Towards Live Programming in ROS with PhaROS and LRP

Pablo Estefó[1], Miguel Campusano[2], Luc Fabresse[1],
Johan Fabry[2], Jannik Laval[1], and Noury Bouraqadi[1]

***Abstract*— In traditional robot behavior programming, the edit-compile-simulate-deploy-run cycle creates a large mental disconnect between program creation and eventual robot behavior. This significantly slows down behavior development because there is no immediate mental connection between the program and the resulting behavior. With live programming the development cycle is made extremely tight, realizing such an immediate connection. In our work on programming of ROS robots in a more dynamic fashion through PhaROS, we have experimented with the use of the Live Robot Programming language. This has given rise to a number of requirements for such live programming of robots. In this text we introduce these requirements and illustrate them using an example robot behavior.**

I. Introduction

In Live Programming [7], the development cycle is made extremely tight: program edits are continuously integrated in the always-running program and their effects are immediately visible. As a result, there is no cognitive dissociation between writing the code and observing its execution, and hence programmers have an immediate connection with the program they are making.

In this paper, we present early experiments introducing live programming into the development cycle of robotic applications. We target the ROS [6] middleware because it is a defacto standard that moreover has an availability of numerous packages provided by an active community backed by the Open Robotics Software Foundation.

However, ROS does not provide any support for live programming. The development cycle of ROS introduces a clear cut between the edit, compile, and run steps of software development. This division already appears in its core concepts since a ROS package is a static entity, whereas a ROS node is a run-time entity.

Nodes to run as well as their name spaces and parameters are statically expressed in a .launch file and cannot be changed at run-time. So, the whole architecture of a ROS-based application is static and, traditionally, any small change requires restarting the entire application.

As a first step in exploring live programming with ROS, we combined the Live Robot Programming (LRP) language [3]: a high-level DSL for live programming of robots, with PhaROS [2]: a bridge between ROS and the dynamic language Pharo [1], [5]. This work served as the foundation for the integration of ROS in LRP. As a result of these experiments, we have encountered a number of requirements for live programming with LRP and ROS, and we present them in this text.

II. Requirements for Live Programming Robots with ROS

A ROS application is a graph of nodes connected by topics. Enabling liveness in ROS implies making it possible to dynamically change any part of the graph. This means :

Requirement 1: Starting and stopping individual nodes or subsets of nodes as many times as required during the application lifetime. For instance, when two nodes are communicating through a topic and the subscriber is shut down, when relaunched it should reconnect and resume communication.

Requirement 2: Changing node parameters at run-time after the node has been launched.

Requirement 3: Partially or fully replacing node code at run-time without stopping it.

Requirement 4: Changing node connections at run-time (*i.e.* the edges of the graph). For any given node, we should be able to dynamically change topics to which it subscribes or publishes.

[1]Dépt. IA - Ecole des Mines de Douai
`firstname.lastname@mines-douai.fr`
http://car.mines-douai.fr
[2]PLEIAD and RyCH labs, Computer Science Department (DCC), University of Chile, Chile

III. Proposed Solution: LRP+PhaROS

Our solution is implemented using the Pharo dynamic language. It consists of the LRP DSL combined with PhaROS: a ROS client for Pharo.

A. LRP

Live Robot Programming (LRP) [3] is a live programming language, of nested state machines, together with an interpreter and visualization (all of which is implemented in Pharo). This kind of machines are said to map well to the problem domain, which is corroborated by the fact that multiple Robocup teams have performed outstandingly in the football competition using these languages [4], [8].

The main differentiator of LRP with regard to other languages using the nested state machine paradigm is its nature as a live programming language. Because of this, LRP code is *always* running and LRP has a well-defined semantics for how it behaves in the face of code that is syntactically malformed, incomplete or erroneous programs, and changes to the code while a program is running. Put briefly, as a live programming language, the priority is to keep the program running in spite of a wide variety of errors. Also, program changes are integrated while the code runs, only restarting program execution when this integration cannot be performed. A full discussion on these features and the language in general is however outside of the scope of this paper. For more information we refer to published work [3], as well as the LRP website[1].

LRP is designed for the programming of robot behaviors, and the communication between LRP and the actual robot is through the use of APIs of specific robot platforms or middleware. The integration of ROS in LRP [3] was achieved based on the results of the experiments we report here.

The LRP language is built based on the following concepts:

- Machines with states and different kinds of transitions.
- Transitions that occur on events, timeouts or immediately after entering in a state.
- Action blocks are snippets of Smalltalk code that have access to the complete Smalltalk environment, as well as LRP variables.
- Variables can be defined in machines, are initialized when declared using an action block, and are lexically scoped.
- Events are explicitly defined, triggering if their action block evaluates to true, *i.e.* any piece of Smaltalk code can serve as a guard for an event.
- States can have action blocks that are run when entering, leaving or when they are active, again enabling any piece of Smalltalk code to be used.
- States can also define state machines, which enables nesting.

[1]Website URL: http://pleiad.cl/lrp

B. PhaROS

PhaROS [2] provides a framework and a set of tools for developing ROS nodes in the Pharo language and environment. Pharo is a pure object-oriented programming language coupled with a dynamic environment.

In PhaROS, ROS nodes are reified as objects, allowing the developer to build and control their execution at a higher level of abstraction. PhaROS also contains tools allowing the deployment of catkin packages: It automatizes the generation of xml launch files, makefiles, type and scripts creation. The intent of this is to let the programmer focus on programming and not on creation and maintenance of infrastructure.

PhaROS nodes can be restarted as many times as needed, due to the dynamic nature of the Pharo environment. Instances of ROS nodes can be terminated at any time and are garbage collected. New instances can be launched directly from Pharos without a need to restart ROS as they transparently reconnect with external existing ROS topics or parameters.

C. Integration of LRP and PhaROS

As said above, LRP relies on the existence of an API to the robot (middleware) being used. In our experiments we worked with a Robulab robot that runs ROS and therefore we needed to construct such an API.

To do this, we implemented a bridge module to ROS as well as a specific interface for the Robulab. The former takes the role of a facade class (facade in Figure 1). It provides access to external ROS resources such as topics or parameters. The latter, a RobulabBridge class, is tailored to

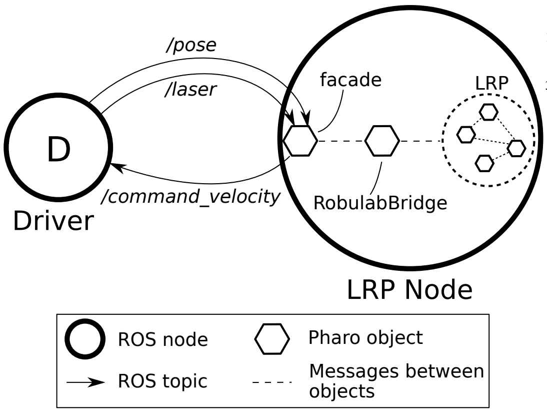

Fig. 1. ROS graph of application shown in experiment.

the Robulab and provides an API for the features specific to the robot, hiding the particularities of ROS. Note that this abstraction of specifying an API specific to the features of the robot can be generalized to any ROS node, where the facade could be simply reused.

The robot's node (Driver in Figure 1) consumes messages on the /command_velocity topic and publishes messages on the /laser and /pose topics. Hence, the RobulabBridge provides methods that wrap messages to and from these topics such as:

- **forward: linearSpeed** publishes on the /command_velocity topic to make the robot move linearly forward at a linearSpeed speed.
- **turn: angularSpeed** similar to forward:, making the robot rotate at angularSpeed.
- **isThereARightObstacle: minimumDistance** selects laser data corresponding to the front-right part of the laser beam, and checks if there is an obstacle at a distance less or equal than minimumDistance.

IV. Experiment: writing an obstacle avoider

In this section, we report on an experiment written in LRP that runs on top of PhaROS and therefore uses ROS. The example we present is simple, but interesting nonetheless because it exemplifies what is possible with live programming. Videos of the experience are available on http://car.mines-douai.fr/ .

In the experiment, we will change a mobile robot's obstacle avoidance behavior incrementally at runtime. First we will implement a simple solution that lets the robot move without crashing into obstacles. After that the behavior will be evolved by adding obstacle avoidance. All of this will be done without needing to restart any part of the ROS platform or equipment used, all software changes are immediately perceived as robot behavior changes.

```
(var f_vel := [0.25])
(var t_vel := [0.5])
(var min_distance := [0.5])
(var robulab := [RobulabBridge uniqueInstance])
```

Listing 1. Initialization of the live programming session

The experiment was performed using a Robulab robot which can be translated and/or rotated on the floor. The robot is equipped with a Sick S300 laser range sensor, which detects obstacles up to 30 meters within 270 degrees around the robot. Technical specifications of the robot are available on its website[2].

A. Stop when an obstacle is detected

First, we will define the four variables that we will use during the experiment. Listing 1 shows how they all are defined. f_vel and t_vel are both linear and angular velocities for the robot respectively (set to 0.25 $[m/s]$ and 0.5 $[rad/s]$ respectively) and min_distance (set to 0.5 $[m]$) is the minimum distance to a physical object to be considered as an obstacle. The fourth variable is initialized with a reference to an instance of RobulabBridge class, presented before. Note that t_vel it is not used in the first behavior we implement, but it could be added later at runtime.

The first behavior to describe in LRP is to make the robot move forward and stop when an obstacle is detected in front. So a simple program should consider two states: forward and stop (lines 2 to 5 in Listing 2). When entering the forward state, due to the statement in line 3 the program will send the message forward: with f_vel as argument to the robulab variable.

As a result the robot will move forward with a speed of f_vel $[m/s]$. When the robot detects that there is an obstacle at distance of min_distance

[2]Website URL: http://www.robosoft.com/products/indoor-mobile-robots/robulab/robulab-10.html

```
1   (machine Tito
2     ( state forward
        ( onentry [robulab forward: f_vel] ))
4     ( state stop
        ( onentry [robulab stop] ))
6     (on obstacle forward —> stop t—stop)
      (on noObstacle stop —> forward t—forward)
8     (event obstacle
        [robulab isThereAnObstacle: min_distance])
10    (event noObstacle
        [(robulab isThereAnObstacle: min_distance) not ])
12  )
    (spawn Tito forward)
```

Listing 2. First behavior of state machine describing its states, transitions and events in LRP

```
1     ( state turnLeft
       ( onentry [robulab turn: t_vel] ))
3     ( state turnRight
       ( onentry [robulab turn: t_vel negated] ))
5     (on rightObstacle stop —> turnLeft t—lturn)
      (on leftObstacle stop —> turnRight t—rturn)
7     (on noObstacle turnLeft —> stop t—tlstop)
      (on noObstacle turnRight —> stop t—trstop)
9     (event rightObstacle [robulab isThereARightObstacle:
          min_distance])
      (event leftObstacle [robulab isThereALeftObstacle:
          min_distance])
```

Listing 3. Aditional code for including obstacle avoidance behavior

$[m]$ or closer, the obstacle event will occur, as specified in line 8, and the machine will perform a state change from forward to stop through the t-stop transition (line 6). When entering the stop state (line 5) the message stop is sent to the robulab making the robot stop moving. If suddenly the obstacle disappears, the event notObstacle occurs (lines 10-11), and through the t-forward transition, the machine changes to forward state making the robot move again.

The last line of Listing 2 initializes the state machine in the forward state.

B. Avoiding obstacles

Now we have the robot stopped in front of an obstacle, so the current status is stop. Next, a simple obstacle avoidance behaviour is added: the robot will turn left or right when an obstacle is detected and when there is no obstacle it will move forward, as already defined.

We define two more states: turnLeft and turnRight (lines 1 to 4 in Listing 3). They are reached by a simple obstacle detection algorithm, and in them the message turn: is sent to robulab with the same speed (t_turn) but a different direction depending on the state. Considering the obstacle detection algorithm, the RobulabBridge provides useful methods to know if the obstacle is in the left or the right side of the front of the robot. These are used to emit rightObstacle or leftObstacle events when needed (lines 9 through 10). These events make the machine change from stop to the turning states by the t-tlturn and t-trturn transitions (lines 5 and 6). When no obstacle is detected, noObstacle is raised and machine changes the state to stop, via the transitions of lines 7 and 8, which will be immediately followed to a transition to forward via t-forward.

When writing this code, immediately when the lines of codes defining the rightObstacle or leftObstacle events are added, one of these occur (since the robot is stopped at an obstacle). This then makes the robot turn to avoid the obstacle, and move forward when the obstacle is avoided.

V. Requirements Evaluation

In this section, the fulfilment of requirements listed in Section II is discussed.

Requirement 1: This requirement is fulfilled for PhaROS nodes. In PhaROS nodes are objects with specific methods for both initializing and finalization. Once they are finalized, they do not continue working nor interacting with the rest of the ROS platform. Nodes can be restarted multiple times, reconnecting to necessary topics and start publishing immediately.

Requirement 2: Typically, node parameters are statically defined in a .launch file and once launched, they are cannot be changed. In contrast, in PhaROS a ROS node is an object and their parameters are materialized as fields accesible through methods. As a consequence, they can be changed at run-time either by any piece of Smalltalk code, for example inside an action block of an LRP program. Furthermore, parameters can also be changed at startup when using the rosrun command using command-line options for changing parameters (rosrun package node _parameter=value).

Requirement 3: The logic behind the use of data from subscriptions, its processing and publishing can be modified when the PhaROS node was already launched. This is illustrated in detail in Section IV as the response of the robot to obstacles evolved at runtime. More specifically, the developer tuned robot behavior in an iterative development process to get the expected response.

Requirement 4: By default in ROS, node connection relies on topic names that are hard-wired inside the code of nodes. Still, at deployment time this can be altered using namespaces. Changing node connections at run-time means changing topic names and namespaces at run-time. PhaROS provides runtime subscription/publishing which implies that topics are created when the ROS node is being executed.

VI. Related Work

To the best of our knowledge, LRP is the only work that proposes the live programming of robot behaviors through nested state machines. Live programming was originally proposed by Tanimoto on Viva [7], a visual programming language for image manipulation. We are aware of two DSLs for robot behavior programming based on nested state machines: the Kouretes Statechart Editor [8] and XABSL [4].

VII. Conclusion

In this work we have presented four requirements to support live programming of robots in ROS: starting and stopping ROS nodes, changing node parameters, hot code swapping and connections between nodes being made at run-time. We illustrated the need for these requirements through an example.

We have shown that the solution we propose fulfills these four requirements for ROS nodes created in PhaROS.

Moreover, we found that developing robot behaviors in the LRP language was quite straightforward. The straightforward use of state machines avoids the programmer losing focus on the task at hand due to setup or technical issues intervening.

Also, the RobulabBridge class made LRP code easy to write and understand, as it abstracts the robot resources and provides access through a rich API. Finally, we consider that the solution of LRP through PhaROS is an effective approach for enabling live programming for ROS applications.

References

[1] A. Bergel, D. Cassou, S. Ducasse, and J. Laval. *Deep Into Pharo*. Square Bracket Associates, 2013.

[2] S. Bragagnolo, L. Fabresse, J. Laval, P. Estefó, and N. Bouraqadi. Pharos: a ros client for the pharo language. http://car.mines-douai.fr/category/pharos/, 2014.

[3] J. Fabry and M. Campusano. Live robot programming. In A. Bazzan and K. Pichara, editors, *Advances in Artificial Intelligence - IBERAMIA 2014*, number 8864 in Lecture Notes in Computer Science. Springer-Verlag, 2014. To Appear.

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

[5] O. Nierstrasz, S. Ducasse, and D. Pollet. *Pharo by Example*. Square Bracket Associates, July 2010.

[6] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

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

[8] A. Topalidou-Kyniazopoulou, N. I. Spanoudakis, and M. G. Lagoudakis. A case tool for robot behavior development. In X. Chen, P. Stone, L. Sucar, and T. Zant, editors, *RoboCup 2012: Robot Soccer World Cup XVI*, volume 7500 of *Lecture Notes in Computer Science*, pages 225–236. Springer Berlin Heidelberg, 2013.