Live Robot Programming

Johan Fabry and Miguel Campusano

PLEIAD and RyCh labs,
Computer Science Department (DCC), University of Chile, Chile
{ jfabry | mcampusa } @dcc.uchile.cl

Abstract. Typically, development of robot behavior entails writing the code, deploying it on a simulator or robot and running it for testing. 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 large 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 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 propose a language that provides for live programming of nested state machines and integrates in the Robot Operating System (ROS). We detail the language, named LRP, illustrate how it can be used to rapidly implement a behavior on a running robot and discuss the key points of the language that enables its liveness.

1 Introduction

Live programming has recently come under the attention thanks to the widely commented talk by Bret Victor at CUSEC'12 [9]. Its origins can however be traced back to the early work of Tanimoto on Viva [7]. 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.

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. 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.

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 [8]. 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 visualisation of the program while it runs and notably has the ability to change the program *while it is running*. LRP has an integration with the Robot Operating System (ROS), yet can also be integrated in any robot software for which an API is available.

This paper presents the following contributions:

- It introduces the concept of live programming for robot behaviors.
- It presents a live programming language for nested state machines, with an associated interpreter and integration in ROS.
- It defines which code changes allow the interpreter to continue seamlessly.
- It states which program errors needs to be ignored to ensure that the interpreter keeps operating in the face of errors.

This paper is structured as follows: The next section introduces the LRP language, using a simple line follower program as example. Section 3 then illustrates how Live Programming in LRP aids in programming the behavior of looking for the line. This is followed in Section 4 by a discussion on how to ensure liveness. The paper then presents related work before concluding.

2 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. 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 [8], and the Lua behavior engine [6]. As such, the language is not intended for computationally intensive applications such as image recognition and the like.

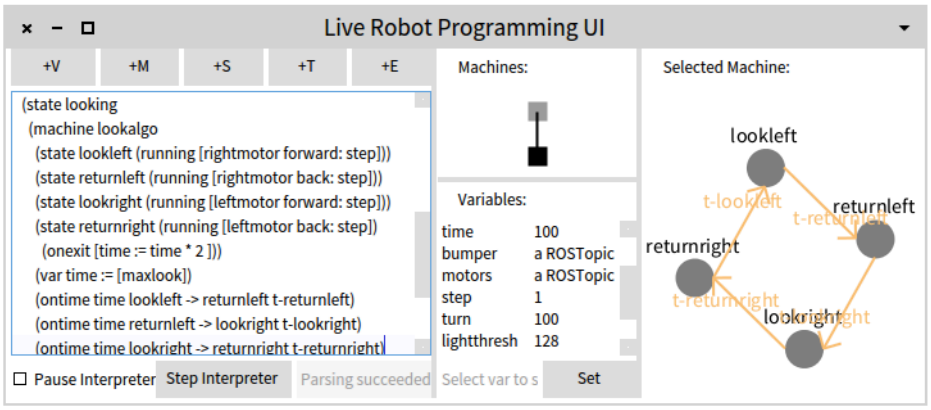

Fig. 1. The LRP editor showing part of the running example of this text.

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. The editor, shown in Figure 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 one pane, the tree of nested machines is shown. The programmer selects which machine to visualize, in another 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.

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 would have no user interface and hence consume 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 the robot's world model, for example.

2.1 Language Design

LRP is a language that allows for the description of nested *state machines*. Nesting in LRP means that a given *state* may contain a complete state machine (whose states may again contain a machine, and so on). The 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 *on entry action*, an *on exit* action and a *running* action. Actions are snippets of code in the imperative object-oriented dynamically typed programming language Smalltalk[1]. When the state becomes the active state, its on entry 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 on exit 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 on exit 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 to ROS, or indeed 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, epsilon transitions trigger automatically when the state is active, *i.e.* after their on entry action is executed. Timeout transitions specify a timeout in milliseconds, either as a literal or a variable reference, and trigger after this timeout. Wildcard transitions have no specific source state, instead they consider all of the states in their machine as the source state.

A last construct is the bootstrapping construct that specifies the machine to *spawn* and which is the start state of that machine. One spawn construct can be

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

placed at top-level, and the on entry actions of states may spawn all machines that are lexically in scope.

2.2 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. 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 lightthresh := [128]) (var maxlook := [100])
2  (var step := [1]) (var back := [10]) (var turn := [100])
```

The first two lines of code show the declaration of variables, declaring five different variables: `lightthresh`, `maxlook`, `step`, `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.* `lightthresh` establishes the threshold between black and white for the light sensor.

For clarity we do not include here the code that creates the connection to ROS, nor the full code of how commands are published, as this is not key to the example. In a nutshell, we represent sensors and motors as variables: respectively `bumper`, `light` and `leftmotor`, `rightmotor`, `motors`. These can later be used inside actions. The initialization of these variables consists of code that connects to the appropriate ROS topics. For example, to be able to publish on the teleoperation topic of a command velocity multiplexer the code is as follows: `ROSbridge publish:'/cmd_vel_mux/input/teleop' typedAs: 'geometry_msgs/Twist'`. This code results in a Smalltalk object that can be assigned to a variable, *e.g.* `control`. This object can be used to publish messages of the `Twist` type on the `teleop` topic. For example, code to go forward could be: `control send: [ :message | message linear x: 10]` sends a Twist message where the `x` value of the linear part is 10, and all other values are 0.

Note that in the remainder of this text code between square brackets is in effect Smalltalk code. The result of this code is the result of the evaluation of the last statement. For example in the variable initialization cases shown above, 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 in detail here.

```
3  (machine follower
4    (state moving (running [motors forward: step]))
5    (on outofline moving -> looking t-looking)
6    (on intheline looking -> moving t-moving)
```

```
7    (event outofline [light read > lightthresh + 10])
8    (event intheline [light read < lightthresh - 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 `motors` topic, instructing to move forward for a `step` distance. 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 on occurrence of the event `outofline`, defined in line 7, and causes a transition from the `moving` to the `looking` state. It has as name `t-looking`. The second 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, which returns an integer value that is higher as the measured luminosity is higher. The `outofline` event on line 7 is triggered when reading the light sensor produces a value that is higher than the light threshold plus ten[2]. The intheline event lowers the threshold by ten and verifies the inverse.

```
9    (state looking
10     (machine lookalgo
11       (state lookleft (running [rightmotor forward: step]))
12       (state returnleft (running [rightmotor back: step]))
13       (state lookright (running [leftmotor forward: step]))
14       (state returnright (running [leftmotor back: step])
15         (onexit [time := time * 2 ]))
16       (var time := [maxlook])
17       (ontime time lookleft -> returnleft t-returnleft)
18       (ontime time returnleft -> lookright t-lookright)
19       (ontime time lookright -> returnright t-returnright)
20       (ontime time returnright -> lookleft t-lookleft))
21     (onentry (spawn lookalgo lookleft)))
```

The `looking` state defines a nested state machine, its definition spans the lines 10 through 20 above. Note that the rightmost column of Figure 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 11 through 15) 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 17 to 20. 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 15). As a result, the size of the sweeping motion doubles at the end of each sweep.

[2] This statement is evaluated by obtaining the `light read` value, summing the light threshold with ten, and testing the given inequality. This results in a boolean value.

Line 21 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 `t-moving` transition of line 6 to trigger. When this happens the `lookalgo` machine is discarded.

```
22    (var nobump := [true])
23    (event bumping [bumper isPressed & nobump])
24    (event ending [bumper isPressed & nobump not])
25    (on bumping *-> bumpturn t-bumpturn)
26    (on ending *-> end t-end)
27    (state bumpturn
28      (onentry [motors backward: back .
29            rightmotor backward: turn .
30            leftmotor forward: turn .
31            nobump := false .
32            LRP delaySec: 2]))
33    (eps bumpturn -> looking t-bumplook)
34    (state end)
35  )
36  (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 22 records if the robot has not yet bumped. The events at lines 23 and 24 occur on a bump sensor press combined with a logical `and` operation on the variable, in line 23, and the negation of this variable in line 24.

Lines 25 and 26 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 `bumpturn` state.

The `bumpturn` has a rather complex on entry action: Line 28 instructs both motors to move back for a distance of `back` (of line 2). Lines 29 and 30 turn the robot around, each motor traveling a distance `turn` (of line 2). Line 31 records the bump, and line 32 pauses the interpreter for two seconds. The latter is to allow enough time for all these actions to be exectued by the robot. Note that pausing the intepreter is possible because the on entry action happens atomically, *i.e.* no events are evaluated during this time and no actions are triggered. The pause also ensures that if the bumper is pressed during this maneuver, the state is not exited and entered again due to the `t-bumpturn` transition triggering. Exit from the state is instead provided by the epsilon transition on line 33. It executes after the on entry action completes and goes to the `looking` state.

The state in line 34 does nothing. Note that the state name `end` has no special status within the language. The last line of the program: 36, 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.

3 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 [9] sparked wide interest, and pioneering work [7] 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 2.2 (lines 10 to 20). 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.

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, suppose having the line on its left hand side. This triggers a transition to the `looking` state, stopping the robot. As the state has no behavior defined, the robot is frozen and the LRP interpreter visualization shows that no further state changes occur. Development of the looking algorithm may now start.

First, an empty nested machine `lookalgo` is added. In `lookalgo`, the states `lookleft` and `returnleft` and their timeout transitions are added (lines 11, 12, 17, 18). In the `looking` state the on entry spawn statement is added. This last edit changes the currently active state, which requires the interpreter to be reset (details on why are in Section 4.1). Interpretation starts in the `moving` state, causing the robot to move briefly forward before the interpreter goes to the `looking` state, as the robot is still out of the line. 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 18 referes to a state that does not exist yet! (Details on handling of incorrect code is given in Section 4.2). This means that the robot will never stop its returning motion. Seeing this, the programmer pauses the interpeter, 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 15. The interpreter is unpaused, the robot looks to the right and finds the line.

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. Alternatively, the programmer may provide a general solution in the form of the `time := time * 2` on exit action

of line 15 omitted above. In that case the sweeps of the robot will progressively get larger, which is visible immediately after exiting the `returnright` state.

4 How the LRP Interpreter Ensures Liveness

Running an existing program is not the essence of live programming. The essence is running a program *while it is being changed by the programmer*. 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 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.

4.1 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 analysed 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 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 on exit 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 an element, *e.g.* changing the name of a state, is equivalent to a removal and an addition operation on this group. A straightforward case of this is changing the name of a transition: the old transition is removed and the new one is added, going from the same state as the old one and to the same state as the new one. The effects of other changes are not so straightforward, yet still sensible: Changing the name of an event 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 causes its incoming and outgoing transitions to no longer be valid, as these use the name of a state that no longer exists. Lastly, and most significantly, changing the initialization value of a variable 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.

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 ends up in an undefined condition is when an active state or an active machine is removed. In this case, and only this case, interpretation of the program needs to resume from scratch. All other changes allow interpretation of the program to seamlessly continue.

4.2 Dealing with Program Errors

Code that is syntactically correct, and hence ran by the interpreter, may however still contain errors. For example, in Section 2.2 we have seen a transition that specifies the name of a destination state that does not exist yet. The interpreter should nonetheless keep running, allowing the programmer to keep entering 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.
- 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 notify when it encounters such errors, using a minimally intrusive error window that auto-closes after a timeout, in the style of MacOS notification banners.

5 Related Work

The Kouretes Statechart Editor [8] 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.

Niemüller *et al.* [6] propose the implementation of behaviors in a general-purpose scripting language. It is however unclear from the text what the concrete syntax for state machine description is, nor what features the system supports. Their tools do provide support for visualization of the state machines, including showing the current state and last taken transitions as the program runs. There is however no support for updating the state machine while it runs, which is fundamental to live programming.

Live programming was first proposed by Tanimoto [7], where the goal was set to provide a maximum of feedback to the programmer while a program is being constructed. 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 [2].

Outside of visual programming, the SuperGlue language [5] is a textual language that is also based on dataflow programming and extended with object-oriented constructs. Burckhard *et al.* add live programming features for UI construction to an existing live programming language [1]. The work of Victor [9] showcases various examples of live programming in Javascript that produce pictures, animations and games. It can be credited for sparking wide-scale interest in Live Programming, which helped the crowdfunding of Light Table [3]: an editor that adds live programming features to a number of general-purpose languages.

None of these languages however consider state machines as their computational model, which is why we consider them radically different of LRP.

6 Conclusion and Future Work

In this paper we presented LRP: a live programming nested state machine language, with a connection to ROS. As a result, it permits live programming of robot behaviors. We have given an overview of the language, shown how it helps development, and discussed how its state machine interpreter achieves liveness.

An interesting observation resulting from this work is that nested state machines are actually quite robust 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, it is feasible to use LRP as a general nested state machine language.

Immediate future work is the implementation of refactorings that allow renaming, *e.g.* of states, allowing some interpreter restarts to be avoided. Longer term goals include testing the limits of expressibility of the language by building a wide variety of behaviors, and adding new language features as required.

Acknowledgments

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. We thank ESUG (http://esug.org) for providing sponsoring for this article.

References

1. Burckhardt, S., Fahndrich, M., de Halleux, P., McDirmid, S., Moskal, M., Tillmann, N., Kato, J.: It's alive! continuous feedback in UI programming. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 95–104. PLDI '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2491956.2462170
2. vvvv group: vvvv - a multipurpose toolkit, web page http://www.vvvv.org/
3. Kodowa Inc: Light table – the next generation code editor, web page http://www.lighttable.com/
4. Lötzsch, M., Risler, M., Jüngel, M.: XABSL - A pragmatic approach to behavior engineering. In: Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS). pp. 5124–5129. Beijing, China (2006)
5. McDirmid, S.: Living it up with a live programming language. In: Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. pp. 623–638. OOPSLA '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1297027.1297073
6. Niemller, T., Ferrein, A., Lakemeyer, G.: A Lua-based behavior engine for controlling the humanoid robot Nao. In: Baltes, J., Lagoudakis, M., Naruse, T., Ghidary, S. (eds.) RoboCup 2009: Robot Soccer World Cup XIII, Lecture Notes in Computer Science, vol. 5949, pp. 240–251. Springer Berlin Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-11876-0_21
7. Tanimoto, S.: VIVA: A visual language for image processing. Journal of Visual Languages & Computing 1(2), 127–139 (June 1990), http://dx.doi.org/10.1016/S1045-926X(05)80012-6
8. Topalidou-Kyniazopoulou, A., Spanoudakis, N.I., Lagoudakis, M.G.: A case tool for robot behavior development. In: Chen, X., Stone, P., Sucar, L., Zant, T. (eds.) RoboCup 2012: Robot Soccer World Cup XVI, Lecture Notes in Computer Science, vol. 7500, pp. 225–236. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-39250-4_21
9. Victor, B.: Inventing on principle. Invited Talk at CUSEC'12, video recording available at http://vimeo.com/36579366