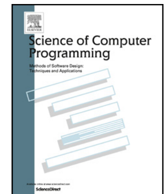
Science of Computer Programming 133 (2017) 1–19

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico



Live Robot Programming: The language, its implementation, and robot API independence ☆

Miguel Campusano *,[1], Johan Fabry

PLEIAD and RyCh labs, Computer Science Department (DCC), University of Chile, Beauchef 851, Santiago, Chile

A R T I C L E   I N F O

Article history:
Received 8 September 2015
Received in revised form 28 March 2016
Accepted 9 June 2016
Available online 17 June 2016

Keywords:
Live Programming
Nested state machines
Robot

A B S T R A C T

Typically, development of robot behavior entails writing the code, deploying it on a simulator or robot and running it in a test setting. If this feedback reveals errors, the programmer mentally needs to map the error in behavior back to the source code that caused it before being able to fix it. This process suffers from a long cognitive distance between the code and the resulting behavior, which slows down development and can make experimentation with different behaviors prohibitively expensive. In contrast, Live Programming tightens the feedback loop, minimizing the cognitive distance. As a result, programmers benefit from an immediate connection with the program that they are making thanks to an immediate, 'live' feedback on program behavior. This allows for extremely rapid creation, or variation, of robot behavior and for dramatically increased debugging speed. To enable such Live Robot Programming, in this article we discuss LRP; our language that provides for live programming of nested state machines. We detail the design of the language and show its features, give an overview of the interpreter and how it enables the liveness properties of the language, and illustrate its independence from specific robot APIs.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Live Programming has recently come under attention thanks to the widely commented talk by Bret Victor at CUSEC'12 [2]. Its origins can however be traced back to the early work of Tanimoto on VIVA [3]. 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 programmers benefit from an immediate connection with the program that they are making. Languages for live programming therefore provide for an immediate, 'live' feedback on program behavior. Such tightening of the feedback loop lightens the cognitive load of building 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 the programmers' mental model of the executing program differing from the actual behavior.

☆ This article is an extended version of the article "Live Robot Programming" by Fabry and Campusano presented at the IBERAMIA 2014 conference [1].

* Corresponding author.
E-mail addresses: mcampusa@dcc.uchile.cl (M. Campusano), jfabry@dcc.uchile.cl (J. Fabry).

[1] Miguel Campusano is funded by CONICYT-PCHA/Doctorado Nacional/2015-21151534.

http://dx.doi.org/10.1016/j.scico.2016.06.002
0167-6423/© 2016 Elsevier B.V. All rights reserved.

Putting this in a robotics context, programming of robot behaviors is however currently far from 'live'. Typically, the robot behavior is written, compiled and then deployed on a simulator (or the robot itself) for testing. Each step in this cycle increases the cognitive distance between the program and the robot's behavior. 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. With Live Robot Programming this gap would all but disappear: the programming language environment includes some form of program simulation that immediately and transparently updates on each program change.

Considering the languages used for programming robot behavior, arguably the most successful are those based on hierarchical state machines, *e.g.* XABSL [4] or the Kouretes Statechart Editor [5]. Such machines are said to naturally map to the problem domain, and multiple RoboCup teams have won the competition using these languages. Live programming in such languages would enable – while the program is running in a simulator, or on the robot itself – to add behavior by adding extra states or machines, or to debug behavior by changing the program on the fly. An example of the latter is a bug in the activation condition of a transition: It should trigger with the current inputs but does not. While the program is running, the condition is edited such that it does trigger, which is immediately observed, confirming that the bug is fixed.

In this text, we introduce a language and associated interpreter for live programming of robot behaviors, called LRP. LRP is a language for nested state machines that provides for a custom visualization of the program while it runs, and notably has the ability to change the program *while it is running*. We discuss the design of the language and show its features. An example is then given that illustrates the usefulness of live programming in a robotics context.

LRP combines interpreted with compiled code by embedding a complete OO 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 show here two instances of such bridges: one to the Robot Operating System (ROS) [6] and one to the Lego Mindstorms Ev3 [7].

This paper is structured as follows: The next section presents live programming and then Section 3 introduces the LRP language, using a simple line follower program as example. Section 4 then illustrates how live programming in LRP aids in programming the behavior of looking for the line. The interpreter of LRP is outlined in Section 5 and this is followed in Section 6 by a discussion on how the interpreter ensures liveness. The implementation of the software bridges to different APIs is presented in Section 7. The paper then presents related work and future work before concluding.

2. Live Programming

Live programming was first introduced by Tanimoto in VIVA [3]. 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. Then, the execution environment will 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 [3]. Hence, in this work, when we talk about live programming we 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 more work has been published about liveness in programming languages, which is not longer restricted to visual languages, such as Flogo II [8] or Swift [9]. Moreover, the live programming concept has been expanded over the years to include many more programming languages, also considering level 3 of Tanimotos definition to be live programming, *e.g.* Self [10].

Swift et al. in their work on Visual Code Annotations [11] 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 perform 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 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.

While it is true in a level 4 of liveness of Tanimoto the amount and rate of feedback is maximized, for Hancock it is not enough for a live programming environment to have only continuous feedback [8]. Hancock claims there are two more things in consideration: 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 [8].

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. He says it is different to aim and shoot an arrow than to aim and "shoot" water with a hose. It is much easier to aim a hose rather than a bow and an 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 the 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 our target at all time, we have continuous action [8].

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 [8], 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 [7] 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 to take continuous actions over a continuous feedback loop on their programs. In this work, we also consider live programming to not only be about feedback, but also about using this continuous actions to build programs.

Summarizing, there are different interpretations and levels of liveness in programming languages, 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. The concept of live programming that we use in this work is the level 4 of Tanimoto, where modifying the behaviors of programs is directly related to a continuous action over a program, as Hancock states.

3. The LRP language

LRP (Live Robot Programming) is a language for nested state machines, which is arguably a natural paradigm for designing and programming robot behaviors. This is because the design of the language is inspired by existing languages for robot behavior programming based on the same paradigm, *e.g.* XABSL [4], Kouretes State Charts [5], and the Lua behavior engine [12]. 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 CASBL [13] and several teams adapted the German team code for their own robots [14]. Locally, the team from our university uses XABSL to program the

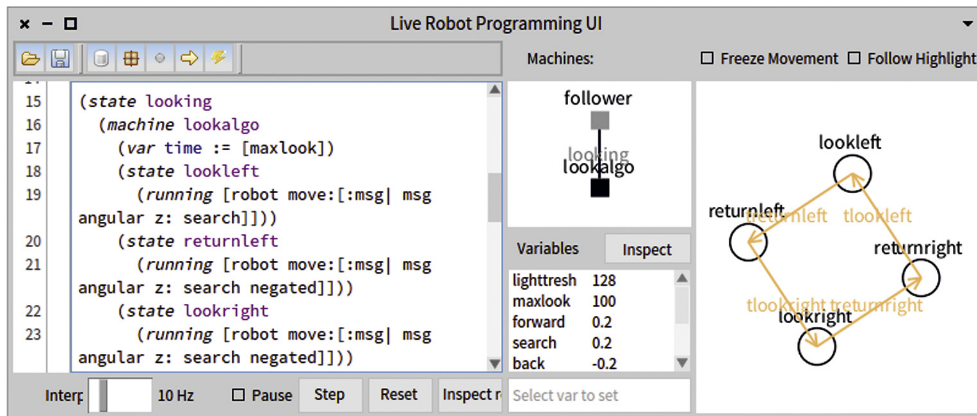


Fig. 1. The LRP editor showing part of the running example of this text. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

behavior for their robots with good results, *e.g.* achieving 4th place in the Standard Platform League of the 2015 RoboCup.[2] Moreover, languages such as XABSL have been an alternative for programming behaviors outside robotics, such as games [4]. Note that LRP, like the languages it is inspired on, is not intended for computationally intensive applications such as image recognition and the like. 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 Section 2). The editor, shown in Fig. 1, 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.

The UI editor is an important part of LRP, since the UI supports an important part of live programming: steady frames (presented in Section 2). With the visualization of variables 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. 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 make it possible for developers to build and test programs from the very beginning naturally. More details on steady frames is given in Section 2.

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 Section 2).

3.1. 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). For the moment, LRP only supports one machine per execution, *i.e.* LRP does not support concurrent state machines. The language is tightly coupled with its interpreter, due to its live programming nature and the need to interface to 'the outside world', *i.e.* the parts of the robot software on which the behavioral layer relies.

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.[3] When

[2] http://result.robocup2015.org/show/item?id=65.

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

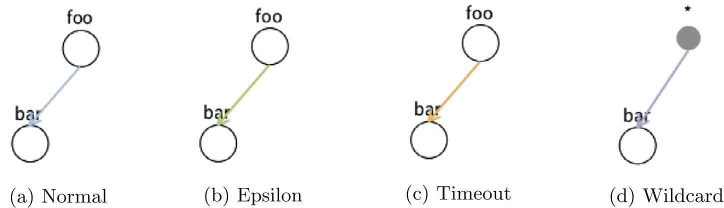


Fig. 2. Types of Transitions in LRP. Note artificial wildcard source state in (d). (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

the state becomes the active state, its onentry action is executed once. This execution is atomic, *i.e.* it cannot be interrupted by a state change. When the state stops being the active state, its onexit action is executed once, also atomically. While the state is the active state, its running action is atomically executed, once per interpretation loop. If a state with a nested machine stops being active, the nested machine is stopped and discarded. As part of this process, the onexit action in the active state of this nested machine is performed after any machines nested in that state are stopped and discarded.

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.

Machines may also define *events*: named actions that are the conditions 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. 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*. 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, 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 Fig. 2.

To determine where to start interpretation, LRP has a bootstrapping construct that specifies the machine to *spawn* and which is the start state of that machine. One spawn construct can be placed at top-level, and the onentry actions of states may spawn all machines that are lexically in scope.

Last but not least, LRP has a means 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. As we will discuss in Section 7, LRP currently provides bridges to two different robotics systems: ROS and the Lego Mindstorms EV3.

3.2. 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 was chosen as it 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. Its task is to first follow a black line painted on the ground until it bumps 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: `lighttlim`, `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.

As said above, the language provides for a bridge to realize the connection to the robot, which is discussed more in detail in Section 7. 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 7.1). 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 7.1). 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 here.

```
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 has as name `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 Fig. 1 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 a logical `and` operation 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 1000 milliseconds (1 second). After that, the program enters the `bumpturn` state, where the robot turns. The robot will turn for 3000 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 turn 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 beginning) 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.

4. 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 [2] sparked wide interest, and pioneering work [3] 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 3.2 (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 2). Some video examples of LRP code being edited are available on the LRP website.[4]

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

[4] http://pleiad.cl/lrp.

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 6.1). 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 6.2). 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.

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

5. 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.

5.1. 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 [15]. 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 active state is made active, updating the UI. The timers for the timeout transitions are restarted and its onentry action is executed. If there is a spawn statement for a nested state machine, the nested state machine is 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 actions of the events of these transitions 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 action of the active state is run. If the active state has an active nested machine, the nested machine is 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 a nested machine, the active state of that machine is left, *i.e.* step 6 is executed for that machine,
   - the onexit action of the state is run,
   - the nested machine, if any, is 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 5.3.

5.1.1. Optimizing checking of events

A program in LRP implements the behavior of a robot, so executing the program slowly is translated 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, because evaluating an event in LRP may mean triggering 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.

5.1.2. 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 5.1. 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.

5.2. Variables, scope and blocks of code

As stated in Section 3.2, 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 that 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 robot, and so its execution should be as efficient as possible (and a small benchmark is discussed in Section 5.3). Hence, 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's run by the interpreter when the action block is run.

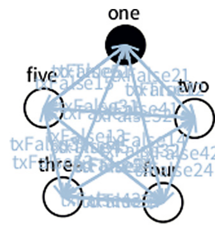


Fig. 3. Program with 5 states fully connected to do a benchmark.

Table 1
Benchmark of a 5 state fully connected program.

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

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 firstly causes the block to be compiled. Secondly, 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.

5.3. A note on performance

The speed at which LRP runs is of importance 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 5.1. 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. A visualization of the program can be seen in Fig. 3.

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 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 can 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 1.

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. [12] report the same time, however on unspecified hardware.

6. 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, as explained in Section 2. 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 here how this is achieved in LRP, together with interpreter features such as pausing and forcing state changes.

6.1. Dealing with program changes

To deal with program changes, the LRP interpreter briefly pauses when they occur, to analyze each change and determine 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, so no onexit action to execute, nor another state to make active. Hence 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.[5] 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 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.

6.2. 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 4 we have seen the addition of a transition that specifies the name of a destination state that does not exist

[5] 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.

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.

6.3. 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 5.1).

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 3.2, 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 to better understand the changes of states and variables. In other words, they can see better 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. 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 3.2, where the program is in the state `looking`, *i.e.* looking for the 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 4: 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 help in avoiding 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.

6.5. 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 2) at different levels, as explained in Section 3. In this section 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 Section 2, 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 is the active state of that machine, as it is highlighted. Thanks to this visualization, we can support steady frames at the "Program Development and Testing" level. As explained in Section 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), 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 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 5.1). Second, it provides buttons to reset interpretation, *i.e.* to restart the program, and to pause the interpretation (see Section 6.3). 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 the bridge is given in Section 7).

Note that, as explained in Section 3, 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.

7. 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 [6] and the Ev3 Lego Mindstorms [7].

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 [16] and JetStorm [17], 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.

7.1. Bridging LRP to PhaROS

ROS 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.

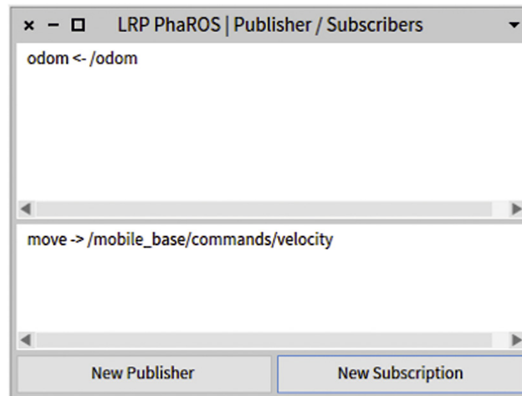


Fig. 4. GUI showing publishers and subscriptions and the buttons to create them.

PhaROS, 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 PhaROS as is 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 put 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 said before, to receive data developers need to make a subscription to a topic. To send data, developers have to build a publisher to a topic. Even when developers know the topic they want to send or receive data, the actual code to do that is 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 a subclass. `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 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 Fig. 4. In the window a developer can choose between generating a publisher or a subscription. When choosing any of the

Fig. 5. Window asking for information to make a subscription.

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 Fig. 5.

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 code in Section 3.2). 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.

7.2. Bridging LRP to JetStorm

JetStorm is a client to the Ev3 Lego Mindstorms [7] 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. JetStorm allows to send commands to the robot to move motors or to read data from the sensors [17]. 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 to be 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.

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. 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.

7.3. 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 `openInterface` 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.

With this implementation we expect that any other API with robot functionalities and GUIs should be implementable as a bridge inside LRP.

8. Related work

Live programming was first proposed by Tanimoto [3], with the goal of providing maximum feedback to the programmer while a program is being constructed. The paper 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 [18], where code is edited by connecting dataflow patches.

Outside of visual programming, the SuperGlue language [19] 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 [20]. YinYang focuses on debugging, combining debugging with editing, then making changes while debugging that also change program execution dynamically.

The work of Victor [2] 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 [21]: an editor that adds live programming features to a number of general-purpose languages.

Apple released Swift [9], 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*.

However, none of these languages consider state machines as their computational model, which is why we consider them radically different from LRP. To our knowledge, the only other live programming language that uses state machines as its computational model is InterState [22]. 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. Participants were able to implement some tasks in InterState twice as fast as using JavaScript.

Considering the use of state machines in a robotics context, the Kouretes Statechart Editor [5] 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, prohibiting any form of live programming, nor is there integration with a simulator or robot.

XABSL [4] 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, but however does not include any simulation support or simulator integration. As a result, it has the same drawbacks as the Kouretes Statechart Editor when compared to LRP.

Another example of hierarchical state machine programs in robotics is Gostai Studio [23]. 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. As in LRP, states in Gostai Studio represent behaviors. Transitions are associated directly with their guard code, contrary to LRP where this code is modularized into events that transitions use to trigger change of behaviors. 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 live programming and LRP. Moreover, LRP allows three different kinds of actions on states – onentry, running, onexit – and Gostai Studio only allows for the equivalent of a running action.

We are aware of two other live programming languages used in robotics. Firstly, Lim [24] extended a live programming music synthesis software with robot programming features. This language is a standard imperative programming language, and not based on nested state machines as in LRP. 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 [8]. 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 Flogo I is similar to LRP in that a dataflow representation is used to program behaviors, like nested state machines, Flogo II differs from LRP because it does not follow the same kind of representation nor goals. Flogo I and LRP have differences as well: Flogo I only accepts numerical values inside its wires whereas LRP supports any data that is supported in Pharo Smalltalk. Moreover, both execution models, dataflow and nested state machines are different. Lastly, the focus of Hancock's research is to improve teaching of programming to people, especially children. In LRP, the focus is on a complete solution for live programming of the behavioral layer of robots using nested state machines. Making software for robots is our goal, not just a motivation. However, both ways of thinking; ours and Hancock's, can be combined at some points to produce better software for robotics, focusing on helping developers as well as teaching and introducing programming to non-developers.

9. Future work

Our main avenue of related work is carrying out user studies to understand the benefits from using LRP, and which LRP features contribute to these benefits, if any. First we will compare LRP with a common platform for building robotic behaviors, more specifically ROS [6] and its behavior programming libraries. Then, we plan to investigate more in depth specific features of LRP to better understand how each of them affects the development of a behavior. In particular, we plan to investigate level 4 liveness, dynamic visualization, the visualization itself and the grammar of the language. To understand how level 4 liveness (see Section 2) affects the development of a behavior in LRP we will compare the original version of LRP with a variant that only has level 3 liveness. For the dynamic visualization we will compare the level 3 liveness LRP version with the same version but with a static visualization. This visualization does not change automatically when the program is written, but the developer should explicitly ask the UI for a visualization (*e.g.* by pressing a button). For the visualization itself, we plan to compare LRP with the static visualization and a version of LRP without any visualization at all. Finally, for the grammar, we compare LRP without any liveness feature nor visualization with the same platform we chose in the first experiment, ROS with its behavior libraries.

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 [2] 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.

We also see improvements to the language itself. In some robotic systems, such as JetStorm (see Section 7.2) 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 to add 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.

Previously, in Section 3.1, we explained how wildcard transitions match on every state of their machine. We could improve this to have other kinds of wildcards to match subsets of states. Continuing with the language design, it would be beneficial to implement refactorings that allows renaming, *e.g.* of states, allowing some interpreter restarts to be avoided. We also explained in the same section that LRP does not support concurrent state machines, we need to investigate however how this would match with the liveness capabilities of LRP before trying to add it into the language.

In the future we want to add visual programming features to LRP. States and transitions could then also be added via the UI. The actions of states, events or even variables could be added using visual programming, which may make LRP more intuitive to use than using only code to program. Also regarding interaction with users, presently errors are detected dynamically, *i.e.* when the program is running. We plan to statically detect more kinds of errors, *e.g.* a transition associated with a non-existing event. Moreover, we will highlight static errors directly in the program code, as in current IDEs.

Longer terms goals of LRP include testing the limits of its expressibility by building a wide variety of complex robotic behaviors using the bridges explained in Section 7.

10. Conclusions

In this paper we presented LRP: a live programming language for the construction of robot behavior using nested state machines. First we introduced live programming. Then 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 however 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 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, and we also presented 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.

Acknowledgements

We would like to thank the following colleagues for fruitful discussions on a precursor of the LRP language that helped shape LRP itself: Wolfgang De Meuter, Pablo Guerrero, Andoni Lombide, Serge Stinkwich and Éric Tanter.

References

[1] J. Fabry, M. Campusano, Live robot programming, in: A. Bazzan, K. Pichara (Eds.), Advances in Artificial Intelligence, IBERAMIA 2014, in: LNCS, vol. 8864, Springer-Verlag, 2014, pp. 445–456.
[2] B. Victor, Inventing on principle, Invited talk at CUSEC'12, video recording available at http://vimeo.com/36579366, accessed: 06-09-2015.
[3] S. Tanimoto, VIVA: a visual language for image processing, J. Vis. Lang. Comput. 1 (2) (1990) 127–139, http://dx.doi.org/10.1016/S1045-926X(05)80012-6.
[4] 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.
[5] 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.), Robot Soccer World Cup XVI, RoboCup 2012, in: Lecture Notes in Computer Science, vol. 7500, Springer, Berlin, Heidelberg, 2013, pp. 225–236.
[6] 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.
[7] The LEGO group, LEGO MINDSTORMS education EV3, https://education.lego.com/mindstorms, accessed: 06-09-2015.
[8] C.M. Hancock, Real-time programming and the big ideas of computational literacy, Ph.D. thesis, Massachusetts Institute of Technology, 2003.
[9] Apple, Inc., Introducing Swift, https://developer.apple.com/swift/, accessed: 06-09-2015.
[10] D. Ungar, R.B. Smith, Self: The Power of Simplicity, vol. 22, ACM, 1987.
[11] 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.
[12] 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.), Robot Soccer World Cup XIII, RoboCup 2009, in: Lecture Notes in Computer Science, vol. 5949, Springer, Berlin, Heidelberg, 2010, pp. 240–251.

[13] T. Röfer, T. Laue, J. Müller, M. Bartsch, M. Batram, A. Böckmann, M. Böschen, M. Kroker, F. Maaß, T. Münder, et al., B-human team report and code release 2013, Only available online: http://www.b-human.de/downloads/publications/2013/CodeRelease2013.pdf, 2013.
[14] T. Röfer, T. Laue, J. Richter-Klug, J. Stiensmeier, M. Sünemann, A. Stolpmann, A. Stöwing, F. Thielke, B-human team description for Robocup 2015.
[15] ROS Wiki: ROS topic documentation, Website: http://wiki.ros.org/rostopic, accessed: 06-09-2015.
[16] 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.
[17] J. Laval, Jetstorm – a communication protocol between Pharo and Lego Mindstorms, Tech. rep. 140616, Mines-Telecom Institute, Mines Douai, jun 2014.
[18] vvvv group, vvvv – a multipurpose toolkit, http://www.vvvv.org/, accessed: 06-09-2015.
[19] 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, http://doi.acm.org/10.1145/1297027.1297073.
[20] 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.
[21] Kodowa Inc., Light table – the next generation code editor, http://www.lighttable.com, accessed: 06-09-2015.
[22] 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.
[23] Gostai, Gostai Studio – editor for hierarchical finite state machines, Web page http://www.gostai.com/products/studio/gostai_studio/index.html, accessed: 06-09-2015.
[24] J. Lim, Live programming for robotic fabrication, J. Prof. Commun. 3 (2) (2014) 165–175.