Live Programming the Lego Mindstorms

Johan Fabry Miguel Campusano
PLEIAD and RyCh Laboratories
Computer Science Department (DCC)
University of Chile
{jfabry,mcampusa}@dcc.uchile.cl

Abstract

Development of software that determines the behavior of robots is typically done in a language that is far from dynamic. Programs are written, compiled, and then deployed on a simulator, or the robot, for testing. This long development cycle causes a cognitive dissociation between writing the code for the robot and observing the robot in action. As a result, writing robot behaviors is much more difficult than it should be. In contrast, live programming proposes an extraordinary tightening of the development cycle, yielding an immediate connection between the program and the resulting behavior. To achieve live programming for robot behaviors, we designed and implemented the LRP language. In this paper we show how LRP interfaces with the Lego Mindstorms EV3, report on experiences programming Lego robots, and discuss how salient features of the language were made possible thanks to its implementation in Pharo Smalltalk.

1. Introduction

The origins of live programming can be traced back to the early work of Tanimoto on Viva [12]. It 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." Such live programming allows programmers to benefit from an immediate connection with the program that they are making. This is because the development cycle is extremely tight and there is no cognitive dissociation between writing the code and observing its execution.

In our research we aim to bring the advantages of live programming to programming of robots, more specifically the behavior layer. The behavior layer is the part of the software of the robot that acts on processed inputs to realize specific actions of the robot, *i.e.* its behavior. Typically, such behavior is written in a language that is far from dynamic, compiled, and then deployed on a simulator (or the robot itself) for testing. In this long cycle the cognitive distance between the program and the resulting robot behavior is vast, resulting in a high degree of difficulty of getting these behaviors to work well. For example, it is frequently the case that the programmer observes the robot (or the simulation) performing some specific movement and it is totally unclear *why* this movement is happening. With live robot programming this cognitive distance almost disappears. This is because the development environment includes a visualization of program execution that transparently updates on each program change, in addition to the execution being reflected in the robot simulator or even on the running robot itself.

To allow live programming of robot behaviors we have developed the Live Robot Programming (LRP) language. This language is based on the nested state machine paradigm, as this paradigm has proven to be well-suited to define robot behaviors [8, 14]. LRP is designed from the onset to be a live programming language, and as such comes with its own state machine interpreter and visualization of existing machines. The language is not hardcoded to a specific robot platform, instead relying on bridging software to access specific robot APIs.

In this paper we show how LRP enables live programming of the Lego Mindstorms EV3 robot platform through JetStorm [7], report on our experiences programming the Mindstorms in LRP, and discuss specific points of the implementation of LRP that were facilitated largely by the language features and infrastructure present in Pharo Smalltalk.

This paper is structured as follows: the next section gives a brief overview of the LRP language, using an example

behavior that also serves to illustrate elements of the rest of the paper. Section 3 reports on the bridge to the Mindstorms and our experience in using it to program robot behaviors. Following this, Section 4 highlights specific elements of Pharo Smalltalk that made the implementation possible. The paper then presents related work, future work and concludes.

2. The LRP Language

Live Robot Programming (LRP) is a live programming, nested state machine based language with an associated interpreter and visualization, implemented in Pharo. The features of LRP are designed for robot programming, yet the language is not hardcoded to a specific robot platform. LRP enables the use of APIs of specific robot platforms and as such comes with bridges towards the Robot Operating System (ROS) [5], and now also to the Mindstorms EV3 [13] through JetStorm [7], as will be discussed in Section 3.

A complete description of LRP is outside of the scope of this paper, we only give a brief overview of its features here, and refer to its website `http://pleiad.cl/LRP` and other published work [4] for more details.

The main language features of LRP are:

- *Machines* with states and different kinds of transitions.
- *Transitions* that can occur on events, occur after a timeout or occur automatically after a state is entered.
- *Events* are explicitly defined and trigger if their included piece of code, called an *action*, evaluates to true.
- *States* can have actions that are run when entering the state, leaving the state, or when the state is active.
- States can define state machines, which enables nesting.
- Machines can define variables, and these are accesible inside actions if the variable is lexically in scope.

LRP has its own language syntax and the interpreter is, in essence, a plain state machine interpreter that consumes the ASTs of the program and provides the standard nested state machine semantics. The only remarkable element is that actions are actually Smalltalk blocks that are compiled after the program is parsed. This process is discussed in Section 4.1.

To show the syntax of the language, clarify how it allows for robot programming on the Mindstorms and provide example material for Sections 3 and 4, we now show and discuss the code for a simple behavior. The behavior is a simple space exploring behavior where the robot goes forward until it encounters a wall, where it backs up, turns a bit, and again goes forward. This behavior is ment to run on a differential-drive robot[1] with the (ultrasonic) distance sensor pointing forward and a touch sensor on both front corners. An example of such a robot constructed using the Mindstorms is shown in Figure 1.

[1] Typically a tricycle that has 2 driven wheels, each with its own motor, and the third wheel being a caster

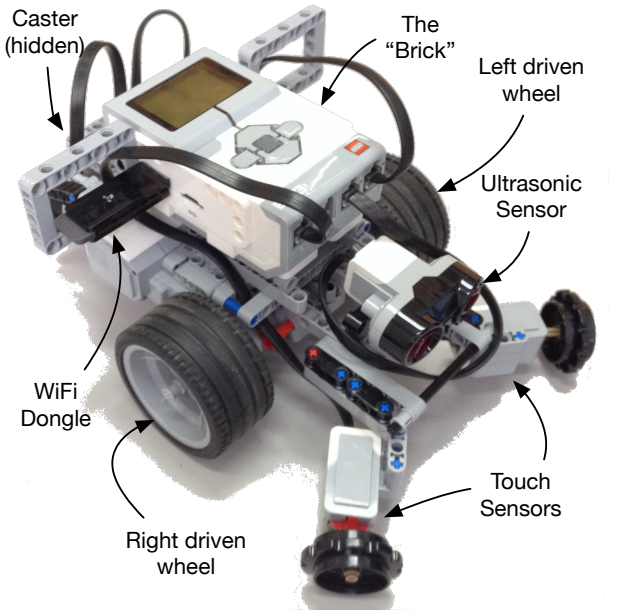

Figure 1. The Lego Mindstorms robot of the explorer behavior example.

The first part of the code, below, takes care of connecting the program to the Mindstorms by reifying the different motors and sensors as variables:

```
1 (var motA := [LRPEV3Bridge motorA])
2 (var motB := [LRPEV3Bridge motorD])
3 (var ultra := [LRPEV3Bridge sensor3])
4 (var righttouch := [LRPEV3Bridge sensor1])
5 (var lefttouch := [LRPEV3Bridge sensor4])
```

Five variables are declared and immediately initialized, which is mandatory. In LRP, code between square brackets are actions, *i.e.* Smalltalk blocks. The class `LRPEV3Bridge` is a facade class responsible for connection to the Mindstorms and making the different sensors and motors available. This is in essence how LRP code interacts with specific robot platforms: reifying relevant elements as variables and subsequently interacting with these variables in actions, *i.e.* in Smalltalk code.

With the variables defined, the definition of the state machine for the behavior starts as below. The machine is called Dora (for Dora the Explorer), and initially defines two states and two transitions:

```
6 (machine Dora
7   (state forward
8     (onentry
9       [ motA value startAtSpeed: 55.
10        motB value startAtSpeed: 55.])
11    (onexit [motA value stop. motB value stop]))
12  (state looking )
13  (ontime 600 forward -> looking t-look)
14  (ontime 120 looking -> forward t-forward)
```

Lines 7 through 11 specify the `forward` state. The block in lines 9 and 10 is executed whenever this state is entered. As it represents the robot moving forward, both motors are started at 55% of top speed. The block in line 11 is executed whenever the robot leaves the `forward` state and therefore stops both motors. The `looking` state in line 12 does not define any actions.

Note that both blocks use the motor variables defined in lines 1 and 2, and always send them the `value` message first. This is because all variables are in fact Smalltalk `ValueHolders`, as we will discuss in Section 4.2.

Lines 13 and 14 show two timeout transitions. The numbers given in the transitions specify a timeout in milliseconds, starting from when the source state is entered, and trigger after the timeout is reached. The text of the remainder of the transition specifies, respectively source state, destination state, and transition name.

With this code in place, the robot alternates between moving forward for 0.6 seconds, and then waiting for 0.12 seconds. In those 0.12 seconds the three different sensors are polled (which takes a bit less than 0.12 seconds), as is defined in the next three lines of code:

```
15 (event wall [ultra value read < 20])
16 (event rightbump [righttouch value read = 1])
17 (event leftbump [lefttouch value read = 1])
18 (on wall looking -> backup t-backup)
19 (on rightbump  looking -> backup t-rt-backup)
20 (on leftbump looking -> backup t-lt-backup)
```

Lines 15 through 17 define events. The interpreter will evaluate the actions for these events only if triggering these events can cause a transition to occur. In this case, the transitions on line 18 through 20 may occur as they start from the `looking` state and go to a `backup` state, defined below.

In summary: if none of the events trigger, the robot goes to the `forward` state, otherwise it goes to the `backup` state.

```
21 (state backup
22   (onentry
23     [ motA value startAtSpeed: -32.
24       motB value startAtSpeed: -32.])
25   (onexit [motA value stop. motB value stop]))
26 (ontime 300 backup -> turn t-turn)
27 (state turn (onentry
28     [ motorA value startAtSpeed: -32.
29       motorB value startAtSpeed: 32.])
30     (onexit [motA value stop. motB value stop]))
31 (ontime 700 turn -> forward t-tforward)
32 )
33 (spawn Dora forward)
```

The above `backup`, and `turn` states, together with the `t-turn` and `t-tforward` transitions implement the behavior of backing up, turning around, and resuming moving forward. The last line of code specifies that the `Dora` machine should be started by the interpreter and that its initial state is `forward`. This `spawn` statement also can be used as an action in an `onentry` of a state, which means that when this state is entered the specified machine should be interpreted.

This code is sufficient for implementing the explorer behavior. When editing this code in LRP, the interpreter is always running and updating the interpreted machine while the programmer types, and moreover the LRP window, shown in Figure 2, displays the tree of current machines, the contents of variables, and a visualization of the machine. The visualization highlights the currently active state (`looking` in the figure) and the last taken transition. Also, variables can be inspected and their values set.

3. Bridging LRP to Robot Hardware: Controlling the Mindstorms

LRP is at its core a live programming language for nested state machines. It is implemented in Pharo, using PetitParser [10] as the parser generator, Roassal2 [2] for the visualization of the state machines, and Spec [11] to build the user interface.

The language features have been designed with the use as a robotics behavior layer in mind, yet the language itself does not have any intrinsic robotics support. This responsibility instead lies on bridging software that spans the gap to specific robot platforms. Currently, LRP comes with a bridge to ROS [5], and the Lego Mindstorms EV3 [13] via JetStorm [7]. In this section we present the latter and discuss a practical issue we faced programming the Mindstorms.

3.1 Hard- and Software

The Lego Mindstorms EV3 [13] is the third iteration of the Lego Mindstorms line. The embedded system of the set is called the *brick*, and it features an 300 Mhz ARM9-based processor, 64MB of RAM which runs Linux 2.6.x. The sensor package (in the education version) is an ultrasound distance sensor, two touch sensors, a color sensor and a gyroscopic sensor. Three motors are supplied, each motor with a built-in rotation sensor. Last but not least, a comprehensive set of Lego bricks are included, enabling the speedy construction of a wide variety of robot hardware.

The brick also includes an USB port, and support for one specific WiFi dongle, which allows the robot to be remote-controlled via WiFi. JetStorm [7] is a Pharo package that allows for the remote control of the EV3 by reifying the brick, sensors and motors as Smalltalk objects that can be sent messages. For example, sending the `startAtSpeed: 55` message to a motor object causes a command to be sent to the brick to start the corresponding motor at 55% of the top speed the motor is capable of.

The LRP Mindstorms bridge currently consists of a facade class `LRPEV3Bridge` that is placed in front of JetStorm. This class provides features for connecting to the brick over IP and retrieving the various sensors and motors connected to the brick. The latter is shown in lines 1 through 5 of the example program. If there is no IP connection to the brick when a sensor or motor is retrieved, the user is prompted for the IP address of the brick and a connection is set up.

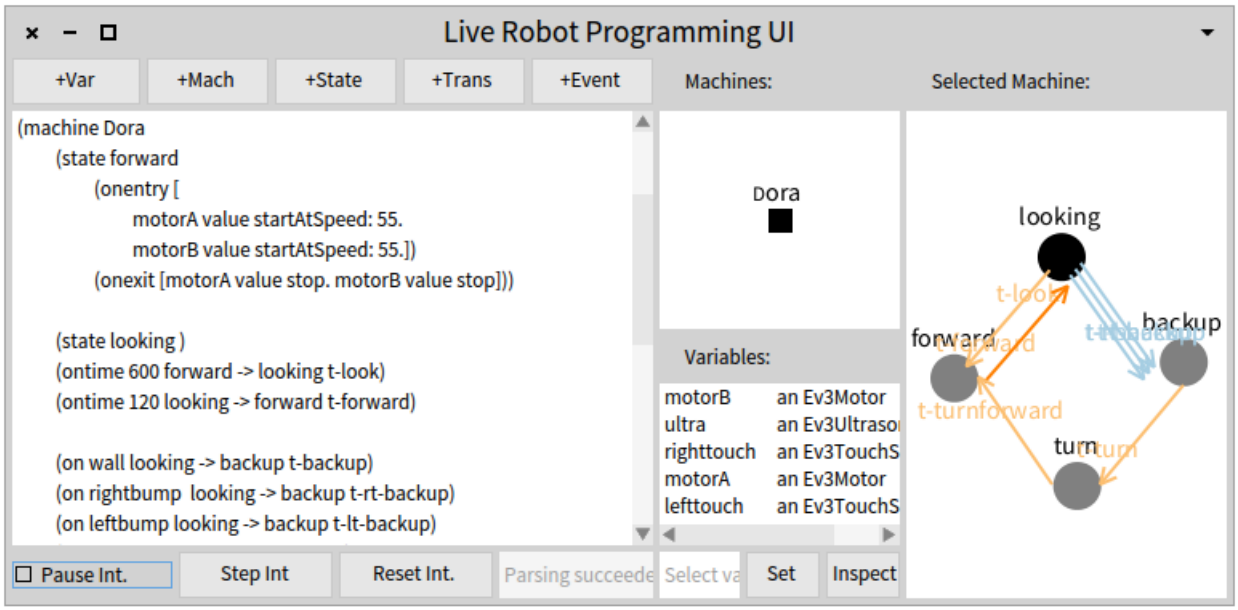

Figure 2. The LRP editor showing part of the example of this text: the Dora machine.

The various sensors and motors that are retrieved are objects provided by JetStorm, no facade is placed in front of them.

In our experience, the one, minimal, facade class has proven to be sufficient to allow small experiments with the Mindstorms. We are however faced with the situation that LRP may grow to have multiple bridges to many different robot API's. For example the API to ROS is quite different. It requires movement vectors to be sent, and their interpretation by the robot eventually causes the respective motors to operate. A wide disparity in how these APIs are exposed to LRP programmers will cause a tight coupling of LRP programs to a specific API and prohibit reuse of behaviors across robot platforms, effectively splintering the language in different versions for different APIs. It would therefore be beneficial to have at least some basic uniformity of the API that the different LRP bridges expose, at least when considering the lowest common denominator of the APIs. Consequently this could possibly require the EV3 Bridge facade to increase in complexity, translating the common API calls to JetStorm calls. We consider the study of such a common API as future work.

3.2 Experience Report: The Issue of Lag

Live Programming of the Lego Mindstorms is a very satisfying experience. It is possible to quickly prototype reasonably complex behaviors, while benefitting from the immediate feedback that live programming brings. We are able to change the behavior of a robot *while it is running* and active in its environment, and the visualization of the state machine allows us to immediately establish in which state the robot is and how it got there. There is only one negative point in the entire experience, and that is the presence of network lag on robot commands.

Sending commands from a computer to the brick over the network and waiting for a reply causes a notable delay in interactions of the LRP interpreter with the robot. Informal microbenchmarks have shown us that it takes approximately 30 microseconds for a sensor read operation to return the sensor's value, and the same time to instruct a motor to start. While this time lag may seem negligible, this turns out not to be the case. For example, in line 15 to 17 of the example code, three sensors are polled, which therefore takes approximately 120 microseconds. This is a noticeable delay, and a time in which the robot may advance a significant distance. For the example the distance traveled in that time is 5 cm, with the motors at 55% of top speed.

It is exactly because of this delay that the Dora behavior is structured in a `moving` and a `looking` phase. The robot first moves for a distance that is deemed 'safe', and then stops to verify if the wall is too close. This results in a stuttering behavior of the robot that is quite noticeable. If reading sensors were immediate, there would be no need for a `looking` state: the robot would continuously poll the sensors for their data. As a result the robot would not stutter and be able to explore at a higher overall speed.

We have experience with programming robots on the predecessor of the EV3, having software run on the brick itself by using the leJOS [6] Java to NXT cross-compiler. While the old brick has significantly inferior hardware, this setup is orders of magnitude more responsive, resulting in robot behaviors that are much more fluid and faster. Consequently, Dora-like behaviors for example can be executed much faster. Note that these experiments were in Java code

and hence did not suffer from any overhead of the LRP interpreter. The overhead of LRP is however almost negligible: in informal tests, the overhead for evaluating events and executing state transitions has been benchmarked to be around one millisecond.

Ideally the robot behavior software would therefore run locally on the EV3 itself. There is however, as yet, no support in Pharo for running on the EV3. As a point of reference, only recently (April 2014) has the first beta release of leJOS on the EV3 been made available. We have not yet been able to experiment with it, nor do we have the resources required to reimplement the LRP interpreter in leJOS.

4. Implementing LRP

The interpreter of LRP is at its core a plain interpreter implementation for nested state machines, extended in two ways for live programming [4]. First it is robust with respect to incomplete programs and keeps on executing in the face of errors. Second it is able to modify the state machine while it is running, adding and removing elements without always requiring a restart.

There are two pieces of the implementation of the interpreter that we discuss here, as they show how the use of Smalltalk has aided us in its implementation, what are limitations due to the implementation and how we plan to address them. These two elements are compilation of the blocks and variables as `ValueHolders`.

4.1 Compiling the Blocks

Actions are used to connect LRP to the API of the specific robot platform. They also may need to perform any kind of computation on sensor inputs and the state of variables to establish whether events occur, and hence may also need to update variables at some point. As a result we found it a natural choice to allow actions to have the full power of Smalltalk available and hence have them be Smalltalk blocks.

Having actions as blocks however raises issues of performance. While the behavioral layer is not a time-critical element in the software of the robot, it does form part of a computation chain that goes from sensor readings up to actuator actions. As such, any overhead that it adds in this process does have effect on the performance of the robot. For this reason, we decided that the overhead of executing actions must be minimal. Hence, in the interpreter actions are compiled blocks: to run them only the `value` message needs to be sent.

Parsing in LRP is performed by PetitParser [10], and action blocks are also parsed, using the Smalltalk parser that is part of PetitParser. As a result, when the interpreter is passed the AST of the state machine to interpret, these blocks have the form of ASTs. The interpreter traverses the complete AST for the program and compiles all action blocks. The result of the compilation of an action block is a `BlockClosure` that has references to all the variables in scope and hence just needs to be sent the `value` message to execute.

The process of compiling the AST of an action block is as follows:

1. a `Dictionary` is created of all variables in scope, taking into account shadowing of variables.
2. Text for the signature for a method is created of the form `captureV:V:V:`, taking as many `V:` arguments as the number of variables in the dictionary.
3. The names of the parameters of this signature are the keys in the dictionary. In the body of the method, the LRP variables are hence in scope of the Smalltalk code.
4. The signature is appended with the string `'^ [1]'` and this complete method definition string is parsed.
5. In the resulting AST method, the subtree for `'[1]'` is swapped with the AST of the block to compile.
6. This method AST is compiled.
7. The resulting method is invoked, passing it the values of the variables in the correct order. This causes the `BlockClosure` to capture variable references such that they may be used inside the code of the action.

For example, let us consider the onentry block of lines 9 and 10 of the Dora example. The result of step 5 is the AST for the following:

```
1 captureV: motB V: ultra V: righttouch
2         V: motA V: lefttouch
3     ^ [motA value startAtSpeed: 55.
4        motB value startAtSpeed: 55 ]
```

Step 6 yields a `CompiledMethod` for the above, *i.e.* a method whose execution returns the `BlockClosure` that corresponds to the action (lines 3 and 4). In step 7, this method is invoked with as arguments the values of the variables `motB`, `ultra`, `righttouch`, `motA`, `lefttouch`. The returned `BlockClosure` has hence captured the references for the variables it uses (`motA` and `motB`). This allows the action to be executed by simply sending the `value` message to this `BlockClosure`.

The compilation of action blocks has turned out to be quite straightforward to implement, taking only about 20 lines of code (of arguably low complexity). We consider that being able to achieve such a complex task so succinctly is a testament to the power and flexibility of Pharo Smalltalk.

There are however two downsides to the current implementation. Firstly, the method that is compiled has no class and an incorrect source code pointer. In our experience this has caused issues when programming: the block cannot be printed, the debugger does not work correctly and in some cases even primitive error handling fails, causing Pharo to crash. An important avenue of future work is to improve the compilation process such that these issues are addressed.

Secondly, methods can only take up to 16 arguments. Consequently, if there are more than 16 arguments in scope, compilation of the action block fails. A possible mitigation of this issue would be to perform a semantic analysis of the block to establish which variables are effectively used inside the block and only pass these as arguments in step 2,3, and 7 above. We also consider this as future work.

In summary We were able to incorporate the full power of an OO language in our state machine-based language thanks to the fact that we have straightforward access to the following:

- a parser of Smalltalk expressions that produces ASTs,
- ASTs of methods allowing for their compilation at runtime, isolated from a class definition,
- blocks that capture the arguments of their enclosing method when they are created.

4.2 LRP Variables are ValueHolders

In LRP, variables are key to interact with specific robot platforms. This is because they are used to reify API elements from these platforms and the code in actions interacts with these elements. For example, in the Dora example above, actions start and stop motors and poll sensors. Variables however serve as more than that, and this can be already seen in the Dora example. The example contains many magic numbers, *e.g.* motor speeds, minimal wall distance (in line 15), and timeouts for the different transitions. All these numbers can (and actually should) be replaced by the use of variables, turning these magic numbers into robot calibration constants. Beyond cleaning up the code, this has as consequence that they can then be modified in the LRP editor while the program runs, effectively recalibrating the robot as it runs. Lastly, if the turning time on line 31 would be a variable, it could contain a random number that is set every time a turn is about to begin. This randomizes the turns, making the exploring behavior immune to being stuck in a loop. Because all of the above reasons, variables must truly be mutable.

Yet these variables are used by three different entities: the original program AST that contains the result of variable initialization, the LRP editor, and the different actions that use these variables. Recall that these blocks get passed these variables by reference when they are constructed, as discussed in Section 4.1. As a consequence, any change to the values of variables is invisible to these blocks! This is because changes to the values do not affect the references that were passed to the blocks as they were constructed. Hence variables may not be changed.

To address the issue that values of variables may not be changed yet at the same time they must be mutable, we have made use of ValueHolders. Every variable is a `ValueHolder` that contains the value. This however entails that reading the value of a variable requires sending the `value` message to the variable, and setting the value of a variable is using the `value:` message instead of normal assignment.

Our experience has shown in practice that in the beginning of writing LRP code it is easily forgotten that variables are `ValueHolders`, leading to widespread errors in behaviors. Such errors are however quickly revealed: simple variable accesses usually already cause problems as the `ValueHolder` class implements few messages. We are planning transparent use of `ValueHolders`, *i.e.* not requiring the use of the `value` and `value:` messages, as future work. We have first considered source code manipulation of the code in the block to automatically transform accesses and modifications to the use of this messages. This however does not address the issue of the variables being used and modified outside of the block, *e.g.* when they are passed as method parameters. A second possible path would be to try the new Slots mechanism. We would have variables as slot instance variables of a purpose-built class. The slot reading and writing mechanism would then implement the extra indirection that is currently achieved by the `ValueHolders`. As the Slots mechanism has not been fully implemented its suitability is however yet to be determined.

In summary We required the use of a double indirection to be able to have mutable variables, and the `ValueHolder` mechanism has shown to be a fitting solution. Requiring the use of `value` and `value:` messages in actions is however suboptimal, and we are planning solutions to this issue.

5. Related Work

Considering robot behaviors using nested state machines, two languages and tools are well-known: The Kouretes Statechart Editor (KSE) [14] and XABSL [8]. In KSE state machines are graphically edited, with an option to start from a text-based description. The tool then follows a model-driven process to generates the executable code for these machines. XABSL is text-based, using an XML representation of the state machines. A variety of support tools are present, for example, a tool that creates (static) diagrams of the machine.

None of the languages above provide any support for live programming, and the live programming languages below do not consider state machines as their computational model.

Live programming was first proposed by Tanimoto [12]. The language presented in that work is VIVA, a visual programming language for image manipulation. More recently, McDirmid proposed the SuperGlue language [9], based on dataflow programming and extended with object-oriented constructs. Live programming of the UI has been proposed by Burckhard *et al.* [3], by adding specific features for live UI construction to an existing programming language. The keynote of Victor [15] shows multiple live programming examples in Javascript, producing pictures, animations and games. A recent addition to live programming is the Swift language by Apple [1], which allows for live programming in specific workspaces called Playgrounds.

6. Conclusion and Future Work

In this paper we have reported on our first experiences of writing Live Robot Programming (LRP) programs for the Mindstorms EV3, and detailed how some of the features of Pharo Smalltalk allowed us to accomplish its implementation.

We first gave a brief overview of LRP through the use of an example program. The program implements a space exploration behavior on a differential drive robot constructed using the Mindstorms (illustrated in Figure 1). We then discussed the LRP bridge to the Mindstorms. LRP allows for the live programming of robot behaviors, yet is not linked to a specific robot platform, instead relying on such bridging software. This was followed by an experience report that focused on how the lag in sending commands to the EV3 negatively impacts robot performance. We then discussed how specific features of Smalltalk have aided in the construction of the LRP interpreter, more specifically the parsing and AST manipulation and compilation support, blocks and ValueHolders.

There are multiple avenues of future work, which we have discussed in some detail along this text. In summary, these avenues consist of the study of a minimal common API for the bridges to different robot platforms, improvements of the compilation process of action blocks, and elimination of the ValueHolder messages for variables.

In our experience, live programming for robot behaviors yields an order of magnitude faster development time, and is a key enabler of fast prototyping of and experimentation with behaviors. Lastly, without the language features of Smalltalk and all the infrastructure available in Pharo its implementation would have been much more demanding, if not impossible, to realize with the resources at our disposal.

More Information, Availability

The home page of LRP is `http://pleiad.cl/LRP` The implementation of the language is open source, MIT license, and download instructions are on its home page.

References

[1] Apple, inc. Introducing swift. https://developer.apple.com/swift/.

[2] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Square Bracket Associates, 2013.

[3] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun 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, pages 95–104, New York, NY, USA, 2013. ACM.

[4] Johan Fabry and Miguel Campusano. Live robot programming. In Ana Bazzan and Karim Pichara, editors, *Advances in Artificial Intelligence - IBERAMIA 2014*, number 8864 in Lecture Notes in Computer Science. Springer Verlag, 2014.

[5] Open Source Robotics Foundation. ROS.org: Powering the world's robots. http://www.ros.org.

[6] The leJOS Group. leJOS: Java for LEGO mindstorms. http://www.lejos.org/.

[7] Jannik Laval. Jetstorm - a communication protocol between Pharo and Lego Mindstorms. Technical Report 140616, Mines-Telecom Institute, Mines Douai, jun 2014.

[8] Martin Lötzsch, Max Risler, and Matthias Jüngel. XABSL - A pragmatic approach to behavior engineering. In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, pages 5124–5129, Beijing, China, 2006.

[9] Sean 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, pages 623–638, New York, NY, USA, 2007. ACM.

[10] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010.

[11] Benjamin Van Ryseghem, Stphane Ducasse, and Johan Fabry. Seamless composition and reuse of customizable user interfaces with spec. *Science of Computer Programming*, (0), 2014. In Press.

[12] Steven Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, June 1990. http://dx.doi.org/10.1016/S1045-926X(05)80012-6.

[13] The LEGO group. LEGO MINDSTORMS Education EV3. https://education.lego.com/mindstorms.

[14] Angeliki Topalidou-Kyniazopoulou, Nikolaos I. Spanoudakis, and Michail G. Lagoudakis. A case tool for robot behavior development. In Xiaoping Chen, Peter Stone, LuisEnrique Sucar, and Tijn Zant, editors, *RoboCup 2012: Robot Soccer World Cup XVI*, volume 7500 of *Lecture Notes in Computer Science*, pages 225–236. Springer Berlin Heidelberg, 2013.

[15] Bret Victor. Inventing on principle. Invited Talk at CUSEC'12, 2012. Video recording available at http://vimeo.com/36579366.