocks are compiled and variable lookup is realized through the normal Smalltalk variable lookup mechanism. We therefore achieve native Smalltalk code performance for these blocks even though LRP is interpreted.

Compilation of action blocks and variable lookup works by using a hierarchy of generated classes that parallels the machine hierarchy, where variables and action blocks are placed in their corresponding machine. More in detail, the process is as follows:

1. For each interpreter a root class is generated, which is a subclass of Object.
2. For a top-level machine, a subclass of the root class is generated.
3. For a nested machine, a subclass of its parent machine class is generated.
4. Variables are added as class variables of their corresponding machine class (or root class for top-level variables).
5. For each action block, a method is generated in the corresponding class, whose body consists of a return of the block. The method is executed, and the resulting closure is what is run by the interpreter when the action block is run.

Steps one through three build the parallel hierarchy of classes, and step four places variable at the scope that corresponds to the lexical scope. Step five first causes the block to be compiled, and second, when the method is executed the creation of the closure captures the class of the method and hence, all variables in scope. An important aspect of the implementation is that variables are not instance variables, but class variables. The reason for this is the sharing of class variables: a subclass shares its class variables with its superclass. Consequently, when a nested machine changes a variable defined in a superscope (*i.e.,* a superclass), these changes are applied in the superscope and visible to all machines in this scope. If we were to use instance variables, changes in a superscope would only be visible inside this machine and moreover disappear when the machine is exited.

Note that due to variables being class variables, we cannot support variable shadowing in LRP. A variable that is defined in a parent scope cannot be redefined since in Smalltalk a class variable cannot be redefined in a subclass. This is the only implementation-based tradeoff present in LRP and we consider it reasonable since we gain native code performance, yet only lose variable shadowing.

6.3.3 Concurrence

LRP supports concurrent machines by allowing the possibility of spawning more than one machine with the `spawn` statement, on the root of the program or in any state as nested machines. For every interpretation loop, every active machine is executed one after the other. This means, one interpretation loop is in charge of execution every pertinent action on every active machine.

LRP does not support having more than one state being active in one machine. This decision is made to simplify the design of the language and its use. If LRP supports more than one active state, it also should support ways of joining the active states into one, ending the execution of one of the states, separating the execution of one state to more, etc. By allowing only one state per machine and multiple machines, all these problems are naturally resolved.

6.3.4 A Note on Performance

The speed at which LRP runs is important to the overall performance of the robot. Even though the behavior layer of the robot is typically *not* a computationally intensive task, it is one element in a long and possibly complex chain of computation. Hence it remains important for LRP programs to be as efficient as possible, and this was taken into account in the language and interpreter design.

Firstly, while development of software happens with the visualization and interpreter running, there is no inherent requirement for the visualization to be present, nor even the user interface for code editing. Indeed, LRP programs can be deployed on a 'bare' interpreter, which would have no user interface at all.

Secondly, even though the state machines are interpreted, all LRP actions in the program are compiled before the interpretation of the state machine starts. This allows actions to achieve the highest possible performance when they run.

Thirdly, while the interpreter is now written in Smalltalk, there is no fundamental requirement to use that language. A version of LRP can easily be envisioned that would run on a highly optimized interpreter for a particular robot platform.

Moreover, the LRP interpreter itself is already quite fast. As a microbenchmark for the LRP interpreter we have chosen to evaluate the speed at which state changes can be made in the presence of minimal actions. Since all LRP actions are compiled, the bottleneck in interpretation is arguably the interpretation loop of Section 6.3.1. To benchmark it, we run a program with 5 states that are fully connected with transitions that never trigger. Additionally, five transitions cause the program to run in a loop from state 1 to 5. Figure 6.4 describes a visualization of the program.

In this program, every onentry and onexit action returns a number, except for state `one`, where it updates the number of loops the program has made. On a laptop with a 1.8 GHz

Figure 6.4: Program with 5 states fully connected as benchmark.

Intel Core i5 CPU we run this program without the UI for one minute and count the number of loops, then we repeat this 10 times and take the average number of loops. With this information we establish how much time it takes to do a loop and how much time it takes to change a state, as can be seen in Table 6.1.

Average number of Loops	10828.7
Standard Deviation	72.79
Time per Loop (ms)	5.54
Time per State in 1 Loop (ms)	1.10

Table 6.1: Benchmark of a 5-state fully connected program

We see that the overhead for a state change is about 1 millisecond. This is on par with other languages for behavior specification, *e.g.,* Niemüller *et al.* [30] report the same time, however on unspecified hardware.

6.4 How the LRP Interpreter Enables Live Programming

Running an existing program along with its visualization is not the essence of live programming. The essence is providing support for running a program *while it is being changed by the programmer*. In other words, LRP supports level 4 on the scale of liveness of Tanimoto (level of liveness are explained in Chapter 5). For example, adding a new state to a machine should not cause its interpretation to start afresh. Instead, the currently active state should remain the same and values of variables should not change *as these parts of the code were not changed*. Correctly dealing with program changes and program errors allows the program to be adapted while it is running: adding new behavior, or modifying existing behavior 'live'. We present how this is achieved in LRP, together with interpreter features such as pausing and forcing state changes.

6.4.1 Dealing with Program Changes

To deal with program changes, the LRP interpreter briefly pauses when they occur, analyzes each change and determines in what way it affects the program being executed. These changes are then applied to the copy of the program used by the interpreter, before resuming interpretation. This process is described next.

A change in the code is first analyzed for syntactical correctness. While a program text has syntax errors, no changes are considered. Consequently, program changes are seen from one syntactically correct program to another syntactically correct program. They are therefore analyzed at the higher level of machines, states, transitions, events and variables. Regarding a program as a group of all these elements, we consider that a program change results in either elements being added or removed from this group. For example, when a new machine has been added to the program, the machine is added to the group, and when a transition has been deleted from the program, it is removed from the group.

Adding an element to the group never leaves the program in an undefined situation. The values of existing variables are unmodified, active states will remain active, hence running machines may keep running. When interpretation resumes, it only needs to take the added elements into account. Removing elements only leaves the program in an undefined situation in one particular case: when the active state or active machine is removed. In any other case, when interpretation resumes it simply needs to be without the removed elements.

Regarding the case of an active state or machine being removed, it is clear that the program can no longer be in that state or machine since it no longer exists. For the active state case, there is no transition that triggers, hence no onexit action to execute, nor another state to make active. Therefore, the machine that contains this state is invalid, which means that the state that contains that machine is invalid, and so on up to the root machine. Similarly, if an active machine is removed, the state that contains it is in an undefined situation as well. As a result, in both cases of the active state or active machine being removed, the entire program is in an undefined condition. In this case, interpretation of the program needs to be restarted from scratch.

Note that changing the name of an element is equivalent to a removal and an addition operation on this group, except for transitions, where the name is optional[3]. While the program keeps running in the face of these changes, making them may still produce non-critical inconsistencies, as listed below:

- Changing the name of an event: This action causes all transitions using it to no longer trigger, as these refer to the event name of an event that no longer exists.
- Changing the name of a state: This causes the states' incoming and outgoing transitions to no longer be valid, as these use the name of a state that no longer exists.
- Changing the initialization value of a variable: This change is the most significant case. It causes the old variable to be removed and a new one with the same name to be added. This is essentially equivalent to resetting the variable used by the interpreter

[3]The identification of transitions is performed by an unique identifier consisting of the union of the type of transition, the states associated to it and a name, if it has one.

to the new initial value.

Changing the body of an element, *e.g.,* adding actions to a state or changing the block of an event, keeps the program valid, *i.e.,* it does not enter in an undefined situation as explained before. However, a developer could edit the active state in ways that may not be reflected in the program right away. For example, a developer may add a nested machine inside the active state, adding the spawn statement on the onentry action. The active state has however already executed the onentry action when it became an active state, hence the nested machine will not spawn until this state will become inactive and active again. To avoid such a delay, the interpreter instead treats these changes as if the state is removed and then added again, *i.e.,* it restarts the program to ensure that this change may be reflected in the normal flow of the program.

To conclude: our analysis reveals that nested state machines are actually quite robust in the context of live programming. The only case where a program needs to restart from scratch is when an active state or an active machine is removed. One notable case of this is when there is a modification of an active state, as it is treated as a removal of the old and adding of the new state. All other changes in program code allow interpretation to seamlessly continue.

6.4.2 Dealing with Program Errors

Code that is syntactically correct, and hence is run by the interpreter, may however still contain errors. For example, in Section 6.2 we saw the addition of a transition that specifies the name of a destination state that does not exist yet. The interpreter should nonetheless keep running, allowing the programmer to keep adding code, supposing that the missing state will be added at some point. Another example is actions referring to variables that are not present. Again, the interpreter needs to keep running, or otherwise the liveness aspect of LRP is lost. In general, the interpreter needs to deal with errors in the program in the most relaxed way possible, prioritizing keeping itself running in a consistent fashion over stopping and throwing an error.

To allow this to happen, the LRP interpreter ignores the following errors:

- A transition makes a reference to an event or state that does not exist.
- An event returns a non-boolean value.
- A spawn statement refers to a machine or state that does not exist.
- A reference is made to a variable that does not exist.
- The execution of an action causes an exception.

Currently, the interpreter ignores the entity that produced the error in the interpretation loop, *e.g.,* in the first case the transition never triggers. It does however notify when it encounters such errors, using an auxiliary window where all errors are gathered. If errors need to be analyzed in detail, in this window the developer can set the option to stop ignoring errors and instead pop up a Smalltalk debugger window when errors are triggered.

6.4.3 Pausing the Interpreter

To give more power to developers, the interpretation loop can be paused. To avoid execution problems, *e.g.,* pausing when an action of a state is executing, the pause of the interpreter is always performed at the beginning of the interpretation loop (before reentering the loop explained in Section 6.3.1).

Sometimes, the execution of a behavior in one state can last long enough to understand program execution, but, in some cases, the behavior of a robot may consist of many state transitions in 'a blink of an eye', giving no time to developers to understand what is happening. For example, if we continue with the line following robot code in Section 6.1.2, if the road has a closed curve, the robot would keep changing the current state from `moving` to `looking`. Moreover, because the `looking` state has a nested state machine, it may be more difficult to analyze what is happening if the current state from the `lookalgo` machine also quickly changes.

When the program is paused, it can be analyzed to see what is happening at that exact point of execution. Because of this, developers can always see what is happening in their programs, irrespective of the speed of state transitions. Moreover, when the interpreter is paused, developers can also advance the execution step by step. Following the same logic as before, the program may run too fast to be analyzed. In these cases, going step by step over the computation helps them better understand the changes of states and variables. In other words, they can better see what is happening with the program that is translated into robot behavior.

Lastly, pausing the program allows developers to build complete behaviors before testing them on the robot. Following the same example of the line following robot, they may want to build the behavior of the machine `lookalgo` before testing it on the robot. They can pause the interpreter and write the code of this behavior, then unpause the interpreter when they feel confident enough that the code will work.

6.4.4 Forcing States

When developers are working on a program they may need to skip to some parts of the computation of the regular execution path. For example, consider the code of a line following robot explained in Section 6.1.2, where the program is in the state `looking`, *i.e.,* looking for a line to follow. If there is a bug in that specific part, developers may want to test only that part of the code without waiting for the program to reach this specific point of execution. If we follow the scenario presented in Section 6.2: developing the behavior of looking for the line, when the program reaches the `looking` state, the nested machine is spawned and starts in the `lookleft` state. If however we are debugging the `lookright` state, *i.e.,* where the line is to the right of the robot, we wish to skip looking to the left and then returning from the left. Instead the robot should immediately be brought into the `lookright` state.

To avoid unnecessary waiting to test specific parts of the program, the interpreter allows the execution to jump to a different state from the currently active state, as instructed by

the developers. To do that, they right-click a state and select one of the following items from a context menu:

- **transition here**: This option allows the program to skip to the desired state, executing the onexit action of the current state and the onentry action of the selected state. This action works as if an epsilon transition is connected between the current state and the desired state, and this transition is immediately removed after it triggers.
- **teleport here**: Contrary to the previous option, this works without executing any action: the program *teleports* from the active state to the desired state, *i.e.,* the interpreter changes the active state for the selected state without executing any actions.

Following the principles of live programming, both ways of skipping a part of the programs' execution work seamlessly. When the developer forces a state, the interpreter briefly pauses, changes the active state and then the program is resumed as if was a completely normal change.

6.4.5 Live Programming and the UI

The UI is an essential part for LRP as it is crucial for the live programming experience. Without the UI, development cannot benefit from live feedback in an intuitive way, *e.g.,* showing the program as a graph of states and transitions and highlighting the state being executed. Moreover, with the UI we can support steady frames (presented in Section 5.2.2) at different levels, as explained in Section 6.1. Here we explain three fundamental parts of the UI: how the live feedback of the program is produced by the UI, how developers can interact with programs using the UI and lastly, how developers interact with the interpreter of LRP using the UI.

First, as said in Chapter 5, fundamental to live programming is showing developers what is happening inside their programs in a meaningful context. To do this, the LRP UI presents a visualization of the program as a graph of states and transitions. To visualize the running program, the interpreter notifies every change of state to the UI, which then highlights the corresponding shape in the visualization. Moreover, the UI shows a visualization of the nested machines in the form of a tree. The machine can be clicked to show its states and transitions. If the machine is active, developers see what the active state of that machine is, highlighted. Thanks to this visualization, we can support steady frames at the "Program Development and Testing" level. As explained in Section 5.2.2, this allows developers to code and test the programs as they constantly grow in a natural way, thanks to the continuous feedback of the visualization.

Second, in the visualization of individual machines, when a right click is made on a state it opens a list of options to interact with the state. The options are: "transition here" and "teleport here" (as explained in Section 6.4.4), and "inspect me" to inspect the internal structure of the state. Also, the UI shows a list of all variables in the scope of the selected machine. Variables reflect their changes in real time, *i.e.,* the value of a variable in this visualization changes if the value changed in the execution of the program. Variables can be inspected to see their internal structure and their values can be modified directly in the

UI. This is important to support steady frames at "Program Execution" level. Variables are always visible and can be modified at any point of execution (more on steady frames in Section 5.2.2).

Lastly, the UI provides ways to interact with the interpreter itself. First, it provides a slider to change the rate of the main loop of the interpreter (as discussed in Section 6.3.1). Second, it provides buttons to reset interpretation, *i.e.,* to restart the program, and to pause the interpretation (see Section 6.4.3). Third, it provides a button to inspect the internal structure of the bridge to the robotic system the program is using at the moment (more information about bridging LRP to robots' API is given in Section 6.5).

As explained in Section 6.1, the UI is however not required to deploy a LRP program. The goal of the UI is to improve the development experience of programs, after the program is finished. It can be discarded to reduce the overhead of LRP, *e.g.,* when deployed inside robots.

6.5 Bridging LRP To Robot APIs

LRP is not coupled to any robotics system in particular, but instead has the potential to work with practically any robot for which some kind of API is available. To illustrate this we built two bridges for two different robotic systems: ROS [17] and the Ev3 Lego Mindstorms [18].

Both API's are of a fundamentally different nature. On the one hand, ROS is a middleware for robots that runs on more than one hundred different robot platforms. It is a heavyweight system, not structured to work with live programming. On the other hand, the Ev3 Lego Mindstorm is an intelligent 'Lego brick' that allows the user to construct their own robots out of Legos. Ev3 is a lightweight system, yet only designed to work with the proprietary Lego programming language. However, thanks to PhaROS [58] and JetStorm [59], Pharo Smalltalk clients for ROS and Lego Mindstorms respectively, we have the opportunity to work with these systems inside Pharo Smalltalk and, consequently, with LRP.

As a result it is feasible to use LRP on a wide range of robots. However both systems were not designed to use live programming, making the building of behavior with LRP on these platforms not as intuitive as it could be. Because LRP is intended to be as intuitive as possible, we built bridges to communicate LRP with both APIs to improve the live programming development experience. This illustrates that we can enable live programming on a wide variety of APIs, through the implementation of an appropriate bridge, as we discuss in this section.

6.5.1 Bridging LRP to PhaROS

ROS [17] takes a more static approach of building programs, compared to live programming. An executable program is called a *Node* and different nodes can communicate via channels called *Topics*. Topics are the main form of communication between programs and the robot

itself. To communicate with topics, developers should specify the type of the data that it will transmit. While topics are dynamic, as they can be created on the fly by nodes, nodes are highly static since they cannot be modified on the fly at all. For more information on ROS, we refer to Section 3.1.

PhaROS [58], a client for ROS in Pharo Smalltalk, does allow the code of a node to be modified at runtime, but is not yet a fully live programming environment like LRP. Since LRP is a DSL implemented in Pharo Smalltalk, it can use PhaROS to access ROS, but using plain PhaROS can be quite troublesome. To receive data from a topic, PhaROS builds a subscription object that holds a callback. This callback is then executed every time the PhaROS subscription receives data. In LRP, we could use this code to build a subscription inside an action of a state. However, because of the live nature of LRP, this action may be executed numerous times. For example, every time the program restarts, it may execute this action again, although the subscription should only be made once. As a result we may end up with multiple subscriptions holding the same callback, resulting in a huge overhead of the program. Moreover, if the developer wants to change the code of the callback, the original subscription is already made and cannot be changed. As a result, a new subscription must be made, and there will be two or more subscriptions doing different things, whereas the developer only wants the last subscription.

Another problem is the complexity of the code for accessing topics to receive or send data. As previously mentioned, to receive data developers need to make a subscription to a topic. To send data, developers have to build a publisher to a topic. These processes are not straightforward. For example, a subscription is done in PhaROS as follows:

```
(aPhaROSPackageInstance controller node buildConnectionFor: aTopic)
  typeAs: aType;
  for: aCallback;
  connect.
```

In this code the `aPhaROSPackageInstance` is an instance of `PhaROSPackage` class, or an instance of a subclass of `PhaROSPackage`. `aTopic` is the name of a ROS topic and `aType` is the type of its data. Next, `aCallback` is the block of code that is executed every time the instance receives data from the topic. Finally, the `connect` statement tells the instance to initiate the connection with the topic.

To publish data, the pattern is as follows:

```
publisher := aPhaROSPackageInstance controller node
                topicPublisher: aTopic typedAs: aType.
publisher send: aBlock.
```

Here, the `aPhaROSPackageInstance` is the same as before. `aTopic` and `aType` also represent the topic and the type. The first two lines return a publisher object that can be used to send data. Lastly, to send data to the robot, a `send:` message is sent to the publisher object with the data constructed inside a block, *i.e.,* `aBlock` in our example.

The level of abstraction in using the API is not where we want it to be since it exposes low-level details that are irrelevant to the LRP programmer. Instead, this programmer should solely deal with the relevant domain concepts: the topic and its message type. Instead, it

should be possible to subscribe to topics and use their data as normal variables, and publish data through a concise API call.

To address these issues, we introduced a special bridge between PhaROS an LRP. It effectively crosses the divide between the extremely dynamic world of LRP and the less dynamic world of PhaROS. This this is done by requiring the programmer to manage topic connections through a user interface and, once these are set up, reifies topics as variables of the `robot` object that can be accessed fully dynamically.

When the bridge is installed into LRP, the system will ask for the name of a class when the interpreter is started. This class is used to generate the subscriptions and publishers, *i.e., aPharosPackageInstance* in the previous example. The class should be either `PhaROSPackage` or a subclass.

After giving the class name, LRP opens with a special window that shows the publishers and subscriptions that have been already made. It also provides the option to create new publishers and subscriptions. This window can be seen in Figure 6.5.

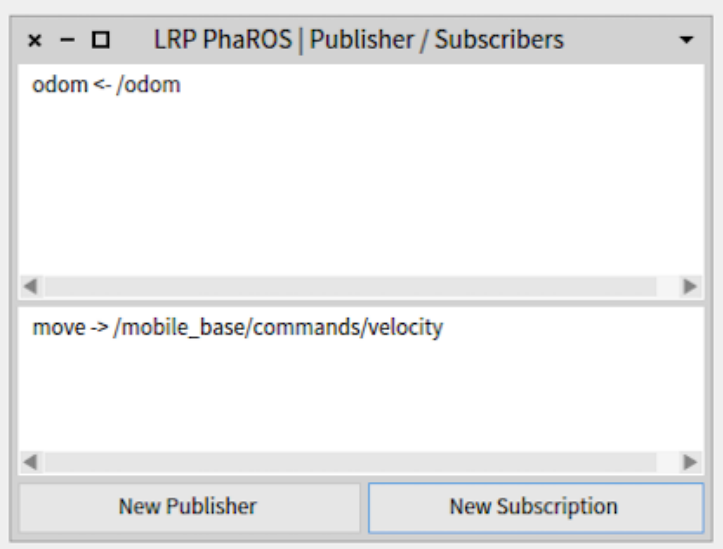

Figure 6.5: GUI showing publishers and subscriptions and the buttons to create them.

In this window a developer can choose between generating a publisher or a subscription. When choosing any of the two, another window pops up asking for more information: the name of a variable that will be used inside LRP, the topic name and the type of message topic. This pop-up can be seen in Figure 6.6.

For example, assume we need to create a subscription for odometry data. This data is available on the topic `odom`, which has the type `nav_msgs/Odometry`, and we wish to access it through a variable named `odom`. When done, in LRP actions can access the last odometry data published on the topic like this:

```
robot odom
```

In this example, the pseudovariable `robot` represents the bridge between LRP and the API that is currently being used, in this case PhaROS (similar to what is shown in the example

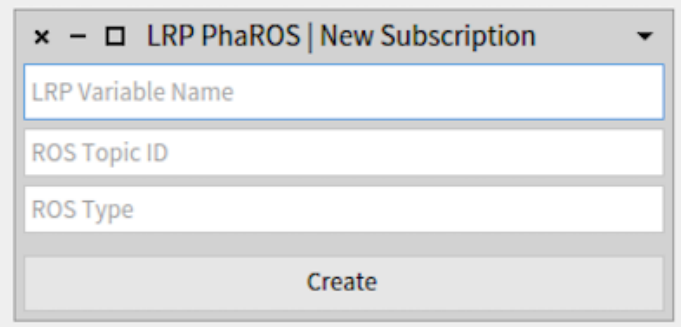

Figure 6.6: Window asking for information to make a subscription.

code in Section 6.1.2). To access the data of a subscription, a getter message is used. In Smalltalk, the name of a getter message is the same name as the variable name, in this case `odom`.

For publishing data, a setter message for the variable is used, where the argument is the data that is sent to the topic. For example, to move a robot on the topic `/mobile_base/commands_velocity` of type `geometry_msgs/Twist` and variable name `move`, we use the following code:

```
robot move: [:data|
  data linear x: forwardSpeed.
  data angular z: angularSpeed
].
```

In this case, the setter message `move:` sends the data to the topic. The argument of the `move:` message is a block of code with one argument: a data object that will contain the data of the message. To fill the object with the corresponding data, setter messages are used to fill the respective instance variables. In our example, we are moving the robot with a linear velocity in `x` of `forwardSpeed`, *i.e.,* moving the robot forward with speed `forwardSpeed`; and angular velocity in `z` of `angularSpeed`, *i.e.,* turning the robot with speed `angularSpeed`.

This bridge offers developers several advantages:

- It is easier to write and remember rather than the raw form of PhaROS.
- In the UI, when writing the name of the topic a list of possible topics appears, allowing developers to select one from the list.
- After selecting a topic, the field of the type of the topic is automatically filled.
- A subscription can be used in different ways in LRP by just referring to the pseudovariable `robot`. Developers need to just add one subscription to a topic and not add multiple subscriptions on the same topic just to change the callback.

6.5.2 Bridging LRP to JetStorm

JetStorm [59] is a client to the Ev3 Lego Mindstorms [18] that allows for remote control of an Ev3 though a TCP/IP connection over WiFi. It is a lightweight API that holds no state

except for the hardware configuration of the Ev3, *e.g.,* its IP address. The JetStorm API allows developers to send commands to the robot to move motors or to read data from the sensors. The Ev3 has 4 ports for motors and 4 ports for sensors and the JetStorm API reifies these 8 ports as instance variables of an API object. For motors it uses the variables `motorA` to `motorD` and for sensors it uses `sensor1` to `sensor4`.

The bridge of LRP to JetStorm consists of simply making the JetStorm API object available as the `robot` pseudovariable. As a result, for example, to read the value of a sensor the message `read` is sent to a sensor instance variable, as below for sensor 1:

```
robot sensor1 read
```

JetStorm provides a rich API to work with motors. For example, to move the A motor with a speed of `aSpeed` the following code is placed in an LRP action block:

```
robot motorA startAtSpeed: aSpeed
```

Due to the nearly stateless nature of the API it resulted in being more straightforward to bridge JetStorm than PhaROS to LRP. For example, there are no possible conflicts between callbacks for message arrival on a topic, since there are no such callbacks. Alternatively, there is no need to set up topics for sending commands to the motors as they are directly accessible. In general, there is no setup needed by the programmer to be able to communicate to sensors or motors. Instead, code that performs such accesses is directly written. This statelessness of the API matches naturally with live programming.

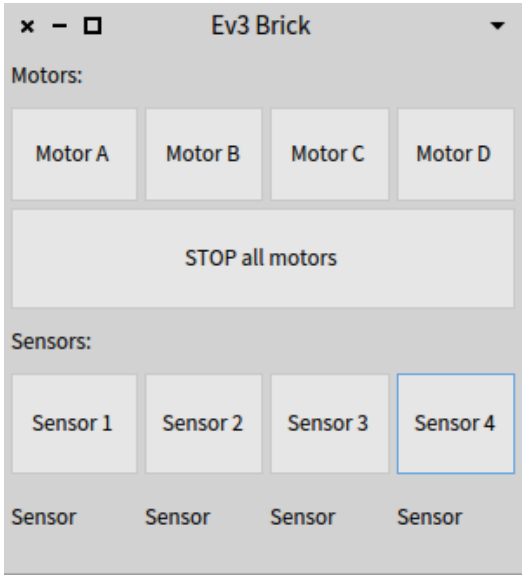

Figure 6.7: GUI of the Ev3 Brick showing all motors and sensors.

There is however one element of state that needs to be taken care of: the connection of the machine running LRP with the Ev3 Brick. As with PhaROS when doing a subscription, we want to connect to the Ev3 just once. However if we put that code inside an LRP program there is the possibility of connecting to the Brick every time the program restarts, which in our experience can crash the API.

To address this issue, the LRP Bridge for JetStorm opens a dialog box asking for the IP address of the brick whenever the interpreter is started. It then sets up the connection to the Ev3 before allowing the interpreter to continue. To further aid in development, the bridge has a GUI that provides access to every motor and sensor of the brick and allows developers to interact with every component individually, as shown in Figure 6.7. Moreover, the GUI provides for a button that acts as a runstop: when pressed it stops every motor, which proved to be a very useful function when we built programs on the Mindstorms.

6.5.3 Building a Custom Bridge

We built the bridge infrastructure with flexibility and extensibility in mind, so any developer can build his/her own bridge to his/her own robotic API. The bridge should however respect the following guidelines:

The bridge should extend the class `LRPAbstractBridge`. When the interpreter is started, it will reflectively determine if the abstract bridge class has a subclass, and if so instantiate an object of this class. This object should create a temporary class dynamically, and in this class generate the variables and methods to access the information of a robot. For example, the PhaROS bridge populates this class with methods to access data from a robot and methods to publish data to a robot whenever topic subscriptions are made. To aid in code generation, the logic for creating this temporary class is incorporated inside the `LRPAbstractBridge` class, and the functions to add variables and methods to the temporary class are also incorporated inside the abstract bridge. As a result, every variable and method to access the robot defined inside the bridge can be reached through the `robot` pseudovariable inside an LRP program.

The only required method for a bridge is the `openUIFor:` method. This method has the responsibility of initializing the bridge, *e.g.,* asking for the IP of the Ev3 Brick in JetStorm or asking for the name of the class in PhaROS. This method also has the responsibility of opening any GUI the bridge may need. The method receives an instance of the interpreter itself if it is necessary for the bridge to work.

With this implementation we expect that any other API with robot functionalities and GUIs should be implementable as a bridge inside LRP. In particular, there are other two robotic systems that are bridged with LRP thanks to the LRP bridge design. These robotic systems are the Nao robot, a humanoid robot; and the Parrot AR.Drone 2.0, a quadcopter drone.

6.6 Conclusions

This chapter presented LRP: a live programming language for the construction of robot behavior using nested state machines. With LRP we answer our second research question and we accept our first hypothesis *H1:* we have a live programming language to build nested state machines in a robotic context using live programming.

We gave an overview of the language, starting with reviewing design considerations and then detailing the language constructs by example. This was followed by an instance of the use of LRP that shows how it enables live programming of robot behavior. LRP is an interpreted and compiled language, the latter by embedding Smalltalk. We outlined the core interpretation loop, showed how Smalltalk code is embedded and compiled, and established that interpretation overhead is on par with other robotics languages for nested state machines.

What enables the liveness of the language is how the interpreter manages to run code while it is being changed by the programmer and the features that allow control over the interpretation loop. We therefore discussed in detail how changes in the program code are dealt with and how errors are handled. Notably, we established that the concept of nested state machines is quite a robust paradigm in the context of live programming. We also presented interaction features such as pausing and restarting the interpreter and forcing a state transition to a given state and ended with an overview of the UI features.

LRP is not restricted to programming robot behaviors. This is because it is inherently decoupled from a specific API. Instead, links to robotics APIs are performed through separate bridge software. We discussed the implementation of two different bridges for two contrasting APIs: one to ROS, arguably the most popular robot middleware and of certain complexity, and one to the Ev3 Lego Mindstorms, which is an extremely lightweight API. This illustrates that LRP can be used on a wide variety of APIs. We also presented more bridges that were built and what needs to be done to implement a new bridge.

An interesting observation resulting from this work is the robustness of nested state machines in the context of live programming. The only cases where program interpretation has to be resumed from scratch is when an active state is changed or removed, or when an active machine is removed. It is also worthwhile to notice that LRP is not necessarily limited to the field of robotics. Since its action blocks can interface with any piece of software for which an API is available, it is feasible to use LRP as a general nested state machine language.

As stated in Section 1.1, in behavior-based robotics, the intelligence of a robot depends on the interaction of the robot with the real world. Thanks to the robustness of nested state machines with live programming, we can seemingly develop behavior-based robots while they interact with the real world. This may improve the efficiency of developing the intelligence of the robot.

Saying that, it is important to notice that we only focus on live programming on the language level and not the hardware itself. As stated in Section 5.3, a family of robotic programs may have problems when using liveness without addressing the state of the robot in the real world. Nevertheless, another family of programs may still work by using live programming, without the need of modifying the state of the robot in the environment.

Chapter 7

LRP in Practice: a Controlled Experiment

In the previous chapter we presented our own live programming language for robotic behavior using nested state machines, LRP. With LRP, we are ready to test our second hypothesis *H2*: Live programming improves the speed and accuracy of program understanding and program writing in a behavior-based context. Even more, we are able to find a solution for our third research question (presented in Section 1.4): we can use LRP to test if live programming improves developer productivity in behavior-based robotics.

Through a literature review we found, to the best of our knowledge, that there are no studies on the advantages of a live programming environment in the context of robotic behaviors, nor even in a more practical setting where we can compare. When we talk about a practical setting we mean a nontrivial context where there are dependencies on possibly complex external API's, like the context of developing robotic behaviors.

These experiments establish whether the advantages of live programming that have been shown in a restricted context are also applicable in the development of behavior-based robots using nested state machines.

We have therefore performed research on live programming in robotic behaviors using LRP as a live programming tool. For this study we considered the development practices of code comprehension and code writing and measure the speed and accuracy of both practices.

We performed studies comparing LRP with SMACH, an arguably popular non-live system to develop robotic behaviors. With this comparison we establish if robotic behavior program understanding and writing benefits from live programming by using LRP. For this, we performed two within-subjects controlled experiments with the aim of evaluating the accuracy and speed of development using LRP against SMACH. The first experiment considers program comprehension and the second experiment considers program writing.

Notably, the quantitative results of the experiment show us that there is **no** difference in program understanding nor in program creation, both considering correctness and time

taken. However, this contradicts previous experiments about live programming in a more restricted context. Nevertheless, the qualitative results support the use of LRP over SMACH. Our observations on subject behavior lead to three possible causes for the low performance of LRP: 1. Live programming is not the panacea of software development, 2. The complexity of the ROS middleware and 3. Missed opportunities by the test subjects.

This chapter is structured as follows: We first detail the overall experiment design. Section 7.2 details our first experiment on program comprehension, and Section 7.3 talks about our second experiment on code writing. We follow with an overall discussion of our results, before treating threats to validity. Lastly, Section 7.6 discusses related work, followed by conclusions.

7.1 Overall Experimental Design

To test if live programming improves the speed and accuracy of program understanding and program writing, we conduct two controlled experiment comparing LRP against a baseline: SMACH. We talked in depth about both tools in Chapter 6 and Chapter 3. An example video of LRP is found online: `http://bit.ly/LRPVideo`. This video shows how the visualization evolves while a developer writes a program. We also present a video of the same example program presented in the video before, but using SMACH: `http://bit.ly/SMACHVideo`. For brevity, in the second video we did not record the program being written from scratch.

The two controlled experiments we conducted share a common design, which is described in this section. Particularities of each experiment are described in Section 7.2 and Section 7.3. We mostly follow the work of Jedlitschka and Pfahl to report our studies [60]. To not repeat the same through both experiments, we grouped information of the two into one in this section where possible.

Both controlled experiments are within-subjects design, *i.e.,* subjects participating in all treatments and we measure all the participations of the subject in every treatment. We then cross-evaluate LRP with SMACH and measure the impact of using LRP.

As part of the design of both experiments we first performed pilot studies to fine-tune the difficulty level of the experiments and to ensure that the overall duration of the sessions is acceptable.

7.1.1 Goals

The purpose of our two controlled experiments is to evaluate the accuracy and speed of using LRP against a traditional approach for the development of robot behavior. We focus on the impact of LRP on two development activities: program comprehension and program writing. In the first experiment we evaluate the understanding of existing code, and in the second experiment we evaluate the building of programs.

7.1.2 Dependent and Independent Variables

In our experiments the dependent variables are the correctness of performing a task in each experiment (*i.e.,* the accuracy) and the completion time (*i.e.,* the speed). Our independent variables are the systems used in the experiment, LRP and the baseline. Lastly, we created two tasks per experiment for subjects to resolve. These tasks constitute another independent variable.

7.1.3 Baseline

We select SMACH as a baseline for our experiments because beyond its popularity, Python and SMACH have key similarities with LRP: the language is dynamically typed, SMACH programs are written as nested state machines with approximately the same features as LRP, and SMACH also provides a visualization of the running machines. We presented a full introduction in Section 3.2.

Note that we want to compare LRP and SMACH as complete systems and avoid comparing individual features. Comparing both systems is ambiguous in terms of which specific features have an impact on our results, however both systems can be compared in terms of accuracy and speed. With this, we only have results on complete systems, but we can not be sure which feature impacts the most on our results.

7.1.4 Experiment Design

Because of the nature of the researched tools and the availability of the necessary equipment, we chose to carry out controlled experiments with only one participant at a time. Both experiments required computers with specialized programs to work, such as ROS.

To minimize noise resulting from the technical aspect of running robots, we conducted our experiment in a simulated environment, the Turtlebot robot[1] in the Gazebo robotics simulator[2]. Note that simulation of robot behavior as part of the development process is a standard procedure in the robotics community as it allows one to abstract from hardware issues and accelerate the development process.

The communication between the program and the robot is made by several channels called *topics*. These topics have meaningful names following the ROS naming convention. Moreover, we also give a list of all useful topics per experiment with a clear description. This list can be used by the participants at any point in the experiment.

Even when both experiments can be performed without our supervision, we decided to supervise them to monitor, in parallel, the behavior of the participants in performing the tasks. This was extremely useful to arrive at several conclusions for both experiments.

[1] `http://wiki.ros.org/Robots/TurtleBot`. Accessed: 04-07-2019.
[2] `http://gazebosim.org/`. Accessed: 04-07-2019.

7.1.5 Participants

The participants were picked for their familiarity in ROS, state machines as a programming model and their availability to be physically present in our laboratory, since we require a particular setup.

As far as we know, the only people in Chile that work with ROS and robotic behaviors using state machines are students from the University of Chile. In particular, there is a robotic team in the university, the UChile Homebreakers team[3]. This team participates in the RoboCup@Home league, made up of Electrical Engineering students.

In the computer science department there was a class of software engineering for robotic applications using ROS. We also used the students of this class as participants of the experiments.

The participants only participated in one of the two experiments to avoid a learning bias where they can improve or deteriorate their performance in the second experiment. This is especially true because we reuse some codes from the first experiment to create the second one.

7.1.6 Task Setup

In both studies, we performed a cross-evaluation experiment where we conceived two different programs that express a robot behavior. The participants then needed to understand and respectively write the robot behavior code. For all tasks we also implemented a simple textual interface for the robot, allowing it to ask the users to take certain actions and to provide them with information.

7.1.7 Work Session

Each work session take about 4 hours. The activity of each work session was structured as follows:

1. Answer a questionnaire with personal information and background including age, the education level and previous knowledge of several tools used in the experiment.
2. Read a description of the system used in the first task. We call this phase the *Warm-up* phase for the first task.
3. Evaluation of the first system, using the first task.
4. 15 minutes break.
5. Warm-up phase for the second task.
6. Evaluation of the second system, using the second task.

[3] `http://robotica-uchile.amtc.cl/about.html`. Accessed: 04-07-2019.

7. Answer a *post-questionnaire* with qualitative information about the participant and the experiment itself.

Warm-up Phase The participants are not required to have knowledge about the systems that we are going to test. To achieve a necessary level of knowledge, we give them reference material of the system to use. The reference materials for LRP and SMACH explain a simple example that gives the required knowledge of the features that are going to be used in the experiment itself. The participant can use the reference material in the experiment at any time. The materials also contain an explanation of the communication API to the robot used by the program. All this material is online at: `http://bit.ly/2j1P1Zb`.

At the end of the material there are two exercises each participant has to resolve. These two exercises are a direct application of the knowledge acquired from the learning material. The reference material is optional to read, but the resolving of the exercises is mandatory. The reference material and the exercise differ in each experiment, because in the experiments we use different features of the languages. The reference material contains the necessary information to complete the experiment, but no more. This is to make sure that participants do not get confused by extra information that is not necessary for the experiment.

Post-Questionnaire Phase After both tasks are completed, *i.e.,* at the end of the experiment, the participant is asked to fill out a final form requesting for qualitative data about the experiment, to establish the opinion of the participants about the experiment, the different systems used and their results. This includes, amongst others, questions on the difficulty of the tasks, a comparison of both systems, a space for freeform feedback and if they would use this type of system again.

7.2 Controlled Experiment: Program Comprehension

The first experiment we performed aimed to measure the correctness and speed of using LRP against using SMACH. The complete programs for both tasks, questionnaires and scans of filled-in questionnaires are available to download at `http://bit.ly/2jtUvbD`.

7.2.1 Goal

In this experiment we answer the following research questions:

Q1 *Does live programming reduce the time necessary for understanding program code for robot behavior written as state machines, compared with a classical approach without using live programming?*

Q2 *Does live programming increase the correctness in understanding program code for robot behavior written as state machines, compared with a classical approach without using live programming?*

Live programming is a paradigm that has been proposed to accelerate the development of programs. An important task in developing programs is understanding existing code. However, there has been no extensive report published on whether live programming has an impact on program understanding, neither in correctness of understanding nor time taken.

The null hypotheses and alternative hypotheses for the previous questions are:

- $\boldsymbol{H2_{01}}$: Live programming does not impact the time required in understanding program code for robot behavior written as state machines.
- $\boldsymbol{H2_{11}}$: Live programming reduces the time required to understand program code for robot behavior written as state machines.
- $\boldsymbol{H2_{02}}$: Live programming does not impact the correctness in understanding program code for robot behavior written as state machines.
- $\boldsymbol{H2_{12}}$: Live programming increases the correctness to understand program code for robot behavior written as state machines.

7.2.2 Experiment Design

As mentioned in Section 7.1, we designed a controlled experiment using cross-evaluation, which requires two tasks per experiment. The tasks of this experiment are:

- **Task A:** The Turtlebot performs an object delivery service, where an object is sent to a given destination and the receiver can send an object back.
- **Task B:** The Turtlebot broadcasts a message to different locations, and the message receiver can request for clarification of the message.

Figure 7.1 is a screenshot of Task A in SMACH and Figure 7.2 is a screenshot of the same task in LRP.

We randomized the participation of the subjects in every work session (Work Sessions are explained in Section 7.1.7) while having a similar number of participants per work session. We designed 4 different work sessions. Each participant participates in exactly one of the four work sessions. The different work sessions are:

- **W1**: First Task A with LRP, then Task B with SMACH.
- **W2**: First Task A with SMACH, then Task B with LRP.
- **W3**: First Task B with LRP, then Task A with SMACH.
- **W4**: First Task B with SMACH, then Task A with LRP.

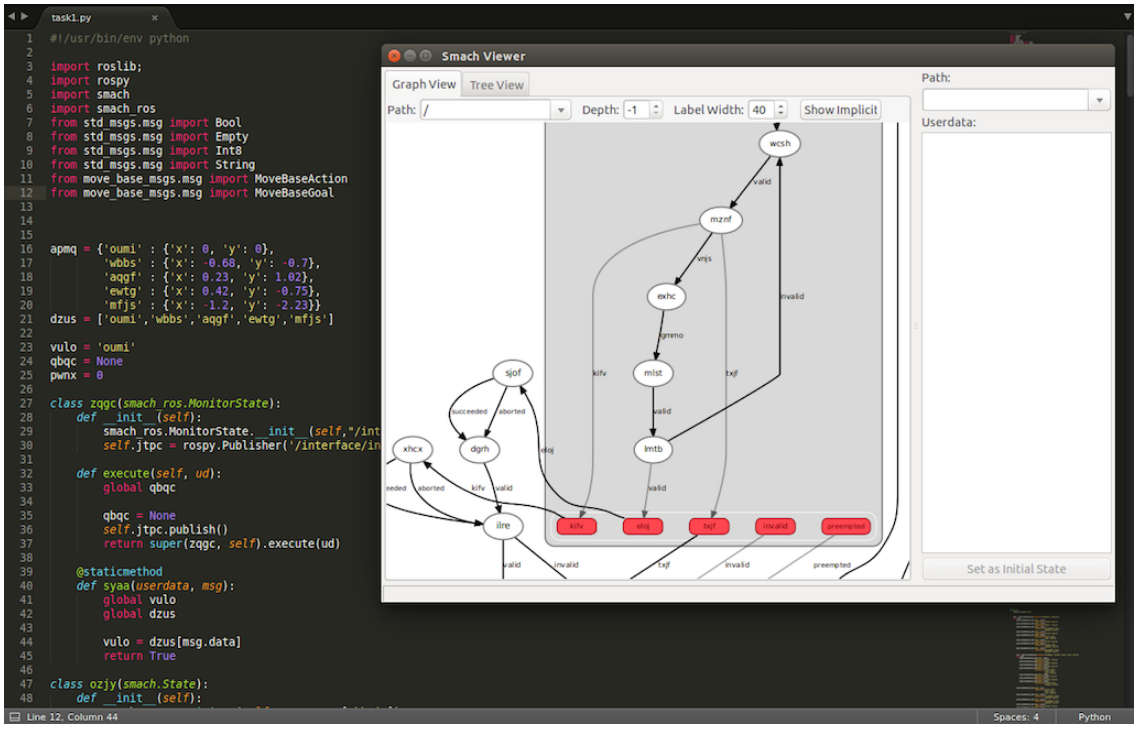

Figure 7.1: Task A of program comprehension experiment in SMACH.

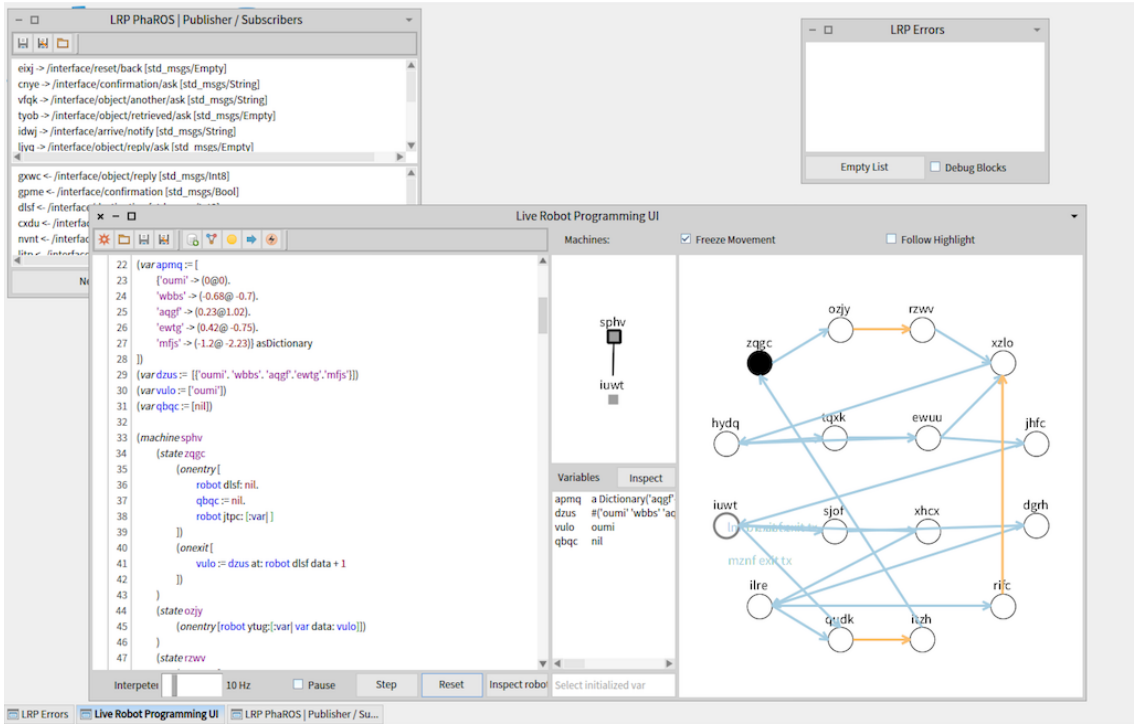

Figure 7.2: Task A of program comprehension experiment in LRP.

7.2.3 Pilot Study

We first conducted a first pilot study to assess the feasibility of the tasks and to uncover possible issues. We found that the test subjects considered the tasks to be too easy and some of the questions to be unclear. We modified the tasks and ran a second pilot study to confirm that this version of the experiment was clear and at an acceptable difficulty level.

In the first pilot we noticed a threat of a common development practice: If we give meaningful names to states, transitions, etc, the tasks were solvable only by using these names instead of reading the source code, executing the program or using other features of the systems. Even with this being a common practice in developing software, this adds a strong bias to the experiment that had to be removed. If we keep the meaningful names, we can not measure the system by their features, but only by their visual representation (which are very similar).

In the second pilot we used obfuscated names and noticed how the subjects did not try to understand the program by the names, but they did immerse themselves in the systems, using all available features of each system.

7.2.4 On Reducing Biases

We designed this experiment to reduce bias as much as possible. We found and reduced bias on the tested features and the name of the variables.

Reducing bias of difficulty from the task and used features The tasks were designed to use a wide variety of nested machine features without being trivial or too difficult. With this, we avoid a bias where developers would understand the programs too quickly or would not understand the programs at all.

Both tasks, while different, are built in a similar way. Both tasks have:

- A comparable number of states and transitions.
- Only one nested machine.
- States in which the program waits for a couple of seconds.
- Several states where the robot moves.
- Non-straightforward execution, *i.e.,* several paths and execution loops.
- Use of variables.
- Obfuscated names (explained in detail below).

We excluded functionalities that benefit the performance of one system over the other. For example:

- Concurrent behaviors: at the time the experiments where conducted, SMACH provided concurrent machines, while LRP did not.

- A transition from every state in the machine to one in particular: LRP provides such 'wildcard' transitions, while SMACH does not.

Reducing naming bias We decided to obfuscate the names of every entity in the system because we want the users to understand the program not just by looking at meaningful names. We see this as an important possible bias for program understanding and hence wished to remove it. For example, for the robot to move to a point in the map it needs to receive the exact point to which to move. This can be seen in the source code if we give a meaningful state name like *WaitingForDestination*: the developer can only look at the state name to understand that the robot is waiting to receive that information in that state. This is a problem, because to understand the program, the subject may just look at the meaningful names and not look at the source nor try to understand the program by running it or any other means, hence adding a bias in the experiment. The subject could just identify the names of the states and answer the questionnaire according to those meaningful names.

To avoid this problem, we replaced every meaningful name using a string formed by 4 random characters. We also decided to avoid using names like *stateX* to indicate states, for example. This is because the creation and use of states – and other program elements – in both systems are quite different. As before, we want the developers to experience this difference by not only looking at meaningful names, but looking at the program structure, further removing bias.

7.2.5 Work Sessions

For the evaluation we prepared one computer with two screens. One screen shows the simulated robot and the communication interface between the subject and the robot, we call this screen the *Robot Space*. The second screen shows the source code and extra features that the systems may provide, *e.g.,* a visualization. We call this screen the *Development Space*.

For the LRP system, the Development Space contains the IDE and a window showing all the communication channels used by the program to the robot. In the IDE the developer can see the source code of the program and the live visualization. For the SMACH system, the Development Space has the source code in a Sublime Text editor[4] and a terminal with two tabs. One of the tabs is ready with the command to run the program and the other tab is ready with the command to open the run-time visualization. When the visualization is open, it is also displayed in the Development Space.

7.2.6 Warm-up Phase

In this experiment, the warm-up phase consists in reading reference material and solving an exercise. In the exercise, the subjects extend the example presented in the reference material,

[4] `https://www.sublimetext.com`. Accessed: 04-07-2019.

focusing on several features that the subject needs to know before doing the experiment. These features are:

- Creating a state.
- Adding a state, *i.e.,* adding the necessary transitions to make the state reachable in the program.
- Sending data to the robot.
- Receiving data from the robot.

7.2.7 Evaluation phase

For each of the two tasks the subjects should answer a questionnaire that serves to measure how well they understand the program. Each questionnaire has 17 questions designed to cover the different strategies we expected the subject would use in a program comprehension task. Some examples are:

- What is the number of outgoing transitions from state X?
- In which state does the robot *do something*?
- What should happen for the robot to go from state X to state Y?

For example, to answer the last question, subjects could look at the source code. For LRP, they could look at the event and the transition that connects both states:

```
(state X (...))
(on hfjl X -> Y)
(event hfjl ["action code"])
(state Y (...))
```

In this example, the answer is given by the action code of the event named *hfjl*: if it returns `true` the transition will be taken. For SMACH, subjects could look at the transition of the state X that connects to state Y, then look at the state X to see what the state does to trigger this transition:

```
class X(smach.State):
  def __init__(self):
    smach.State.__init__(self,
                         outcomes=['itul', 'fjhd'])
  def execute(self,ud):
    qbqc = #do something to fill this variable
    if qbqc is None:
      return 'itul'
    else:
      return 'fjhd'
def main():
    sphv = smach.StateMachine()
    with sphv:
        smach.StateMachine.add('X', X(),
          transitions={'itul': 'Y', 'fjhd': 'Z'})
```

Part.	Work Session	LRP Score	SMACH Score	LRP Time (min)	SMACH Time (min)	LRP Exp	SMACH Exp
1	W3	16	16	100	76	Yes	No
2	W4	14	13	49	96	Yes	No
3	W1	16.5	15	47	33	No	No
4	W2	15.5	16	48	54	No	Yes
5	W4	16	15	55	60	No	No
6	W1	10	12	47	34	No	Yes
7	W2	15	10	45	55	No	No
8	W3	14	15	53	50	Yes	No
9	W3	15	15	67	45	No	Yes
10	W4	16	17	63	68	No	Yes
Average		**14.8**	14.4	57.4	**57.1**		
Median		15.25	15	51	54.5		
σ		1.792	2.010	15.787	18.229		

Table 7.1: Quantitative results of the program comprehension experiment.

To answer the previous question in SMACH, subjects should look at where state X is added, then see that the transition named *itul* is responsible for going from X to Y. Lastly, they should go to the definition of the state X and find out when the *execute* method returns the string *itul*.

We designed some questions that can only be answered by running the program. This is because we wanted the experiment to treat the complete running system such that some of the properties of live programming come into play. Without these questions, the questionnaire may be answered by looking at a static representation of the visualization and the source code, where it does not matter if the system is a live programming environment or not.

At the end of each task, we collect the questionnaire and note how much time it took to finish the questionnaire. When evaluating the answers, we counted how many right answers the subject had. There are some questions where subjects had to justify their answers. For questions where the answer is wrong, but the justification is correct, we added half a point.

7.2.8 Results

We have ten test subjects, all students from the engineering faculty of the University of Chile. All subjects are at least in their fourth year. Eight students are undergraduate students, 2 of them are graduate students. There are 6 students from Computer Science and 4 students from Electrical Engineering. Each experiment session took about four hours.

Two of the subjects state that they have a Beginner level of SMACH and 2 more state they have a Knowledgeable level of SMACH. Three of them state that they have a Knowledgeable level of LRP. The other subjects state that they do not have any knowledge of LRP or

SMACH, however, all of them state that they have knowledge of Python, a beginner level to advanced.

Quantitative Results

We present the quantitative data of the experiment on Table 7.1. The LRP and SMACH columns are the scores obtained on the questionnaire for the tasks carried out using LRP and SMACH respectively. The table presents the time it took to complete the tasks done in LRP and SMACH. The second column shows the type of work session of the subject. The last two columns indicate whether the participant has some experience using LRP and SMACH. At the end of this table we see the average, median and standard deviation of the data. These figures give a slight advantage to LRP in accuracy, but slightly faster speed on SMACH.

We analyze these results considering four data combinations: score and time between Task A and Task B, and score and time between SMACH and LRP. After completing this analysis we need to employ a statistical test to verify whether these differences are significant. First, we analyze the normality of the data, as this may impact the statistical test to employ. Running the Shapiro-Wilk test on our four data combination indicates that only two combinations are normally distributed: (i) the scores and times to complete Task B and (ii) the scores and times of SMACH. Since not all the data are normally distributed, we use the Mann-Whitney test to determine whether the set of values are different.

In Table 7.2 we first see the values of the Mann-Whitney, two-tailed test for LRP vs SMACH. In this table we present two values: the probability value P and the Mann-Whitney statistic value U. The P value allows us to test the statistical significance of the data, when $P < 0.05$, we claim that the data is significantly different with a confidence level 95%. The U value is a component of the Mann-Whitney test used to calculate the P value: the smaller the value of U, the smaller the value of P. In Table 7.2 we reported the U value for comparison when replicating the Mann-Whitney test.

Treatment	P value	U value	Difference?
LRP score vs SMACH score	0.6455	43.50	**No**
LRP time vs SMACH time	0.8970	48	**No**
LRP score: Task A vs Task B	0.9048	11.50	**No**
SMACH score: Task A vs Task B	0.3968	8	**No**

Table 7.2: Mann-Whitney, two-tailed test for several data combinations of the program comprehension experiment.

For both score and time $P > 0.05$, which means there are no significant differences. As a consequence, the observed differences are likely due to the participants and experiment setting, and cannot be related to the treatment of the experiment in SMACH or LRP. Moreover, the average and median of each of the four data combinations indicates that there is no tendency that could be increased by including more participants. We therefore conclude that the number of subjects in the experiment is sufficient to support our claim of there being no significant difference.

We have produced two tasks, A and B, completed in both LRP and SMACH. Lastly, we have compared the results of tasks A and B for each treatment, as seen in the last two rows of Table 7.2. This figure indicates that both task A and B are similar since $P > 0.05$.

Quantitative results: Participant with no LRP Experience

In Table 7.1 we see that 3 participants had experience in using LRP before the experiment was conducted. We analyzed the data without these 3 participants.

Here we use the same analysis as with the original results. Without taking account the experienced LRP participants, LRP still has a slight advantage in terms of correctness against SMACH. However, SMACH improves a lot its speed compared to LRP. We see these results in Table 7.3.

	LRP Score	SMACH Score	LRP Time (min)	SMACH Time (min)
Average	**14.857**	14.286	53.143	**49.857**
Median	15.5	15	48	54
σ	2.048	2.250	8.114	12.159

Table 7.3: Statistical results of the program comprehension experiment removing participants with LRP experience.

In Table 7.4 we see, without taking account participants with experience using LRP, that there is no statistical differences among score and time using LRP or SMACH of the completed tasks ($P > 0.05$).

Treatment	P value	U value	Difference?
LRP score vs SMACH score	0.5746	19.5	**No**
LRP time vs SMACH time	0.6911	21	**No**

Table 7.4: Mann-Whitney, two-tailed test for score and time of LRP vs SMACH for the program comprehension without taking into account participants with LRP experience.

Qualitative Results

We present the raw data of the subjects' opinions on Table 7.5. All the questions presented here have a score ranging from 1 to 5.

An important qualitative result is that every subject in the experiment thinks that it is easy to understand programs written in LRP, except for one. In contrast, with SMACH there are mixed opinions: 6 subjects think that it is easy to understand programs using this platform and 4 subjects think that it is hard. Moreover, the subjects also think that LRP is better for program understanding than SMACH, with an average result of 4. When asked if they would prefer to use one system over the other, LRP also has better results, with an average score of 4 against an average result of 3.1 for SMACH.

Part.	Is LRP better than SMACH?	Would use SMACH	Would use LRP
1	4	2	4
2	5	2	5
3	4	4	4
4	5	1	3
5	4	4	4
6	3	4	4
7	4	3	4
8	4	4	4
9	3	4	4
10	4	3	4
Avg	4	3.1	4

Table 7.5: Qualitative data of the program comprehension experiment. The worst result is 1 and the best is 5.

Of the ten participants, LRP got three positive comments claiming that it was easier to understand code (translated from Spanish): *"In LRP there are less concepts, so it is easier to understand"*, *"LRP was easier to understand"*, *"In LRP the code was easier to understand"*. There is also a comment about a specific feature in LRP that connects the visualization with the source code by clicking on an element of the visualization: *"The visualization of LRP has a nice integration with the source code"*. However, there are some positive comments about the SMACH visualization, where LRP falls short: *"The SMACH visualization is tidy and you can see everything, even the state of the nested machine"*, *"The SMACH diagram was very easy to follow, instead, the LRP diagram was harder to follow"*.

7.2.9 Observations on Subject Behavior

Important benefits of live programming are said to stem from immediately seeing the effects of program changes. In this experiment we however noticed one important behavior of the subjects: they did not modify the source code when trying to understand the program. Instead they mostly used other means, such as the visualization of the program at run-time or taking notes. It therefore makes sense that LRP has the same performance as SMACH. There is no live programming advantage, given that the subjects did not modify the programs at all.

It is important to know the results of a similar experiment where subjects actually behave differently, modifying their programs and actually using liveness. This requires a follow-up experiment.

7.2.10 Conclusions on Program Comprehension

The quantitative results of the experiment do not allow us to reject both null hypotheses $H2_{01}$ and $H2_{02}$: there is no difference between using LRP or SMACH to understand complete programs for robotic behaviors, both considering correctness and time taken. In contrast to this, the qualitative results of the experiment reveal that the subjects actually think that the answer of the research question is positive. Concretely, the subjects' opinions are that it is easier to understand programs written in LRP and moreover they prefer to use LRP over SMACH.

Put differently: while the users' opinions about the different systems confirm the tenet of live programming yielding a better developer experience, the quantitative data show that in this experiment the developers' performance is actually unchanged.

7.3 Controlled Experiment: Program Writing

The goal of our second experiment is to measure the accuracy and speed of writing a program for robot behavior using live programming. As mentioned in Section 7.1, we compare LRP with SMACH using a within-subjects design. The complete programs for both tasks, questionnaires and scans of filled-in questionnaires are available to download at `http://bit.ly/2jbixXD`.

7.3.1 Goal

We will answer the following research questions:

Q3 *Does live programming reduce the time in writing state machine programs for robot behavior, compared with a classical approach without using live programming?*

Q4 *Does live programming increase the correctness in writing state machine programs for robot behavior, compared with a classical approach without using live programming?*

Live programming has been proposed to accelerate the development of programs, which logically includes writing the program. Program writing in live programming has been studied before, though there are no in-depth studies that consider the domain of robot behaviors nor the state machine paradigm nor even with a complex API.

The null hypotheses and alternative hypotheses for the previous questions are:

- $\boldsymbol{H_{03}}$ **:** Live programming does not impact the time required in writing state machine programs for robot behavior.

- $\boldsymbol{H_{13}}$ **:** Live programming reduces the time required to write state machine programs for robot behavior.
- $\boldsymbol{H_{04}}$ **:** Live programming does not impact the correctness of the solution when writing state machine programs for robot behavior.
- $\boldsymbol{H_{14}}$ **:** Live programming increases the correctness of the solution when writing state machine programs for robot behavior.

7.3.2 Tasks to Evaluate

We used some elements of the program comprehension experiment explained in Section 7.2 and simplified them. In this experiment, each program is divided in 4 steps, where the first step (Step 0) is a warm-up session where subjects get used to the development environment. Each step represents a part of a complete program, by finishing the last step, the subjects complete the overall behavior. After the warm-up step, the next 3 steps have an increasing level of difficulty and come with a time limit: 10, 20 and 30 minutes each. The last step combines multiple functionalities that are used in the previous steps, adding new functionalities only presented in this step.

We present the two tasks here:

- **Task A:** In this task the Turtlebot greets someone in a place specified by a person.
- **Task B:** The Turtlebot should deliver a message to different, fixed destinations.

A comparative screenshot of the complete Task A in SMACH and LRP can be seen in Figure 7.3 for SMACH and Figure 7.4 for LRP. These images present the program and visualization of Task A for SMACH and LRP respectively.

We randomized the participation of the subjects in every work session (Work Session is explained in Section 7.1.7) while having similar number of participants per work session. We designed 4 different work sessions. Each participant participates in only one of the four work sessions. The different work sessions are:

- **W1**: First Task A with LRP, then Task A with SMACH.
- **W2**: First Task A with SMACH, then Task A with LRP.
- **W3**: First Task B with LRP, then Task B with SMACH.
- **W4**: First Task B with SMACH, then Task B with LRP.

7.3.3 Pilot Study

As before, we performed a pilot study before conducting the experiment. The most important lesson learned in this pilot was the amount of time needed to perform the complete experiment. We originally miscalculated, resulting in 5:30 hours in total. We had to perform 2 pilot studies to reach an acceptable time of 4 hours in total.

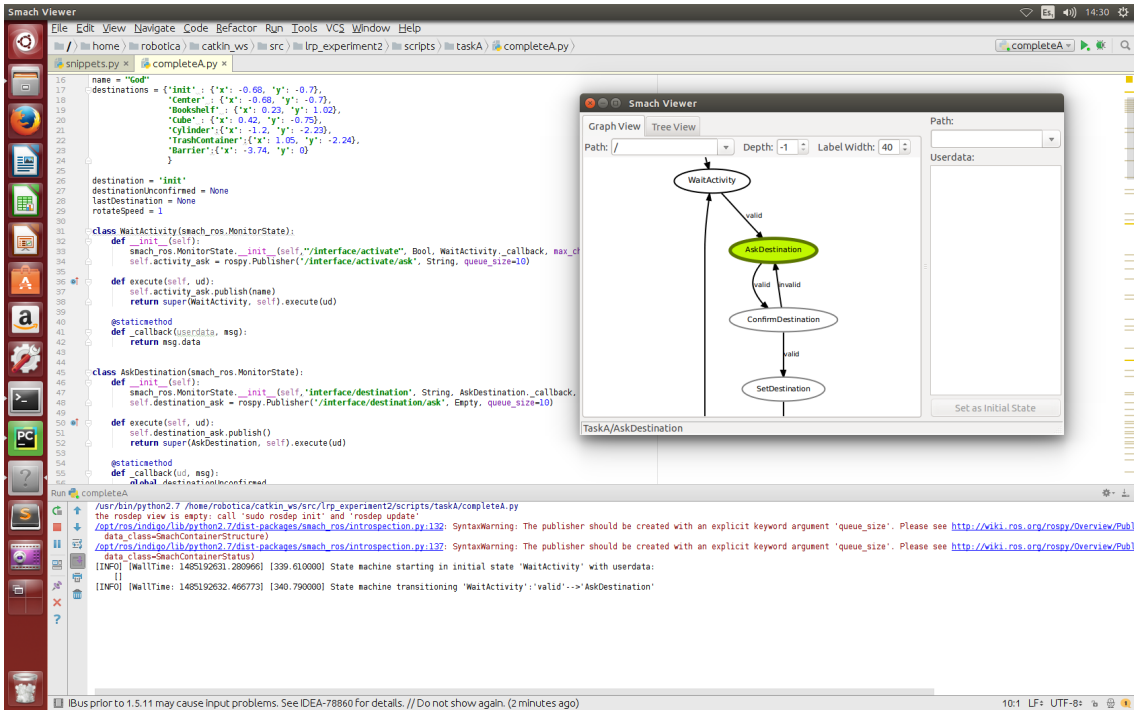

Figure 7.3: Complete program of Task A in SMACH for program writing experiment.

7.3.4 On Reducing Bias

We designed this experiment to reduce bias as much as possible. We reduced the bias of having two different tasks and reduced the bias of a subject not being able to end a step.

Reducing bias of used features

The tasks were built to not be too difficult, giving the subjects higher odds to finish every step, which reduces the bias of not having any results at the end due to a high complexity. We removed features that proved too complex to understand in the given span of time, like concurrent behaviors.

Also, the tasks were designed to use the same functionalities, while not building the same program. We included this property in every step, except in Step 0 which is exactly the same. This is to allow the subjects to understand the basics of each system in the same way, reducing bias.

Reducing different starting point bias

When building a step, every subject has the potential to build the program in many different ways. Since the program is added to in each step, it could end up differing greatly between subjects. If we measure steps using a different program as a starting point, we may introduce a bias. To avoid this, each step starts with a base program that is provided by us.

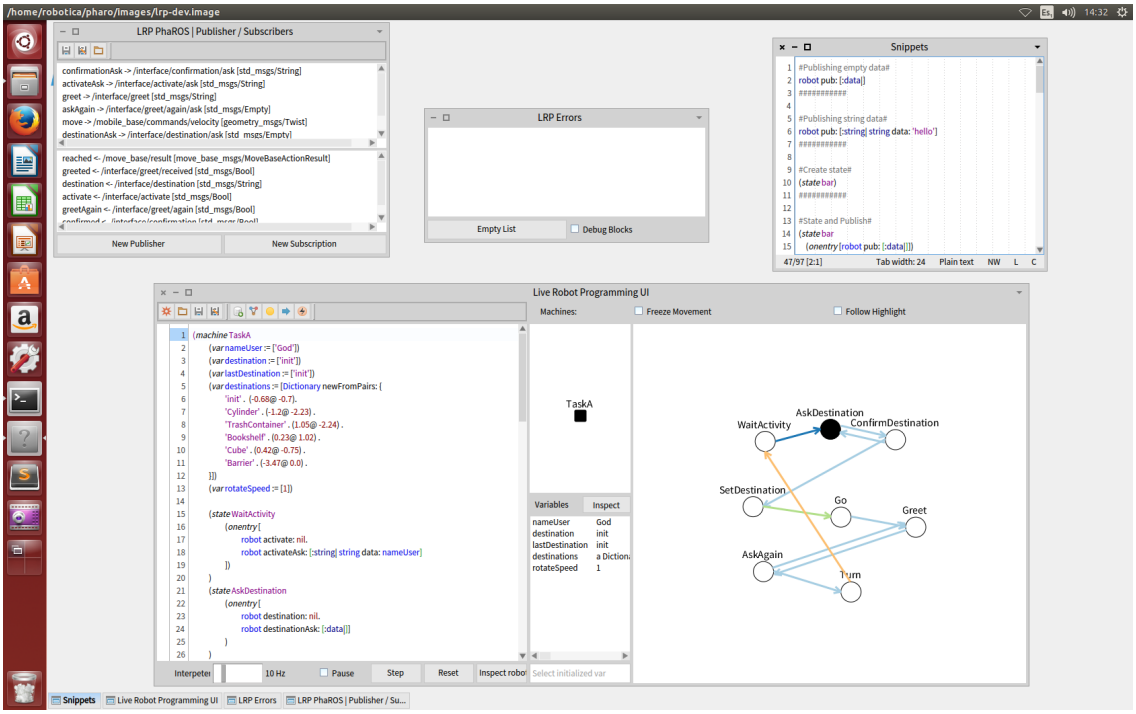

Figure 7.4: Complete program of Task A in LRP for program writing experiment.

By introducing a new program at every step, we may introduce a bias of understanding in the different systems. If the base program is more difficult to understand in one system, then the subjects may have an additional problem to extend it in this system. However, in our experiment of program comprehension, presented in Section 7.2, we have seen that there is no significant difference in program understanding between LRP and SMACH. We can therefore use this way of working without adding any significant bias.

Lastly, by giving a base program to start with, we can reduce bias when subjects do not finish a step. If a subject does not finish a step, the subject can still continue with the next step using the base program.

7.3.5 Work Sessions

For the evaluation of this experiment we prepared two computers, one with one screen and the other with two screens. The computer with two screens represents the robot environment: in one screen we have the simulation and the other we have the textual user interface of the robot.

The computer with one screen represents the development environment. For LRP, this screen contains the IDE to write programs, the error window and the window that creates the communication channels between the robot and LRP. Additionally, it contains another window with code snippets of recurrent code patterns in LRP. For SMACH, the development environment contains, in full screen, PyCharm, a Python IDE[5]. In PyCharm there are always

[5] `https://www.jetbrains.com/pycharm/`. Accessed: 04-07-2019.

two open tabs, one with the current program being written, and another with code snippets of recurrent code patterns similar to those in LRP. Finally, there is a terminal ready to execute the SMACH visualization.

7.3.6 Warm-up phase

In the warm-up exercise in this experiment the subjects should write a program in the system, instead of on a paper. We decided to go with this because this would be the first attempt of the subjects to write a program and use the system. This may introduce some beginner errors and we want to reduce such a threat in our evaluation.

As we explained earlier, the code the subjects should write is the program for Step 0, which is the same for both systems.

7.3.7 Evaluation phase

For both tasks, subjects should write a program divided in three incremental steps. Step 1 and Step 2 are guided steps in terms of the states and transitions the subjects should write. Step 3 is the most open step, with no guide to how many states and transitions the subjects should write, but descriptive enough for the subjects to understand what they should do. Step 3 features the same functionalities presented in the previous steps, but adding a behavior where the robot turns 180 degrees.

The turning behavior is an important addition because to the subjects it is initially unclear how to achieve the correct amount of degrees of turn. Test subjects need to experiment with different turn speeds or times. This exemplifies a scenario where developers are not sure which parameters an algorithm needs at the beginning and they need to fine-tune some parameters until they find the right values.

With every step we measure the correctness and the time to complete the step. Every step has a time limit: 10, 20 and 30 minutes respectively. When subjects could not complete all the steps, we scored only the correct parts they managed to write. Recall that in Section 7.3.4, we mentioned that at the beginning of each step every subject starts with the same base program, hence not completing a previous step does not introduce a strong bias in the results.

Each step has a different scoring function due to the difference in difficulty. The maximum points for each step are: 8, 11 and 19. Because the systems are different, it is not trivial to be fair in scoring each task. To do this we divided each program in smaller sections and scored them independently. The division is: sending data to the robot, receiving data from the robot and extra functionalities[6].

[6]A complete table of the scoring in each step for each task can be found in `http://bit.ly/2kpWGg5`

7.3.8 Results

Part.	Work Session	LRP Score				SMACH Score				LRP Exp	SMACH Exp
		S1	S2	S3	Σ	S1	S2	S3	Σ		
1	W3	6	7	16	29	7	6	7	20	Yes	No
2	W4	8	11	19	38	8	11	19	38	No	No
3	W2	5	11	19	35	7	7	19	33	No	No
4	W3	6	10	12	28	8	11	18	37	No	No
5	W4	8	11	19	38	6	11	10	27	No	No
6	W1	8	10	18	36	8	6	19	33	No	No
7	W1	6	11	19	36	7	11	19	37	No	No
8	W2	8	10	19	37	8	11	19	38	Yes	No
9	W4	6	11	18	35	7	4	13	24	No	No
10	W3	4	7	18	29	8	10	18	36	No	Yes
Average		6.5	**9.9**	**17.7**	**34.1**	**7.4**	8.8	16.1	32.3		
Median		6	10.5	18.5	35.5	7.5	10.5	18.5	34.5		
σ		1.36	1.51	2.10	3.7	0.66	2.6	4.23	6.1		

Table 7.6: Score for every step of the program writing experiment, for both LRP and SMACH. Σ represents the sum of scores of the three steps.

We have 10 test subjects who are all students from the engineering faculty of the University of Chile, but they are not the same from the previous experiment. All subjects are at least in their fourth year. Six of them are undergraduate students and the other four are graduate students. All of the students are from the Computer Science Department.

All the participants, except one, declared having no knowledge of SMACH, while 2 of them stated that they have a Beginner level of LRP. Notably, only 2 students declared not having any knowledge in robotic development. Moreover, all of the students declared to have at least a Beginner level of Python, with 2 of them declaring an Advanced level in Python.

Quantitative Analysis

Table 7.6 and Table 7.7 present the raw data. Table 7.6 gives the score for every step and the sum of the three steps. Table 7.7 indicates the time to complete each step and the total time to complete the three steps. If a subject did not finish the step, we show this with an "x". The sum column considers an "x" as the maximum time allowed for completing the task (S1 = 10 minutes, S2 = 20 minutes, and S3 = 30 minutes). For both tables, the last columns present if the participant has some experience using LRP and SMACH. Because it may not be fair to give the maximum time allowed when summing the time of the steps, Table 7.8 presents the normalized time of completed steps only.

In Table 7.6 and 7.7 we present the average, median and standard deviation of the data. With these figures we see a slight advantage to LRP for the sum score of the tasks. The average and median score for S2 and S3 is higher with LRP than with SMACH. For the sum

Part.	Work Session	LRP Time				SMACH Time				LRP Exp	SMACH Exp
		S1	S2	S3	Σ	S1	S2	S3	Σ		
1	W3	x	x	x	60	x	x	x	60	Yes	No
2	W4	10	18	23	51	10	20	30	60	No	No
3	W2	x	20	19	49	x	x	27	57	No	No
4	W3	x	x	x	60	10	20	x	60	No	No
5	W4	10	20	27	57	x	20	x	60	No	No
6	W1	10	x	x	60	10	x	25	55	No	No
7	W1	x	20	26	56	x	20	25	55	No	No
8	W2	7	x	19	45	9	18	19	46	Yes	No
9	W4	x	20	x	60	x	x	x	60	No	No
10	W3	x	x	x	60	9	x	x	59	No	Yes
Average		**9.7**	**19.8**	**26.4**	**55.8**	9.8	**19.8**	27.6	57.2		
Median		10	20	28.5	58.5	10	20	30	59.5		
σ		0.9	0.6	4.32	5.25	0.4	0.6	3.5	4.21		

Table 7.7: Time to complete the program writing experiment. The letter "x" means that a step is not finished within the time limit. Σ represents the total time to complete the three steps. Whenever there is an "x", Σ considers the maximum time per step.

time, participants completed their tasks slightly faster using LRP. This claim is true for each task, except for step S2, where time LRP and SMACH are tied. When comparing only the time of completed steps by normalizing the time between 0 and 1, we see that participants still completed their tasks slightly faster using LRP, as shown in Table 7.8.

Similar to the previous experiment, we measured the normality of the data before measuring the significance between the score and time values. Not all the data sets are normal. We therefore employ, as with the previous experiment, the Mann-Whitney test. Between the sum score and the sum time, there is no statistical significance of using LRP vs SMACH: $P > 0.05$ (In Section 7.2.8 we explained the P and U values of the Mann-Whitney test). Table 7.9 presents this data.

Similar to the program comprehension experiment, there is no statistical difference in score nor time for each step, despite slightly better values for LRP. Again, this is due to the experimental setting and it cannot be related to the treatment.

We also compared the score of tasks A and B for each treatment to measure if the two tasks are comparable, as seen in the last two rows of Table 7.9. These two test results indicate that both tasks A and B are comparable and similar in their difficulty, hence reducing the bias that one task could be more difficult than the other.

Quantitative results: participants with no LRP Experience

In the Tables 7.6 and 7.7 we see that 2 participants had experience in using LRP before the experiment was conducted. We analyzed the data without these 2 participants. In this case,

Part.	Work Session	LRP Time [0-1]			SMACH Time [0-1]			LRP Exp	SMACH Exp
		S1	S2	S3	S1	S2	S3		
1	W3	x	x	x	x	x	x	Yes	No
2	W4	1.00	0.90	0.77	1.00	1.00	1.00	No	No
3	W2	x	1.00	0.63	x	x	0.90	No	No
4	W3	x	x	x	1.00	1.00	x	No	No
5	W4	1.00	1.00	0.90	x	1.00	x	No	No
6	W1	1.00	x	x	1.00	x	0.83	No	No
7	W1	x	1.00	0.87	x	1.00	0.83	No	No
8	W2	0.70	x	0.63	0.90	0.90	0.63	Yes	No
9	W4	x	1.00	x	x	x	x	No	No
10	W3	x	x	x	0.90	x	x	No	Yes
Completed Steps		14			15				
Average		**0.886**			0.926				
Median		0.95			1				
σ		0.139			0.101				

Table 7.8: Table 7.7 with normalized times between 0 and 1. We consider only the completed tasks to calculate the average, median and σ.

Treatment	P value	U value	Difference?
LRP sum score vs SMACH sum score	0.7286	45	**No**
LRP sum time vs SMACH sum time	0.7636	46	**No**
LRP sum score: Task A vs Task B	0.0556	3.5	**No**
SMACH sum score: Task A vs Task B	0.8968	11.5	**No**

Table 7.9: Mann-Whitney, two-tailed test for several data combinations of the program writing experiment.

LRP still has a slight advantage in terms of correctness and speed against SMACH, while SMACH is more stable when measuring time. For each task, LRP has a slight advantage for S2 and S3, while SMACH has a slight advantage in S1, for both correctness and speed. Table 7.10 and Table 7.11 presents this data. Even in Table 7.12, with the statistical results of the normalization of the time between 0 and 1, LRP keeps its slight advantage over SMACH, and SMACH remains more stable with half the standard deviation.

As with the data including the participants with LRP experience, the Mann-Whitney test indicates that scores and time between LRP and SMACH are not significantly different. Table 7.13 shows the results of the test.

Qualitative Analysis

The raw data of the subjects' opinions is in Table 7.14. All the questions presented here have a score from 1 to 5.

	LRP Score				SMACH Score			
	S1	S2	S3	Σ	S1	S2	S3	Σ
Average	6.38	**10.25**	**17.75**	**34.38**	**7.38**	8.88	16.88	33.13
Median	6	11	18.5	35.5	7.5	10.5	18.5	34.5
σ	1.41	1.3	2.22	3.57	0.7	2.62	3.22	4.78

Table 7.10: Statistical results of the program writing experiment removing participants with LRP experience.

	LRP Time				SMACH Time			
	S1	S2	S3	Σ	S1	S2	S3	Σ
Average	10	**19.75**	**26.88**	**56.63**	**9.88**	20	28.38	58.25
Median	10	20	28.50	58.5	10	20	30	59.5
σ	0	0.66	3.82	4.12	0.33	0	2.18	2.11

Table 7.11: Statistical results of the program writing experiment removing participants with LRP experience.

	LRP Time [0-1]	SMACH Time [0-1]
Completed Steps	12	12
Average	**0.923**	0.955
Median	1	1
σ	0.114	0.067

Table 7.12: Normalized times of Table 7.8, while removing LRP expert subjects. The table shows the statistical results of normalized time of completed steps only.

Treatment	P value	U value	Difference?
Sum Score: LRP vs SMACH	0.6981	28	**No**
Sum Time: LRP vs SMACH	0.7476	28.5	**No**

Table 7.13: Mann-Whitney, two-tailed test for score and time of LRP vs SMACH for the program writing experiment without taking into account participants with LRP experience.

Part.	Is LRP better than SMACH?	Would you use SMACH?	Would you use LRP?
1	5	3	5
2	4	2	2
3	4	3	4
4	3	5	4
5	4	3	4
6	2	4	3
7	4	3	3
8	3	4	4
9	4	1	4
10	2	4	3
Avg	3.5	3.2	3.6

Table 7.14: Subjects' opinion of the program writing experiment. The worst is 1 and the best is 5.

Also in this experiment we get a positive qualitative result in favor of LRP. Six of the subjects think that it is easy to write programs in LRP, while the other four think it is hard. In SMACH five subjects think that it is hard to write programs, while the other five think that it is easy. It is important to notice that only two of the subjects answer that LRP is hard and SMACH is easy. Moreover, the subjects in general think that LRP is a better tool for writing programs, with a score of 3.6 of 5, while for SMACH the score is 3.2. While the results are not as strong as the experiment in program comprehension for this kind of result, LRP still gets an advantage over SMACH.

We asked the subjects if the liveness features of LRP help them write and debug programs. Notably, every subject answered affirmatively to both questions. Moreover, LRP and its liveness features has a lot of positive comments (translated from Spanish): *"LRP is good because it allows one to jump between states"*, *"(with liveness) it is easier to see which step causes the problem"*, *"the jump function is for finding out if a state is working correctly"*, *"(with liveness) I can focus on the local problem instead of the whole process"*, *"(with liveness) it is easy to change values and immediately see what happens"*, *"the program shows you what it is doing"*. There is also feedback to take into account when developing in LRP: *"The tool (LRP) seems good, but it is difficult to use in the beginning"*, *"I need to get used to the LRP syntax"*, *"It (LRP) seems like a good tool, however (...) it takes time to get used to it"*.

7.3.9 Observations on Subject Behavior

Though live programming has previously been found to help in the construction of programs, we discovered that it is not the case in this experiment. From our observations of the subjects we see that one important negative influence is the interface between the language and the robot: the ROS middleware, even though we ensured that the subjects are only minimally exposed to the complexity of ROS.

We noted that there was a number of uncompleted steps, even when we tried to make them accessible in this experiment. In particular, for 10 subjects in the LRP version: 4 completed S1, 5 completed S2 and 5 completed S3. For the SMACH version: 5 subjects completed S1, S2 and S3. The subjects that completed the steps are not necessarily the same between LRP and SMACH. Moreover, they are not necessarily the same between steps.

Yet, we noted that in almost every unfinished step of the experiment, the subject had a problem using ROS. All of these errors made by the subjects were incidental and did not treat the construction of the behavior itself, *i.e.,* building a state machine, but the errors reflected how complicated it is to use the robot API. In 21 of the 31 failed steps the subjects lost time due to problems with ROS or the API between ROS and the system. Moreover, in our observations we saw that subjects had more problems with ROS in the first task than the second task, indicating a learning effect. In addition, we saw that they had more problems in the first step, which is the easiest one in terms of programming, but also the first step in which they are exposed to ROS without any outside help.

Lastly, ROS is an arguably complex middleware and ecosystem, and we found that this difficulty prohibits the design of an experiment with bigger and more complex programs. This is because we would need to teach more about ROS and its API in Python and SMACH, and in LRP. One subject even commented on this: *"The [ROS API] should be explained earlier and separately for each tool"*. For this we would need more time for the experiment. However, given that the subjects already spent 4 hours on the experiment, it is not straightforward how to increase the time of the experiment.

7.3.10 Conclusions on Program Writing

Like the previous experiment, the data of the experiment does not allow us to reject the third and fourth null hypotheses $H2_{03}$ and $H2_{04}$. There is no difference between using LRP or SMACH to write programs for robotic behaviors, both considering correctness and time taken.

We believe that negative impact of the robot API minimizes the benefits of using live programming for this case. As in the previous experiment, the qualitative results of the experiment reveal that the subjects think that it is easier to write programs in LRP than in SMACH.

7.4 Discussion

This chapter delved deep into the liveness features of LRP that should help developers and addresses why we believe robotic behavior development is indeed a difficult task to only use live programming

7.4.1 Why should LRP perform better?

There are several works stating the advantages of live programming [13, 14] and how developers behave in these systems [14, 15, 12]. We designed LRP with liveness features and conducted our own set of experiments, however we did not find any statistical differences using live programming in robotic development, even when we strongly believe that LRP has several features that should help developers.

In particular, we list some of the unique features of LRP:

- Immediate connection between the source code and the running program in the robot itself.
- Meaningful feedback via the LRP interface:
  - Visualization of the running state in the context of the whole program.
  - Visualization of the value of all variables at all times.
- A UI to connect with ROS API to simplify the code written.

However, LRP has its own disadvantages as well:

- Learning curve: developers are not used to LRP nor liveness features. In particular, LRP uses Pharo Smalltalk, a relatively obscure language for developers.
- To use the ROS UI developers still need to know how to work with ROS in LRP.

It is important to take into account the disadvantages of LRP in the experiment. With the training material and the constant supervision of the researcher in both of the experiments, we do not believe the learning curve of LRP has a huge impact on the experiment. We believe the disadvantage of understanding the robotic system strongly impacts the results.

Moreover, the disadvantage of understanding ROS does not only apply to LRP, but also to SMACH. In fact, every API that works with ROS would have this problem.

7.4.2 Why does LRP NOT perform better?

We claim that the particular disadvantages of LRP are not enough to obscure the studied benefits of live programming. Why then does live programming not have positive impact on our experiments?

The recent work by Kubelka *et al.* [61] offers insights on why live programming does not work. They found that some developers do not use liveness features where these features should help them.

While we were performing our experiments we noticed some similarities with the work of Kubelka *et al.* Moreover, because of the specific setting of robotic behaviors, we found even more behaviors that play a role in our negative results, as described below.

Live programming does not help in all development tasks Considering program comprehension (Section 7.2), we already explained that live programming does not help because subjects do not modify the program to understand it. The root cause of this may be that only two of the subjects are accustomed to working in a live programming environment. Hence most subjects do not change a program in order to observe these changes in the behavior. However, the two subjects with experience did not make any change either, so there may be other factors at play as well.

Complex, low-level API In the experiment of program writing (Section 7.3), we explained how the impact of live programming may be reduced because of the use of the ROS middleware. We gave full details on this in Section 7.3.9. One important observation is that most of the failed steps in the program writing experiment where due to problem with ROS or the ROS API used in the systems (21 of the 31 failed steps). The ROS API is complex to use by itself.

Moreover, subjects had more problems in the first step of the task and in the first task completely, indicating a learning effect. We attribute this to the learning effect of using ROS because: (i) the first step is arguably the easiest, but also the first one in which they are exposed to the ROS API; and (ii) in the second task they already had less problems using ROS, even when they worked in a different system.

In LRP we use a UI to make the connections easier with ROS. However, developers still need to use these low-level connections to work with the robot. This may have a negative impact in our results even when developers are used to the ROS API.

Missing opportunities We observed that the subjects' result times could be improved by better using the features of the live environment. In other words, we spotted several missed opportunities where, if subjects used live programming, they would improve their times to complete a task. This may be happening because here also only two of the subjects are used to a live programming environment.

The missed opportunities are especially noticeable in Step 3 of each task - the most extensive and difficult one for the program writing experiment. Remember that this step includes the turning behavior, which exemplifies a case of where live programming is said to be advantageous (as explained in Section 7.3.7). We thought that subjects would skip over parts of the program to just test this behavior, using the "jump" feature of LRP (explained in Section 6.4.4)). Eventually, subjects did use the feature, but just four of them, only after they had already reached that part of the program through normal execution. This again reduces the impact of liveness features. Remarkably, in SMACH we encountered a different strategy to save time testing this behavior. Three out of ten participants set the turning state as the initial state of the machine to test it, minimizing the time needed to test this particular behavior. Moreover, even when they did not do that, the program was not time consuming enough to lead to a significant performance impact.

On a positive note, the subjects did use the liveness features in Step 1 and Step 2 of the experiment in program writing, although on a smaller scale. We believe these programs are

not big enough to notice a real impact of live programming, since code complexity is low and hence it is straightforward to build a mental model of this code. In our experiment we could not make the subjects build bigger programs because of time constraints. We did expect to see a significant impact of liveness in the last step, but as we explained before, we encountered other obstacles.

7.4.3 How to improve our experiments?

To test our hypotheses about how live programming may help in practice we need to tackle the previous considerations:

- People need to be trained to use the liveness features or they will miss opportunities
- API's elements outside the live programming system can have a strong negative influence on the advantage of liveness. We need to address this.

7.5 Threats to Validity

As in any experiment, there are several threats to validity in this work. We analyze four kinds of threats, as classified by Wohlin *et al.* [62].

Construct Validity This considers the validity of the construction of the experiment with regard to the obtained results and if the results really represent what we want to measure. Our main concern in both experiments is whether the tasks may benefit one system over the other by focusing on specific features. To avoid this, we defined the tasks to not use specific features. Furthermore, even while the tasks are artificially created, they were designed to solve possible real life problems with real applications for both experiments, such as delivering objects.

Another concern is the intrinsic difference between LRP and SMACH. They both are specially built for robotic behaviors, however they use different base languages (Python and Pharo Smalltalk) and LRP is a DSL while SMACH is an API. We supervised each part of the experiments to see how subjects behave in them. We noticed that the differences in the programming environment did not have a strong influence of the results. In Section 7.4.2 we gave our opinion on the influences of the results based on the subjects behavior on the experiments.

For the experiment in program writing we gave the subjects a base program to start each step (more on this in Section 7.3). This may benefit one system if the task of understanding the base program is more difficult on one of the systems. However, the results of the experiment in program understanding (explained in Section 7.2) show us there is no such difference.

Internal Validity This considers whether there are causal relations unknown to the researchers that may affect the independent variables. Because the within-subjects design of our experiments, the results are affected by a learning effect between tasks. We minimize this threat by designing the tasks to be similar, but not identical.

Also, time may introduce a bias since each session has a duration of approximately 4 hours. The subject may get tired, leading to worse results in the second task for each experiment. To address this, while the subjects were performing the tasks, we were supervising to check if they were performing worse over time, which was not the case.

Moreover, in the program comprehension experiment, we see this learning effect: subjects always took less time to complete the second task, though not necessarily with improved scores. However, as we randomize the subjects participation in 4 work sessions, the impact of the learning effect is minimized. We see this not only in our results (LRP and SMACH are similar), but also in the score of the same tasks: subjects do not improve nor diminish their performance between tasks. We believe they only get used to the dynamic of the experiment.

Finally, we believe the ease of building a robot behavior is influenced by the robot API. In this case, our opinion is that the ROS API factor minimizes the effect of using live programming (as explained in Section 7.4.2).

External Validity This considers whether the experiment can be replicated and if the results can be generalized. Our most significant issues are the type of the participants and their previous knowledge. We had only students in both of the experiments. The reason is that robotics is a research area where a large amount of work is done in universities with students of different areas of knowledge and different degrees, *i.e.,* exactly the type of users in our study. We do not claim that this experiment could not be generalized. Even while we lack another group of subjects, we believe the results of using students is a good measurement of developers in the robotics world.

We are also aware of the problem of using students from our own campus. The quantitative results of experiments do not show a problem with this, however, the qualitative results may be influenced from the origin of the subjects. Even if the subjects are all from our campus, we claim the qualitative information will be replicated even with subjects outside our campus because developers recognize the importance of software engineering tools when developing programs in general. If we compare LRP against SMACH, LRP is a complete IDE to develop robotic behaviors with liveness features, unlike SMACH that provides an API and an external visualization.

The effect size is a threat to validity too. The lack of participants (10 per experiment) may lead to a lack of generalization of our results. However, even with the small amount of participants no difference is perceived, not even a trend. We do not believe increasing the amount of participants will change our conclusions. Live programing does not help with everything, nor everyone and it would have problems with more complex API (as explained in Section 7.4.2).

Also, the complexity of the settings may introduce a problem to other studies when trying

to replicate or generalize these results. To minimize this, we use a simulator of a real robot to remove the bias of working with real hardware and to make the experiment easier to replicate by just using computers, without the need of a real robot.

Reliability This considers whether or not the researchers influenced the results. We argue that if the experiment is conducted using the provided format, the results will be similar, no matter the person conducting the experiment. Getting the results in both experiments is a straightforward process: in the first experiment the subjects must answer a fixed questionnaire. In the second experiment they must write a program that is scored by a given correctness table. Both experiments do not need the presence of the researchers to measure the results. However, a presence of the researcher is fundamental for understanding the conclusions of why live programming may not help.

7.6 Related Work: Live Programming Experiments

To the best of our knowledge, there has yet been no extensive report of a study of the concrete advantages of live programming for understanding existing code as well as program writing in a practical setting, such as robot behaviors. Our related work is restricted to live programming systems that have presented some form of user study and studies that consider code creation (but not code understanding).

As part of the validation of Interstate Oney *et al.* [13] performed a comparative laboratory study where Interstate was tested against JavaScript. This study had 20 participants and consisted of 2 tasks where participants needed to make modifications and express new behaviors. There is however no description of how the experiment was conducted, nor how the tasks were divided amongst the participants. The conclusions of the experiment were that Interstate is faster than JavaScript at modifying already existing programs and expressing new behaviors. There is however no report of a study that measures the accuracy or speed of Interstate in understanding existing code.

The work of Wilcox *et al.* [14] revealed that continuous visual feedback in direct manipulation of programs helps in the accuracy of debugging certain tasks. They compare a live version of Forms/3 [63], against another version with immediate feedback removed. In this experiment there are 29 subjects, where half of them work on two different tasks using first the live version and then the non-live version, while the other half do exactly the opposite. They use two completely different tasks for this study - one to emphasize a graphical program, and the other to emphasize a mathematical program. The degree of improvement, as the authors conclude, depends on the type of problem, the type of user, and the type of bug.

Kramer *et al.* [15] complemented the work of Wilcox *et al.* [14] by analyzing code creation, comparing a live version against a non-live version of JavaScript. In this work the authors analyzed 10 subjects where each subject solves three different tasks: one to parse an RSS feed, one to convert between two object representations of a date, and one to implement Dijkstra's algorithm. The authors noticed that while live programming did not speed up the

time to complete a task, it did significantly decrease the time for fixing bugs introduced while writing the task. They state that they found no indication that live programming speeds up the process of code creation because of the small sample size and/or that the data is overshadowed by inter-subject differences.

In addition to the previous mentioned small studies on whether live programming improves the speed in development, there are three studies on how developers behave in a live programming environment [14, 15, 12]. These show that developers interact more with the live systems by performing more changes in programs or by regularly checking the code for correctness. The rationale is that this is because these systems promote more interaction between developers and their programs. The recent work of Kubelka *et al.* [61] also shows how developers behave in a live programming environment, including more practical tasks of fixing bugs in a real-unknown system and extend a familiar system using liveness features.

Lastly, Hundhausen and Brown [64] investigated the impact of continuous feedback on novice programmers. To do this each subject was exposed to three different systems: no feedback, self-select feedback and automatic feedback. In the first system the subjects mentally simulated the program, in the second the subject explicitly requested syntactic and semantic feedback, and the third system the subject had feedback with every keystroke. The subjects completed three tasks that involved creating, populating and iterating over arrays. The authors found that even when subjects did significantly better in the systems with feedback, there was no difference between both feedback treatments. Note that liveness implies continuous and meaningful feedback [12] and that is one of the conclusions of the work of Hundhausen and Brown [64]: *"rather than coming up with ways to facilitate liveness (in terms of feedback), programming environment designers ought to be putting their efforts into designing effective semantic feedback that benefits users"*.

None of the previous experiments focus on the impact of live programming on robotic behaviors nor on the use of a more complex system, except for the work of Hancock [12] that focuses on teaching children to program robot behaviors. However, Hancock only presents a study on how children behave using his system in robotics, with no comparative analysis on the impact of live programming in the development of robot behaviors. Moreover, the system that connects with the robot is much less complex than using ROS, which may explain why this work does not report on the problems we explain in Section 7.4.

7.7 Conclusions

In this chapter we reported on two controlled experiments that gauge whether, in the practical settings of robot behaviors, code understanding and code writing in a live programming system is more accurate and faster than a non-live setting. To the best of our knowledge, this is the first such in-depth study on the advantages of live programming in a complex domain, such as robotic behaviors.

The quantitative results of the experiments show us that there is *no significant* difference in program understanding nor in program creation, both considering correctness and time

taken. With these experiments we deny our third hypothesis *H2*: LRP does not improve the speed and accuracy of program understanding and program writing in a behavior-based context. This novel result seems to contradict the previous works in live programming.

From observations of test subjects, we learned three important aspects that reduce the impact of live programming on our experiments:

Complexity of robot API. The robotic API between the language and the robot is a strong influencing factor in the result. Many issues in program construction arose because of the complexity of the API, which negatively impacted the distinction between the two systems under testing.

Missing opportunities. We observed that the subjects do not always use the live programming features to their fullest, leading to a notable amount of missed opportunities for live programming to positively impact the results.

Limited scope of live programming. In particular in the experiment on program comprehension, we noticed that live programming does not help because subjects did not change the program. This limits the effects of live programming to a simple run-time visual feedback of the program.

We believe that the contradictory results between our work and the related work is due to the previous statements. We need to perform further studies on each statement and measure its influence on the benefits of using live programming.

Moreover, the qualitative results contradict our results: the subjects' opinions in the experiments are that it is easier to develop in a live programming system. The apparent contradiction of our results with the subjects' opinion is remarkable and merits further study.

Chapter 8

Conclusions

In this chapter we summarize the contributions of this dissertation. We also summarize the main concepts of study of this work: giving an effective mapping of the state machine model to developers to understand and build robotic behaviors. We emphasize future work before giving our final remarks.

8.1 Contributions

This dissertation contributes to the field of robotic behaviors by using software engineering techniques. In particular, we applied and evaluated improvements of the programming environment in behavior-based robotics development using nested state machines.

These improvements focus on fast understanding of robotic behaviors built using nested state machines. With different tools and techniques, we map developers' mental models to the source code of their programs. In the case of behavior-based robots, the programs are represented using the model of nested state machines.

To achieve this mapping, we first use meaningful feedback to help developers understand how the robot is behaving. Then, we apply immediate connection to allow changing of the program to be reflected in the behavior of the robot on-the-fly. With the combination of both we achieve live programming in robotic behaviors.

We divide the contributions of this dissertation in two main tools: VizRob and LRP. With VizRob we first studied how important it is to have meaningful feedback in behavior-based robotics. With LRP we studied live programming (meaningful feedback + immediate connection) in robotic behaviors integrated with a complex robotic API.

In this section we summarize the main features of each tool: VizRob and LRP. We then focus on answering the research questions and evaluating the hypotheses we declared in Section 1.4 and Section 1.5.

8.1.1 VizRob

VizRob is a tool that offers several visualizations to debug robotic behaviors using nested state machines. This tool addresses the necessity of robotic behavior developers to clearly understand how the robot is behaving and why.

For easy integration, VizRob uses information that is already presented in most behavior-based programs: the nested state machine definition, execution time and log information. Developers should not worry on tagging or adding code to their already written behaviors. This information is easily visible in the visualizations, where developers can see and understand the most important parts of their programs based on execution time and quantity and severity of logs.

We tested VizRob using a small case study of six robotic software engineers. In this preliminary study we find that developers were successful in finding errors in robotic behaviors and even they suggested solutions, without the need for looking at the source code. Developers perceived VizRob positively and they wanted to adopt it in the creation of their behaviors. A full explanation of VizRob and its small scale case study are detailed in Chapter 4.

8.1.2 LRP

LRP is a live domain specific language (Live DSL) to program nested state machines. The development of robotic behaviors is typically written, compiled and then deployed on a robot for testing. If the robot is behaving faulty, these steps are repeated until the desired behavior is achieved or seemingly achieve. Live programming removes these steps by giving a constant and meaningful feedback of the program to developers, while they have an immediate connection with the program, allowing them to make changes on-the-fly on the behavior of the robot.

One important conclusion of LRP it is that nested state machines are robust in the context of live programming. Programs can be modified without the need of restarting the program except for one case: when a change on the source code produces a modification of an running state. In all other cases, the robot can keep running the behavior while processing new changes.

We ran several experiments to study the impact of LRP to understand and write robotic behaviors. As far as we know, we conducted the first experiments to test if live programming improves developers productivity in the complex scenario of developing behavior-based robots. However, we did **not** find empirical evidence that LRP improves the development of robotic behaviors.

Regardless, subject opinions differ from the evidence. They believe LRP improves their performance on understanding and writing robotic behaviors. We have several insights on why LRP does not actually improve the development on robotic behaviors.

Chapter 6 explains the implementation of LRP. Chapter 7 details the experiments we

conducted to measure the impact of LRP.

8.1.3 Research Questions & Hypotheses

We built VizRob and LRP to answer the research questions presented in Section 1.4 and to validate the hypotheses presented in Section 1.5.

Answering RQ1: In the special context of behavior-based robotics, how the behavior of a program using nested state machine can give a meaningful feedback to developers?

With VizRob we can answer this research question in the context of developing robotic behaviors using nested state machine. To give a meaningful feedback of a program to a developer, it is not enough to only show a static visualization of the running program. A meaningful visualization should use also information of the execution of the behavior itself.

Several visualizations of nested state machines show also the current state of the running program. However, a running program is more than the part that is executing, the are many other factors involved, such as variables of the program or log statements. These factors are important and they need to be presented to developers somehow.

In the case study we performed we noticed how important were logs to understand why a behavior is executing faulty. Also we found that putting logs in the context of the executing of nested state machines was also important, errors may raised way before the actual execution of the faulty behavior. Putting the logs in the context of the execution also showed participants that, even in seemingly correct behaviors, there were problems internally in the behavior that are not being explicitly manifested. When testing in an isolated environment, those errors may not seem important, however in more intricate situations where different behaviors are interacting between each other, these errors may trigger.

We address the importance of visualizing other factors, such as variables of the running program. However, we also address that developers may not be willing to adopt a tool that require heavy changes in their source code, such as tagging variables or adding source code. That is why we choose to use logs with VizRob. Developers normally use logs to debug their programs and they are already presented in behavior programs. This makes it easy for developers to adopt VizRob in the future.

Answering RQ2: Is it possible to reduce (or remove) the cognitive distance using a practical solution in a behavior-based robotics context?

To answer this question, we develop LRP. Live programming states to reduce the cognitive distance between developers' mental model and their programs. LRP is a live programming language to develop behavior-based robotics using nested state machine. LRP can be used in any robotic systems that built a connection between the robot and the language, making LRP a practical solution to develop robotic behaviors. With LRP we can positively answer this question.

By answering this question positively, we also accept our first hypothesis: *H1: It is possible to have a live programming language for nested state machines in a robotic context.*

Answering RQ3: If the reduction (or removal) of the cognitive distance is possible, does this improve the developers productivity of behavior-based robots?

With LRP at hand, we conducted several experiments to answer this question. However, we found not positive answer to this question, LRP does not improve developers productivity.

In our experiments we measure speed and accuracy on program understanding and writing of robotic behaviors using LRP against a non-live system. In these experiments, we do **not** find a significant difference between LRP and a non-live system to develop robotic behaviors.

Nevertheless, we acknowledge the possibility that LRP is not the right tool to answer this question. Maybe the model of nested state machines is not the right one for behavior-based robots using live programming, or maybe the liveness features of LRP are not enough to measure an improvement. For now, we answer negatively to this question, but we give several possibilities on why this is happening in Section 8.4.

By answering this question negatively, we reject our second hypothesis: *H2: Live programming improves the speed and accuracy of program understanding and program writing in behavior-based robotics compared to a classical programming language paradigm (*i.e., *non-live)*

8.2 Why Meaningful Feedback Matters When Building Robotic Behaviors

When developers build programs, they should know when the program is working correctly. In the case of behavior-based robotics, it is easy to see if the robot is behaving correctly or not by just looking at the robot in the real world.

However, when we do not have feedback from the program and the robot, we encounter two problems:

- If the robot is behaving incorrectly, the cause of the error may not be apparent.
- The robot seems to behave correctly, but it is actually behaving incorrectly, though we perceive that everything looks fine.

The second statement can be especially troublesome: we believe that the robot is behaving right when actually it is not. It appears not to be a problem, unless we test it again and then notice that the robot is behaving poorly. Robotics is a discipline surrounded by uncertainty: the environment where the robot interacts is inherently unpredictable, sensors of the robot are limited to their ranges and resolutions, and even the actuators of the robot itself are affected by wear-and-tear [65]. It is not uncommon for the second problem to actually happen.

In the context of robotic behaviors, where we have several steps between thinking of a solution and actually seeing the behavior of the robot (see Section 1.3), it is important to know what is happening and why. If the robot is behaving incorrectly, developers need to know why as soon as possible. Because the development process of robotic behaviors has a particularly long cognitive distance, developers need to repeat the development process as few times as possible. If the robot seems to behave right, but it is actually failing, developers need to know this so they can address the error.

That is why we built VizRob: to help developers quickly understand what is happening with the behavior of the robot and why.

8.3 Why Live Programming Matters in Robotic Behaviors

Mapping the behavior of the robot to the source code can be done by giving meaningful feedback of the program to developers. Even when this may reduce the number of times developers go through the development process, this development process still has a large cognitive distance from developers' mental model to the behavior of the robot itself. Live programming aims to reduce or even remove this cognitive distance. To do this, it does not use only meaningful feedback, but combines it with immediate connection to the program itself. This allows for fast changes to the program on-the-fly.

As stated in Chapter 2, behavior-based robotics are based on building robots in the real world, with real and active interactions and testing them in the real world [3]. Live programming may have a huge impact in this process.

LRP is a general solution for building robotic behaviors using nested state machines with live programming. However, our experiments do not show that LRP actually improves development, even when developers themselves think that LRP helps them (as discussed in Chapter 7).

8.4 Why LRP Does Not Improve Robotic Behavior Development

Section 7.4.2 presented several insights on why LRP does not improve development. In particular we divide them in 3 aspects:

There are no silver bullets. There are no techniques that can help in every context. In our experiment of program understanding, we notice that participants do not change the program when trying to understand the behavior of the robot. This diminishes the impact of liveness for the development process of understanding a program.

Missing opportunities. For the experiment on program writing, we noticed that subjects do not use liveness features in tasks where they could improve their time or correctness in writing the behavior of the robot. We spotted several missed opportunities to use live programming. We believe this happens because subjects of the experiment are not used to working with live programming. It might not be common to have liveness features in the robotic development. This is particularly interesting: if developers need to get use to live programming and developers in the robotic context are, mostly, not experienced with live programming, then live programming would not help them right away, unless they practice.

Complexity of the based system: ROS. Finally, we believe the complexity of the robot API, in our case ROS, diminishes the advantages of using live programming in LRP against not using live programming with SMACH. We noticed that, even with our best efforts of obscuring the difficulty of using ROS, developers made a lot of mistakes just by not understanding the ROS API. Even if we teach ROS beforehand, it requires too much explanation because of its inherent complexity, prohibiting the design of an experiment with bigger and more complex programs in a reasonable time.

8.5 Future Work

We divided the future work of this dissertation in three main sections: VizRob, LRP and LRP validation.

8.5.1 VizRob

Even with the feedback already provided by VizRob, it still lacks features that developers believe would be useful for the development of their behaviors:

Live Visualization. For now, VizRob collects and shows the data when the recollection of data is completed. It is important to show the data while the program itself is running to spot possible errors earlier. Even more, it is important to analyze the data at any given point of the execution.

Multiple Machines per Visualization. LRP allows concurrency by executing multiple machines in the same state. VizRob supports concurrent behavior only when multiple states are being executed inside one machine. VizRob needs to support the visualization of multiple machines to show the concurrence case of LRP.

Visualization of Variables. Logging variables is a common debugging practice in software. An important feature to better understand what is happening is to visualize how these variables change while the program is running.

We can then expand the validation of VizRob with a complete experiment to measure the real impact of VizRob in the understanding and development of robotic behaviors.

8.5.2 LRP

LRP allows us to answer positively our second research question and, with that, we can accept *H1*. Nevertheless, LRP can be improved in different aspects:

Language Itself. In some robotic systems, such as JetStorm (see Section 6.5.2) the command to move a motor needs to be sent just once, and to stop that motor, another command needs to be sent to stop it. The action of starting a motor is then typically placed inside an onentry action and the action to stop the motor is placed inside the onexit action in the same state. As a result, when the program is paused, while the motor is running, it will keep running. To solve this, we are considering adding a global action whenever a pause is made. If this action commands to stop all the motors it will solve the previous problem, yet it is not immediately clear how to proceed whenever the program is resumed.

Meaningful Feedback. VizRob gave a lot of useful visualizations for improving the mapping of developers' mental model with the behavior of their robots. LRP can include these visualizations as well: to keep onwardly pushing the liveness feature that LRP already has with more meaningful feedback.

Liveness Features. Arguably one of the most striking features of some live programming languages is the ability to play with time, reversing or fast-forwarding program execution, for example as presented in the keynote of Bret Victor [7] in the videogame example. We would also like to introduce the time variable into programs. When programs are paused, developers could then control the time variable in their programs. They could save a part of the execution of the program, change variables, events, transitions, etc., and replay the program to analyze how this part of the execution changes. This would give more control to developers over the actions of the robot without re-executing the code every time they change something. These will improve the liveness features of LRP, improving the mental mapping with it.

Moreover, it is important to address how changes in the running program may affect the robot being working in a real, changing environment. For example, if we decide to change the execution of a pick-and-grab program while the robot has the object in its gripper, we also may need to remove the object manually, if the modified program expected that the robot has nothing in its gripper. We also have safety measures we need to attend to, such as stated by Lim [11]: a live editing of the program may damage the robot or the environment. These types of problems are new when we introduce live programming in robotics and it is important to address them.

8.5.3 LRP Validation

Even when we reject *H2*, in Section 7.4.2 we offered several insights into why we got that result: complex robot API and missing opportunities. Before conducting another experiment, we need to confirm these insights with more studies. Depending on the results of that study, we can design and conduct an improved experiment to test *H2* again.

Moreover, we can continue our line of experiments by testing LRP in another important development practice: debugging robotic behaviors. Maybe the same problems will arise in debugging robotic behaviors: a complex API would obscure the advantages of live programming. That is why we need to understand first how a complex API reduces the positive impact of developing robotic behaviors using live programming.

8.6 Final Remarks

In this dissertation we research the improvement of behavior-based robots development by applying software engineering techniques. In particular, we focus our improvements on fast understanding of robotic behaviors built using nested state machines.

We use two techniques for our purposes: meaningful feedback and live programming. When we researched our hypotheses, we noticed the potential of both techniques and how they can improve the development of robotic behaviors. While we only have positive insights on meaningful feedback using a small preliminary study, we believe the development of robotic behaviors can be significantly improved with the right tools that provide meaningful feedback. In the future we expect to improve VizRob and test our hypotheses in a more complete experiment.

Furthermore, even if our experiment did not confirm the hypothesis that LRP, with its liveness features, improves the development of robotic behaviors we still achieve an important goal: this is one of the first experiments on live programming in complex systems. We also offer several insights on why this is happening (*e.g.,* complex robot APIs). Even when live programming is not a new concept, it is just gaining popularity and has the potential to change how robotic development works.

The world of robotics has improved thanks to software engineering techniques. With middleware such as ROS, there is ready to use software development for several robots, and researchers can share their code to any other researchers who may want to try it in their own robots. However, we still have a long way to go as developing software for robots is still an arduous process. There are techniques, tools, frameworks and much more in the world of computer science that can be used in robotics to reduce the burden of developing software for robots. Nevertheless, it is not enough to adapt every technique we see, but we must also measure the real impact of these techniques in the world of robotics. To improving the development process of software for robots, we will continue to contribute in this area, mapping how the robot is behaving with the developers' mental model in different ways, such as useful visualizations (*e.g.,* VizRob) and fast connections with their programs (*e.g.,* LRP).

Bibliography

[1] D. Brugali, E. Prassler, Software engineering for robotics [from the guest editors], IEEE Robotics & Automation Magazine 16 (1) (2009) 9–15.

[2] R. C. Arkin, Behavior-based robotics, MIT press, 1998.

[3] R. A. Brooks, Intelligence without representation, Artificial intelligence 47 (1-3) (1991) 139–159.

[4] M. Loetzsch, M. Risler, M. Jungel, Xabsl-a pragmatic approach to behavior engineering, in: Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on, IEEE, 2006, pp. 5124–5129.

[5] A. Topalidou-Kyniazopoulou, N. I. Spanoudakis, M. G. Lagoudakis, A case tool for robot behavior development, in: RoboCup 2012: Robot Soccer World Cup XVI, Springer, 2013, pp. 225–236.

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

[7] B. Victor, Inventing on principle, Invited Talk at CUSEC'12, video recording available at `http://vimeo.com/36579366`. Accessed: 04-07-2019.

[8] S. L. Tanimoto, Viva: A visual language for image processing, Journal of Visual Languages & Computing 1 (2) (1990) 127–139.

[9] J. Wienke, S. Wrede, Results of the survey: failures in robotics and intelligent systems, arXiv preprint arXiv:1708.07379.

[10] P. Schillinger, S. Kohlbrecher, O. von Stryk, Human-Robot Collaborative High-Level Control with an Application to Rescue Robotics, in: IEEE International Conference on Robotics and Automation, Stockholm, Sweden, 2016.

[11] J. Lim, Live programming for robotic fabrication, Journal of Professional Communication 3 (2).

[12] C. M. Hancock, Real-time programming and the big ideas of computational literacy, Ph.D. thesis, Massachusetts Institute of Technology (2003).

[13] S. Oney, B. Myers, J. Brandt, Interstate: a language and environment for expressing interface behavior, in: Proceedings of the 27th annual ACM symposium on User interface software and technology, ACM, 2014, pp. 263–272.

[14] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, C. R. Cook, Does continuous visual feedback aid debugging in direct-manipulation programming systems?, in: Proceedings of the ACM SIGCHI Conference on Human factors in computing systems, ACM, 1997, pp. 258–265.

[15] J.-P. Kramer, J. Kurz, T. Karrer, J. Borchers, How live coding affects developers' coding behavior, in: Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on, IEEE, 2014, pp. 5–8.

[16] M. Campusano, J. Fabry, Live robot programming: The language, its implementation, and robot api independence, Science of Computer Programming 133 (2017) 1–19.

[17] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, ROS: an open-source Robot Operating System, in: ICRA workshop on open source software, Vol. 3, 2009, p. 5.

[18] The LEGO group, LEGO MINDSTORMS Education EV3, `https://www.lego.com/en-us/mindstorms`, Accessed: 04-07-2019.

[19] M. Campusano, J. Fabry, A. Bergel, Live programming in practice: A controlled experiment on state machines for robotic behaviors, Information and Software Technology 108 (2019) 99–114.

[20] M. Campusano, A. Bergel, VizRob: Effective visualizations to debug robotic behaviors, in: 2019 Third IEEE International Conference on Robotic Computing (IRC), IEEE, 2019, pp. 86–93.

[21] R. Murphy, R. R. Murphy, Introduction to AI robotics, MIT press, 2000.

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

[23] S. Thrun, W. Burgard, D. Fox, Probabilistic robotics, MIT press, 2005.

[24] D. Harel, Statecharts: A visual formalism for complex systems, Science of computer programming 8 (3) (1987) 231–274.

[25] N. Mohamed, J. Al-Jaroodi, I. Jawhar, Middleware for robotics: A survey, in: Robotics, Automation and Mechatronics, 2008 IEEE Conference on, Ieee, 2008, pp. 736–742.

[26] A. Elkady, T. Sobh, Robotics middleware: A comprehensive literature survey and attribute-based bibliography, Journal of Robotics 2012.

[27] A. Topalidou-Kyniazopoulou, N. I. Spanoudakis, M. G. Lagoudakis, A case tool for

robot behavior development, in: X. Chen, P. Stone, L. Sucar, T. Zant (Eds.), RoboCup 2012: Robot Soccer World Cup XVI, Vol. 7500 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 225–236.
URL http://dx.doi.org/10.1007/978-3-642-39250-4_21

[28] M. Lötzsch, M. Risler, M. Jüngel, XABSL - A pragmatic approach to behavior engineering, in: Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS), Beijing, China, 2006, pp. 5124–5129.

[29] T. Röfer, CABSL – C-Based Agent Behavior Specification Language, in: H. Akiyama, O. Obst, C. Sammut, F. Tonidandel (Eds.), RoboCup 2017: Robot World Cup XXI, Springer International Publishing, Cham, 2018, pp. 135–142.

[30] T. Niemüller, A. Ferrein, G. Lakemeyer, A Lua-based behavior engine for controlling the humanoid robot Nao, in: J. Baltes, M. Lagoudakis, T. Naruse, S. Ghidary (Eds.), RoboCup 2009: Robot Soccer World Cup XIII, Vol. 5949 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 240–251.
URL http://dx.doi.org/10.1007/978-3-642-11876-0_21

[31] Gostai, Gostai Studio - editor for hierarchical finite state machines, quickstart https://www.youtube.com/watch?v=i3t943v_bZc. Acccessed: 04-07-2019.

[32] M. Klotzbücher, H. Bruyninckx, Coordinating robotic tasks and systems with rfsm statecharts.

[33] S. G. Brunner, F. Steinmetz, R. Belder, A. Dömel, Rafcon: A graphical tool for engineering complex, robotic tasks, in: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2016, pp. 3283–3290.

[34] G. Steinbauer, A survey about faults of robots used in RoboCup, in: RoboCup 2012: robot soccer world cup XVI, Springer, 2013, pp. 344–355.

[35] J. Valdman, Log file analysis, Department of Computer Science and Engineering (FAV UWB)., Tech. Rep. DCSE/TR-2001-04 (2001) 51.

[36] ROS, The rqt_console, http://wiki.ros.org/rqt_console, Accessed: 04-07-2019.

[37] Southwest Research Institute Robotics, The swri_console, https://github.com/swri-robotics/swri_console, Accessed: 04-07-2019.

[38] M. Campusano, A. Bergel, Vizrob case studies, DOI: 10.5281/zenodo.1182762 (Feb. 2018). doi:10.5281/zenodo.1182762.
URL https://doi.org/10.5281/zenodo.1182762

[39] T. H. J. Collett, B. A. Macdonald, An augmented reality debugging system for mobile robot software engineers, JOURNAL OF SOFTWARE ENGINEERING IN ROBOTICS 1 (1) (2010) 18–32.

[40] P. Renner, F. Lier, F. Friese, T. Pfeiffer, S. Wachsmuth, Wysiwicd: What you see is

what i can do, in: Companion of the 2018 ACM/IEEE International Conference on Human-Robot Interaction, ACM, 2018, pp. 382–382.

[41] M. Lanza, S. Ducasse, Polymetric views—a lightweight visual approach to reverse engineering, Transactions on Software Engineering (TSE) 29 (9) (2003) 782–795. doi:10.1109/TSE.2003.1232284.
URL http://scg.unibe.ch/archive/papers/Lanz03dTSEPolymetric.pdf

[42] S. L. Tanimoto, A perspective on the evolution of live programming, in: Proceedings of the 1st International Workshop on Live Programming, IEEE Press, 2013, pp. 31–34.

[43] Apple, inc. Introducing Swift, https://developer.apple.com/swift/, Accessed: 04-07-2019.

[44] D. Ungar, R. B. Smith, Self: The power of simplicity, Vol. 22, ACM, 1987.

[45] B. Swift, A. Sorensen, H. Gardner, J. Hosking, Visual code annotations for cyberphysical programming, in: Proceedings of the 1st International Workshop on Live Programming, IEEE Press, 2013, pp. 27–30.

[46] Willow Garage, RVIZ - 3D visualization environment, http://wiki.ros.org/rviz, Accessed: 06-01-2020.

[47] vvvv group, vvvv - a multipurpose toolkit, http://www.vvvv.org/, Accessed: 04-07-2019.

[48] S. McDirmid, Living it up with a live programming language, in: Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA '07, ACM, New York, NY, USA, 2007, pp. 623–638.
URL http://doi.acm.org/10.1145/1297027.1297073

[49] S. McDirmid, Usable live programming, in: Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software, ACM, 2013, pp. 53–62.

[50] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, J. Kato, It's alive! continuous feedback in UI programming, in: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, ACM, New York, NY, USA, 2013, pp. 95–104. doi:10.1145/2491956.2462170.
URL http://doi.acm.org/10.1145/2491956.2462170

[51] Kodowa Inc, Light Table – The next generation code editor, http://www.lighttable.com, Accessed: 04-07-2019.

[52] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, G. Wachsmuth, DSL engineering: Designing, implementing and using domain-specific languages, dslbook. org, 2013.

[53] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, M. Ziane, Robotml, a domain-specific language to design, simulate and deploy robotic applications, in: International Conference on Simulation, Modeling, and Programming for Autonomous Robots, Springer, 2012, pp. 149–160.

[54] U. P. Schultz, D. J. Christensen, K. Stoy, A domain-specific language for programming self-reconfigurable robots, in: Workshop on automatic program generation for embedded systems (APGES), 2007, pp. 28–36.

[55] A. Nordmann, S. Wrede, J. Steil, Modeling of movement control architectures based on motion primitives using domain-specific languages, in: Robotics and Automation (ICRA), 2015 IEEE International Conference on, IEEE, 2015, pp. 5032–5039.

[56] T. Röfer, T. Laue, J. Richter-Klug, J. Stiensmeier, M. Sünemann, A. Stolpmann, A. Stöwing, F. Thielke, B-human team description for robocup 2015.

[57] ROS Wiki: ROS Topic documentation, Website: http://wiki.ros.org/rostopic, Accessed: 04-07-2019.

[58] S. Bragagnolo, L. Fabresse, J. Laval, P. Estefó, N. Bouraqadi, PhaROS: a ROS Client for the Pharo Language, http://car.mines-douai.fr/category/pharos/ (2014).
URL `http://car.mines-douai.fr/category/pharos/`

[59] J. Laval, Jetstorm - a communication protocol between Pharo and Lego Mindstorms, Tech. Rep. 140616, Mines-Telecom Institute, Mines Douai (jun 2014).

[60] A. Jedlitschka, D. Pfahl, Reporting guidelines for controlled experiments in software engineering, in: Empirical Software Engineering, 2005. 2005 International Symposium on, IEEE, 2005, pp. 10–pp.

[61] J. Kubelka, R. Robbes, A. Bergel, The road to live programming: insights from the practice, in: Proceedings of the 40th International Conference on Software Engineering, ACM, 2018, pp. 1090–1101.

[62] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.

[63] M. Burnett, J. Atwood, R. W. Djang, J. Reichwein, H. Gottfried, S. Yang, Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm, Journal of functional programming 11 (2) (2001) 155–206.

[64] C. D. Hundhausen, J. L. Brown, An experimental study of the impact of visual semantic feedback on novice programming, Journal of Visual Languages & Computing 18 (6) (2007) 537–559.

[65] W. Burgard, D. Fox, S. Thrun, Probabilistic robotics, The MIT Press.