Live Programming in Practice: a Controlled Experiment on State Machines for Robotic Behaviors

Miguel Campusano[a,*], Johan Fabry[b], Alexandre Bergel[a]

[a] *ISC lab, Computer Science Department (DCC), University of Chile, Beauchef 851, Santiago Chile*
[b] *Raincode Labs, Brussels, Belgium*

Abstract

Context: Live programming environments are gaining momentum across multiple programming languages. A tenet of live programming is a development feedback cycle, resulting in faster development practices. Although practitioners of live programming consider it a positive inclusion in their workflow, no in-depth investigations have yet been conducted on its benefits in a realistic scenario, nor using complex API.

Objective: This paper carefully studies the advantage of using live programming in defining nested state machines for robot behaviors. We analyzed two important aspects of developing robotic behaviors using these machines: program comprehension and program writing. We analyzed both development practices in terms of speed and accuracy.

Method: We conducted two controlled experiments, one for program comprehension and another for program writing. We measured the speed and accuracy of randomized assigned participants on completing programming tasks, against a baseline.

Results: In a robotic behavior context, we found that a live programming system for nested state machine programs does not significantly outperform a non-live language in program comprehension nor in program writing in terms of speed and accuracy. However, the feedback of test subjects indicates their preference for the live programming system.

Conclusions: The results of this work seem to contradict the studies of live programming in other areas, even while participants still favor using live programming techniques. We learned that the complex API chosen in this work has a strong negative influence on the results. To the best of our knowledge, this is the first in-depth live programming experiment in a complex domain.

Keywords: Live Programming, Controlled Experiment, Robot Behaviors, Live Robot Programming, Nested State Machines

*Corresponding author
Email addresses: mcampusa@dcc.uchile.cl (Miguel Campusano), jfabry@gmail.com (Johan Fabry), abergel@dcc.uchile.cl (Alexandre Bergel)

1. Introduction

In Live Programming [1], software development is augmented by performing continuous real time modifications of running programs, which are accompanied by a visualization of their execution. The goal of live programming is to improve development by providing immediate feedback. This is done on the one hand by integrating code changes without the need for restarting the program, and on the other hand by the visualization reflecting the state of the running program. A central tenet of live programming is that since this feedback allow developers to immediately see the results of their changes, they will better understand the code, yielding higher productivity.

To the best of our knowledge, there have been no studies on the advantages of a live programming environment in a more practical settings. With this we mean a nontrivial context where there are dependencies on possibly complex external API's, as well as the study considering both code comprehension and code writing. Such experiments are however needed to establish whether the advantages of live programming that have been shown in a restricted context are also applicable in practice.

We have therefore researched live programming 'in practice' for both program comprehension and code writing. For an 'in practice' setting we choose the context of programming the behaviors of robots using discrete events. To do this, several software abstractions have been proposed, and one notable approach is based on hierarchical state machines, *e.g.*, XABSL [2] and the Kouretes Statechart Editor [3]. These machines are used to program behaviors used for the robots in the RoboCup competition, with notable results.

Work in the area of robotic behaviors consists of creating the overall way of behaving of a robot, taking into account inputs generated by external algorithms such as image recognition and map localization, and sending instructions to, *e.g.*, a robot navigation or object grasping algorithm. The robotics research community makes such external algorithms available online, relying on middleware to enable the joint operation of all these pieces of computation. ROS [4] is the current de-facto middleware in robotics. It is a publish-subscribe middleware with support for over 100 different robots, and an extensive software ecosystem providing a wide variety of possibilities for robot behavior programming. The behavior program interfaces with the different robotics algorithms through the ROS middleware. This results in a possibly complex API between the program and the external functionalities, important in an experiment for programming 'in practice'.

ROS allows for two different ways to program robot behaviors using nested state machines: the arguably popular non-live system SMACH [5] and our live programming system LRP [6]. We hence performed user studies comparing LRP with SMACH, to establish if robotic behavior program understanding and writing benefits from live programming.

We performed two controlled experiments with within-subjects designs, each to evaluate the accuracy and speed of development using live programming versus a classical means for robot behavior programming. The first experiment considers program comprehension and the second experiment considers program writing. In this paper we present the results of our experiments.

Notably, the quantitative results of the experiment show us that there is no difference in program understanding nor in program creation, both considering correctness and time taken. However, the qualitative results contradict this, for both experiments the subjects prefer to use LRP over SMACH. Our observations on subject behavior lead to three possible causes for the low performance of LRP: Live programming is not the panacea of software development, the complexity of the ROS middleware and missed opportunities by the test subjects.

The paper is structured as follows: In the next section we present the live programming system LRP, followed by the definition of our baseline: SMACH. Then, we detail the overall experiment design. Section 5 details our first experiment on program comprehension, and Section 6 talks about our second experiment on code writing. We follow with an overall discussion of our results, before treating threats to validity. Lastly, Section 9 discusses related work, followed by conclusions and future work.

2. Live Robot Programming

LRP is a live domain specific programming language (live DSL) for the specification of the behavior of robots [6]. A DSL is designed to take advantage of the problem domain in their syntax and semantic to improve the developers productivity [7]. The development of software for robotics can be improved by using a DSL for a specific robotic domain [8, 9, 10].

An LRP program is the textual definition of a tree of nested state machines. LRP adds two language features to the arguably well-known model of nested state machines: lexically scoped variables and second, several types of actions defined in states. Actions are Pharo Smalltalk [11] blocks code that have access to the variables in scope. Actions can be executed when entering or exiting a state, or executed in a loop when being in a state. All interaction with the robot is performed inside these Pharo Smalltalk blocks, even guards for transitions. As a consequence, LRP has no binding to a specific robotic API or middleware.

Transitions are connection between two states. When a state is executing, the program can change to another state only defined by the transitions that starts from the current state. When one of these transitions trigger, the current state of the program changes to the one defined by the transition. LRP provides several types of transitions to wrap common user cases when program robotic behaviors:

- *normal transitions*: They are defined with a block code as a guard. When the block returns true, the transition triggers, changing the current state.
- *epsilon transitions*: They trigger automatically, they are equivalent to using normal transitions with a block code that always returns true.

- *time transitions*: These transitions are used to wait on a state for a defined period of time. The time is defined in the definition of the transition. After the time passed, the transition triggers, changing the current state.

- *wildcard transitions*: They work in a similar way as the normal transitions, but they do not have an initial state. They may trigger in any state of the machine. They are specially useful when the behavior has emergency cases (*i.e.*, a button to stop the behavior, no matter which state the behavior is executing).

- *exit transitions*: They are used to exit a nested machine. They go from a state inside the nested machine to a state in the parent machine.

The live nature of LRP allows for the direct construction, visualization and manipulation of the program's run-time state. While the program is running, LRP allows the program to be modified at run-time without losing the program state. This means that developers can add, remove and modify parts of the program without always needing to restart it to apply them.

The direct manipulation in LRP is achieved firstly by having its integrated development environment (IDE) visualize the running state machine as it is being programmed. Secondly, the IDE also shows the values of variables at runtime and these values can be inspected and manipulated in the IDE without needing to change program code. Thirdly, developers can "jump" to other states while the program is running. When the state machine is in one state, a developer can force it to immediately transition to any other state in the program, even when there are no connections between these states.

For security reasons, LRP provides a pause option to stop some of the liveness features of LRP while writing a program. This pause stops the execution of the code, while keeping most of the other liveness features. In particular, the visualization of the program keeps updating, but the program will not execute new code on the robot.

An example video of LRP is found online: http://bit.ly/LRPVideo. In this video we show how the visualization evolves while a developer writes a program.

While LRP is not coupled to a specific robot API, it does provide out-of-the box support for four robot APIs. It provides support for ROS [4], the de-facto standard middleware for robotics, the Aldebaran NAO robot[1], the Parrot AR.Drone[2] and the Lego Mindstorms EV3[3].

The ROS bridge is provided by PhaROS [12], a Pharo Smalltalk API for ROS. When this bridge is activated, LRP starts up with a separate UI that allows developers to create subscriptions and publishers to receive and send data to the robot via ROS. When LRP uses the ROS bridge, a pseudovariable called

[1] https://www.ald.softbankrobotics.com/en/cool-robots/nao
[2] http://developer.parrot.com/
[3] https://education.lego.com/mindstorms

robot is accessible in the source code. The data received from the robot is accessed via this pseudovariable that also has methods to allow sending commands to the robot.

A full introduction to LRP is outside of the scope of this paper. For more information we defer to the published literature [6, 13] and the official website: http://pleiad.cl/lrp.

3. Baseline: SMACH

SMACH [5] is a library for building robot behaviors in Python when using the ROS [4] robotic middleware. ROS is the de-facto standard middleware in research on autonomous robots, and SMACH is its standard solution for writing behavior as nested state machines.

The core of SMACH is independent of ROS. The machines and states provided by SMACH are Python classes.

The states in SMACH are subclasses of *SimpleState*. These classes must define the method *execute*. This method is executed when the state is active, returning an *outcome*. The outcome is a simple string that is used to select the transition that triggers after the execution of the *execute* method.

Even when the core of SMACH is independent of ROS, there are several classes that make it easier to compose SMACH with ROS. One of the most important used in the experiments is the state *MonitorState*. This state monitors a data from the robot. It receives a callback function that is executed when the data from the robot is received. This callback function should return *True* or *False*.

When using ROS with SMACH, it is possible to access to the SMACH visualization: *smach_viewer*. This visualization shows the static representation of the state machine, highlighting the state that is being executed.

We present an example video of a program being built in SMACH in: http://bit.ly/SMACHVideo. In this video we show the same example presented in Section 2 for LRP. For brevity we did not record the program being written from scratch.

Again, a full introduction to SMACH is outside of the scope of this paper. For more information we refer to the published literature [5] and the official website: http://wiki.ros.org/smach.

4. Overall Experimental Design

The two controlled experiments we conducted share a common design, which is described in this section. Particularities of each experiment are described in Section 5 and Section 6. We mostly follow the work of Jedlitschka and Pfahl to report our studies [14]. To no repeat the same through both experiments, we grouped information of the two into one in this section where possible.

Both controlled experiments are within-subjects design, *i.e.*, subjects participating in all treatments and we measure all the participations of the subject

in every treatment. We then cross-evaluate LRP with the baseline and measure the impact of using LRP.

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

4.1. Goals

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

4.2. Dependent and Independent Variables

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

4.3. Baseline

We select SMACH as a baseline for our experiments because beyond its popularity, Python and SMACH have key similarities with LRP: the language is dynamically typed, SMACH programs are written as nested state machines with approximately the same features as LRP, and SMACH also provides a visualization of the running machines. We presented a full introduction to SMACH in Section 3.

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

4.4. Experiment Design

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

To minimize noise resulting from the technical aspect of running robots, we conducted our experiment in a simulated environment, the Turtlebot robot[4] in

[4] http://wiki.ros.org/Robots/TurtleBot

the Gazebo robotics simulator[5]. Note that simulation of robot behavior as part of the development process is a standard procedure in the robotics community as it allows one to abstract from hardware issues and accelerate the development process.

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

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

4.5. Participants

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

As far as we know, the only people in Chile that work with ROS and robotic behaviors using state machines are students from the University of Chile. In particular, there is a robotic team in the university, the UChile Homebreakers team[6]. This team participates in the RoboCup@Home league, made up of Electrical Engineering students.

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

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

4.6. Task Setup

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

4.7. Work Session

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

[5] http://gazebosim.org/
[6] http://robotica-uchile.amtc.cl/about.html

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

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

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

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

5. Controlled Experiment: Program Comprehension

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

5.1. Goal

In this experiment we answer the following research questions:

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

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

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

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

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

5.2. Experiment Design

As mentioned in Section 4, we designed a controlled experiment using cross-evaluation, which requires two tasks per experiment. The tasks of this experiment are:

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

Figure 1 is a screenshot of Task A in SMACH and Figure 2 is a screenshot of the same task in LRP.

We randomized the participation of the subjects in every work session (Work Sessions are explained in Section 4.7) while having similar number of participants per work session. We designed 4 different work sessions. Each participant participates in exactly one of the four work sessions. The different work sessions are:

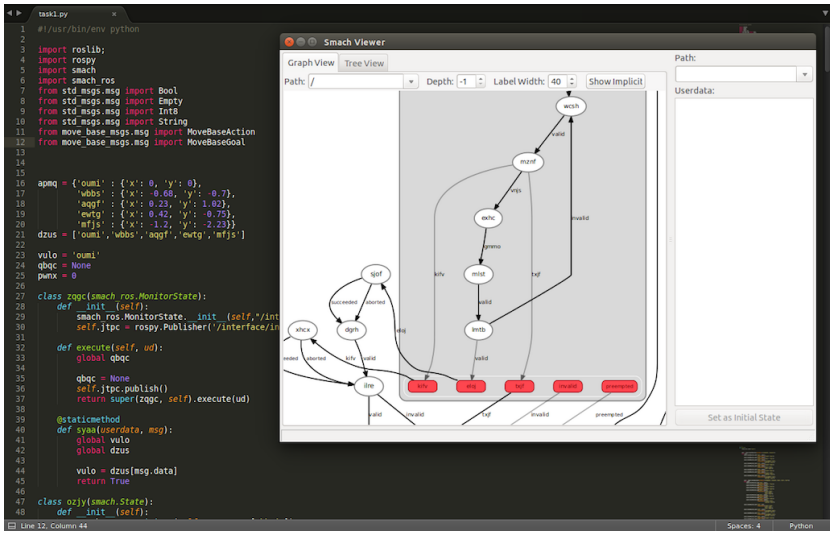

Figure 1: Task A of program comprehension experiment in SMACH.

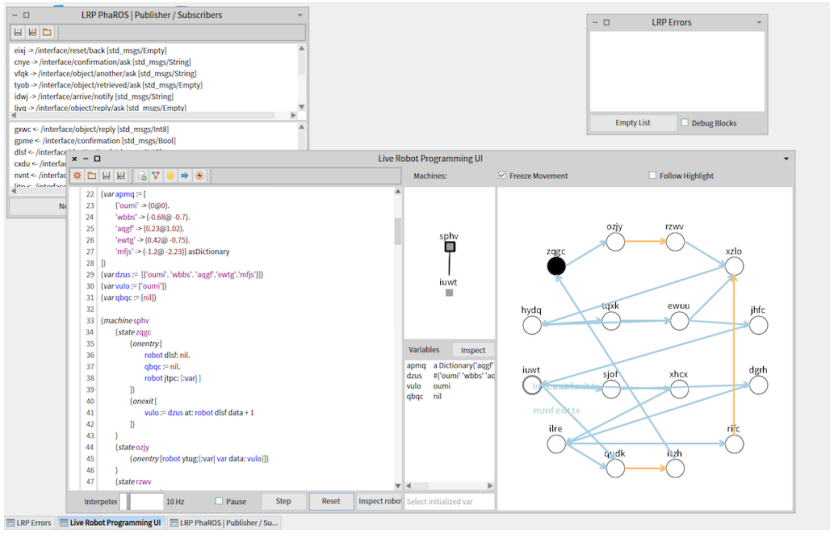

Figure 2: Task A of program comprehension experiment in LRP.

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

5.3. Pilot Study

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

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

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

5.4. On Reducing Biases

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

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

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

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

We excluded functionalities that could be interesting but may benefit the performance of one of the system over the other. For example:

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

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

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

5.5. Work Sessions

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

For the LRP system, the Development Space contains the IDE and a window showing all the communication channels used by the program to the robot. In the IDE the developer can see the source code of the program and the live visualization. For the SMACH system, the Development Space has the source code in a Sublime Text editor[7] and a terminal with two tabs. One of the tabs is ready with the command to run the program and the other tab is ready with the command to open the run-time visualization. When the visualization is open, it is also displayed in the Development Space.

[7] https://www.sublimetext.com

5.6. Warm-up Phase

In this experiment, the warm-up phase consists in reading reference material and solving an exercise. In the exercise, the subjects extend the example presented in the reference material, focusing on several features that the subject needs to know before doing the experiment. These features are:

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

5.7. Evaluation phase

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

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

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

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

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

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

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

Table 1: Quantitative results of the program comprehension experiment. We can see the average score of LRP is slightly better than the average score of SMACH. SMACH average time is slightly better than LRP average time.

```
with sphv:
    smach.StateMachine.add('X', X(),
      transitions={'itul': 'Y', 'fjhd': 'Z'})
```

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

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

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

5.8. Results

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

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

Table 2: Mann-Whitney, two-tailed test for several data combinations of the program comprehension experiment. The data combinations presented are: LRP vs SMACH in score and time to complete the tasks; score of Task A vs Task B of LRP and SMACH. With this test, we see no significant difference in any data combination for the program comprehension experiment: every P value is greater than 0.05. We present the U value for better comparison of the statistical test used here.

Two of the subjects state that they have a Beginner level of SMACH and 2 more state they have a Knowledgeable level of SMACH. Three of them state that they have a Knowledgeable level of LRP. The other subjects state that they do not have any knowledge of LRP or SMACH, however, all of them state that they have knowledge of Python, from Beginner to Advanced.

5.8.1. Quantitative Results

We present the quantitative data of the experiment on Table 1. The LRP and SMACH columns are the scores obtained on the questionnaire for the tasks carried out using LRP and SMACH respectively. The table presents the time to complete the tasks done in LRP and SMACH. The second column shows the type of work session of the subject. The last column indicates whether the participant has some experience using LRP.

We analyze these results considering four data combinations: score and time between Task A and Task B, and score and time between SMACH and LRP.

At the end of Table 1 we can see the average, median and standard deviation of the data. These figures give a slight advantage to LRP in accuracy but a slight faster speed on SMACH.

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

In Table 2 we first see the values of the Mann-Whitney, two-tailed test for LRP vs SMACH. In this table we present two values: the probability value P and the Mann-Whitney statistic value U. The P value allows us to test the statistical significance of the data, when $P < 0.05$, we claim that the data is significantly different with a confidence level 95%. The U value is a component of the Mann-Whitney test used to calculate the P value: the smaller the value of U, the smaller the value of P. In Table 2 we reported the U value for comparison when replicating the Mann-Whitney test.

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

Table 3: Statistical results of the program comprehension experiment removing participants with LRP experience. The average score of LRP improves, but the average score of SMACH diminishes from Table 1. The average time of LRP and SMACH improve from Table 1, with SMACH being faster again.

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

Table 4: Mann-Whitney, two-tailed test for score and time of LRP vs SMACH for the program comprehension. These results do not take into account participants with LRP experience. As in Table 2, there is no significant difference in score nor time: $P > 0.05$.

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

We have produced two tasks, A and B, completed in both LRP and SMACH. Lastly, we have compared the results of tasks A and B for each treatment, as seen in the last two rows of Table 2. This figure indicates that both task A and B are similar since $P > 0.05$.

5.8.2. Quantitative results: Participant with no LRP Experience

In Table 1 we see that 3 participants had experience in using LRP before the experiment was conducted. We made an analysis of the data without these 3 participants.

Here we use the same analysis used with the original results. Without taking account the experienced LRP participants, LRP still has a slight advantage in terms of correctness against SMACH. However, SMACH improves a lot its speed compared to LRP. We can see these results in Table 3.

In Table 4 we can see, without taking account participants with experience using LRP, that there is no statistical differences of the score and time using LRP or SMACH of the completed tasks ($P > 0.05$).

5.8.3. Qualitative Results

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

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

Table 5: Qualitative data of the program comprehension experiment. The worst result is 1 and the best is 5. The average results give a preference of LRP over SMACH for the subjects.

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

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

5.9. Observations on Subject Behavior

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

programming advantage, given that the subjects did not modify the programs at all.

5.10. Conclusions on Program Comprehension

The quantitative results of the experiment disallow us to reject both null hypotheses H_{01} and H_{02}: there is no difference between using LRP or SMACH to understand complete programs for robotic behaviors, both considering correctness and time taken. In contrast to this, the qualitative results of the experiment reveal that the subjects actually think that the answer of the research question is positive. Concretely, the subjects' opinions are that it is easier to understand programs written in LRP and moreover they prefer to use LRP over SMACH.

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

6. Controlled Experiment: Program Writing

The goal of our second experiment is to measure the accuracy and speed of writing a program for robot behavior using live programming. As mentioned in Section 4, we compare LRP with SMACH using a within-subjects design. The complete programs for both tasks, questionnaires and scans of filled-in questionnaires are available to download at `http://bit.ly/2jbixXD`.

6.1. Goal

We aim to answer the following research questions:

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

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

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

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

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

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

6.2. Tasks to Evaluate

We used some elements of the program comprehension experiment explained in Section 5 and simplified them. In this experiment, each program is divided in 4 steps, where the first step (Step 0) is a warm-up session where subjects get used to the development environment. Each step represents a part of a complete program, by finishing the last step, the subjects complete the overall behavior. After the warm-up step, the next 3 steps have an increasing level of difficulty and come with a time limit: 10, 20 and 30 minutes each. The last step combines multiple functionalities that are used in the previous steps, adding new functionalities only presented in this step.

We present the two tasks here:

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

A comparative screenshot of the complete Task A in SMACH and LRP can be seen in Figure 3 for SMACH and Figure 4 for LRP. These images present the program and visualization of Task A for SMACH and LRP respectively.

We randomized the participation of the subjects in every work session (Work Session are explained in Section 4.7) while having similar number of participants per work session. We designed 4 different work sessions. Each participant participates in only one of the four work sessions. The different work sessions are:

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

6.3. Pilot Study

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

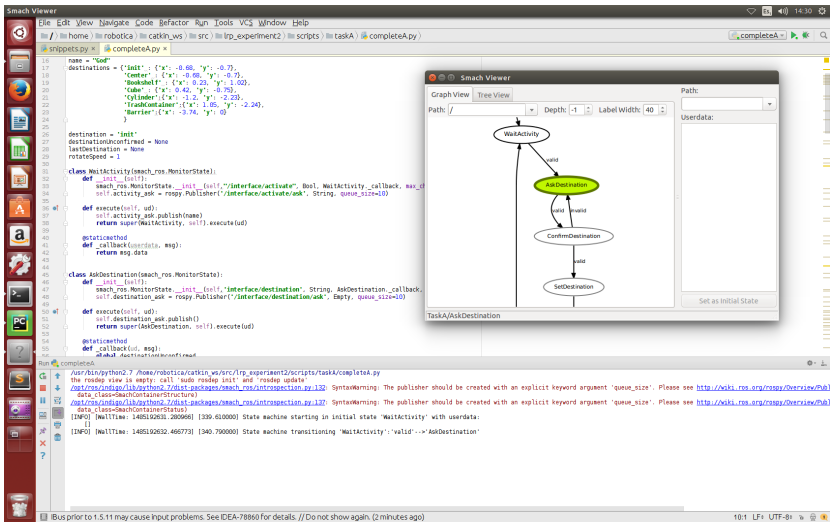

Figure 3: Complete program of Task A in SMACH for program writing experiment.

6.4. Reducing Bias

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

6.4.1. Reducing bias of used features

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

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

6.4.2. Reducing different starting point bias

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

By introducing a new program at every step, we may introduce a bias of understanding in the different systems. If the base program is more difficult to understand in one system, then the subjects may have an additional problem to extend it in this system. However, in our experiment of program comprehension, presented in Section 5, we have seen that there is no significant difference in

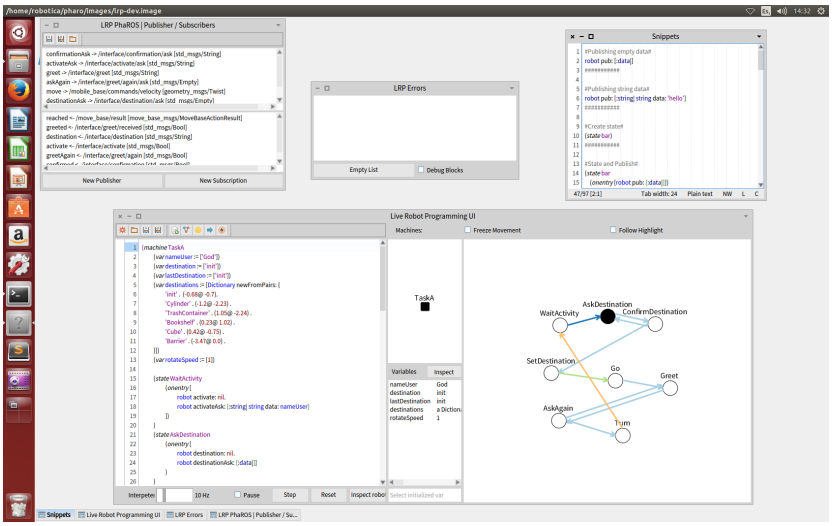

Figure 4: Complete program of Task A in LRP for program writing experiment.

program understanding between LRP and SMACH. We can therefore use this way of working without adding any significant bias.

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

6.5. Work Sessions

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

The computer with one screen represents the development environment. For LRP, this screen contains the IDE to write programs, the error window and the window that creates the communication channels between the robot and LRP. Additionally, it contains another window with code snippets of recurrent code patterns in LRP. For SMACH, the development environment contains, in full screen, PyCharm, a Python IDE[8]. In PyCharm there are always two open tabs, one with the current program being written, and another with code snippets of recurrent code patterns similar to those in LRP. Finally, there is a terminal ready to execute the SMACH visualization.

6.6. Warm-up phase

In the warm-up exercise in this experiment the subjects should write a program in the system, instead of on a paper. We decided to go with this because

[8] https://www.jetbrains.com/pycharm/

this would be the first attempt of the subjects to write a program and use the system. This may introduce some beginner errors and we want to reduce such a threat in our evaluation.

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

6.7. Evaluation phase

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

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

With every step we measure the correctness and the time to complete the step. Every step has a time limit: 10, 20 and 30 minutes respectively. When subjects could not complete all the steps, we scored only the correct parts they managed to write. Recall that in Section 6.4.2, we mentioned that at the beginning of each step every subject starts with the same base program, hence not completing a previous step does not introduce a strong bias in the results.

Each step has a different scoring function due to the difference in difficulty. The maximum points for each step are: 8, 11 and 19. Because the systems are different, it is not trivial to be fair in scoring each task. To do this we divided each program in smaller sections and scored them independently. The division is: sending data to the robot, receiving data from the robot and extra functionalities[9].

6.8. Results

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

All the participants, except one, declared having no knowledge of SMACH, while 2 of them stated that they have a Beginner level of LRP. Notably, only 2 students declared not having any knowledge in robotic development. Moreover,

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

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

Table 6: Score for every step of the program writing experiment, for both LRP and SMACH. Σ represents the sum of scores of the three steps. We can see LRP has a slight advantage over SMACH in tasks S2, S3 and the sum of the three tasks. SMACH has a slight advantage in task S1.

all of the students declared to have at least a Beginner level of Python, with 2 of them declaring an Advanced level in Python.

6.8.1. Quantitative Analysis

Table 6 and Table 7 present the raw data. Table 6 gives the score for every step and the sum of the three steps. Table 7 indicates the time to complete each step and the total time to complete the three steps. If a subject did not finish the step, we show this with an "x". The sum column considers an "x" as the maximum time allowed for completing the task (S1 = 10 minutes, S2 = 20 minutes, and S3 = 30 minutes). For both tables, the last column presents if the participant has some experience using LRP. Because it may not be fair to give the maximum time allowed when summing the time of the steps, Table 8 presents the normalized time of completed steps only.

In Table 6 and 7 we present the average, median and standard deviation of the data. With these figures we can see a slight advantage to LRP for the sum score of the tasks. The average and median score for S2 and S3 is higher with LRP than with SMACH. For the sum time, participants completed their tasks slightly faster using LRP. This claim is true for each task, except for step S2, where time LRP and SMACH are tied. When comparing only the time of completed steps by normalizing the time between 0 and 1, we can see that participants still completed their tasks slightly faster using LRP, as shown in Table 8.

Similar to the previous experiment, we measured the normality of the data before measuring the significance between the score and time values. Not all

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

Table 7: Time to complete the program writing experiment. The letter "x" means that a step is not finished within the time limit. Σ represents the total time to complete the three steps. Whenever there is an "x", Σ considers the maximum time per step. We can see LRP has a slight advantage over SMACH for task S1, S3 and the sum of the three tasks. LRP and SMACH are tied in task S2.

the data sets are normal. We therefore employ, as the previous experiment, the Mann-Whitney test. Between the sum score and the sum time, there is no statistical significance of using LRP vs SMACH: $P > 0.05$ (In Section 5.8.1 we explained the P and U values of the Mann-Whitney test). Table 9 presents this data.

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

We also compared the score of tasks A and B for each treatment to measure if the two tasks are comparable, as seen in the last two rows of Table 9. These two test results indicate that both tasks A and B are comparable and similar in their difficulty, hence reducing the bias that one task could be more difficult than the other.

6.8.2. Quantitative results: participants with no LRP Experience

In the Tables 6 and 7 we see that 2 participants had experience in using LRP before the experiment was conducted. We analyzed the data without these 2 participants.

Without taking into account the experienced LRP participants, LRP still has a slight advantage in terms of correctness and speed against SMACH, while SMACH is more stable when measuring time. For each task, LRP has a slight advantage for S2 and S3, while SMACH has a slight advantage in S1, for both

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

Table 8: Table 7 with normalized times between 0 and 1. There are 14 and 15 completed steps for LRP and SMACH respectively. We consider only the completed tasks to calculate the average, median and σ. We can see that LRP has a slight advantage over SMACH.

correctness and speed. Table 10 and Table 11 presents this data. Even in Table 12, with the statistical results of the normalization of the time between 0 and 1, LRP keeps its slight advantage over SMACH, and SMACH remains more stable.

As with the data including the participants with LRP experience, the Mann-Whitney test indicates that scores and time between LRP and SMACH are not significantly different. Table 13 shows the results of the test.

6.8.3. Qualitative Analysis

The raw data of the subjects' opinions is in Table 14. All the questions presented here have a score from 1 to 5.

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

We asked the subjects if the liveness features of LRP help them to write and debug programs. Notably, every subject answered positively to both questions. Moreover, LRP and its liveness features has a lot of positive comments

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

Table 9: Mann-Whitney, two-tailed test for several data combinations of the program writing experiment. The data combinations presented are: LRP vs SMACH in score and time of the tasks; score of Task A vs Task B of LRP and SMACH. With this test, P value is greater than 0.05 for every there is no significant difference in any data combination for the program writing experiment. The U value is also given for further reference.

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

Table 10: Statistical results of the program writing experiment removing participants with LRP experience. The table shows the score of the every task and the sum of the tasks. The average scores are similar to the one presented in Table 6. Again, LRP has the advantage on tasks S2, S3 and the sum of the scores, and SMACH has the advantage on task S1.

(translated from Spanish): *"LRP is good because it allows one to jump between states"*, *"(with liveness) it is easier to see which step causes the problem"*, *"the jump function is for finding out if a state is working correctly"*, *"(with liveness) I can focus on the local problem instead of the whole process"*, *"(with liveness) it is easy to change values and immediately see what happens"*, *"the program shows you what it is doing"*. There is also feedback to take into account when developing in LRP: *"The tool (LRP) seems good, but it is difficult to use in the beginning"*, *"I need to get used to the LRP syntax"*, *"It (LRP) seems like a good tool, however (...) it takes time to get used to it"*.

6.9. Observations on Subject Behavior

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

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

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

Table 11: Statistical results of the program writing experiment removing participants with LRP experience. The table shows the time to complete every task and the sum of the times. The average times are similar to the one presented in Table 7. In this case, LRP has the advantage on tasks S2, S3 and the sum of the times, while SMACH has the advantage on task S1. In this case, SMACH is more stable than LRP, with almost half the value of the standard deviation.

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

Table 12: Normalized times of Table 8, while removing LRP expert subjects. The table shows the statistical results of normalized time of completed steps only. Again, LRP has the advantage over SMACH, however, the advantage is slower than Table 8. SMACH keeps being more stable with, again, almost half the standard deviation.

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

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

6.10. Conclusions on Program Writing

Like the previous experiment, the data of the experiment does not allow us to reject the third and fourth null hypotheses H_{03} and H_{04}. There is no difference

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

Table 13: Mann-Whitney, two-tailed test for score and time of LRP vs SMACH for the program writing experiment. These results do not take into account participants with LRP experience. As in Table 9, there is no significant difference in score nor time: $P > 0.05$ for both time and score.

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

Table 14: Subjects opinion of the program writing experiment. The worst is 1 and the best is 5. As in the experiment of program comprehension, the average results give again a preference of LRP over SMACH.

between using LRP or SMACH to write programs for robotic behaviors, both considering correctness and time taken.

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

7. Discussion

In this section we explain in depth in the liveness features of LRP that should help developers and why we believe robotic behavior development is a difficult task, even when using live programming.

7.1. Why should LRP perform better?

There are several works stating the advantages of live programming [15, 16] and how developers behave in these systems [16, 17, 18]. We designed LRP with liveness features and conducted our own set of experiments, however we did not find any statistical differences using live programming in robotic development,

even when we strongly believe that LRP has several features that should help developers.

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

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

However, LRP has its own disadvantages as well:

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

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

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

7.2. Why does LRP NOT perform better?

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

The recent work of Kubelka *et al.* [19] offers insights on why live programming does not work. They found that some developers do not use liveness features where these features should help them.

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

Live programming does not help in all development tasks. Considering program comprehension (Section 5), we already explained that live programming does not help because subjects do not modify the program to understand it. The root cause of this may be that only two of the subjects are accustomed to working in a live programming environment. Hence most subjects do not change a program in order to observe these changes in the behavior. However, the two subjects with experience did not make any change either, so there may be other factors at play as well.

Complex, low-level API. In the experiment of program writing (Section 6), we explained how the impact of live programming may be reduced because of the use of the ROS middleware. The ROS API is complex to use by itself.

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

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

The missed opportunities are especially noticeable in Step 3 of each task - the most extensive and difficult one for the program writing experiment. Remember that this step includes the turning behavior, which exemplifies a case of where live programming is said to be advantageous (as explained in Section 6.7). We thought that subjects would skip over parts of the program to just test this behavior, using the "jump" feature of LRP (explained in Section 2). Eventually, subjects did use the feature, but just four of them, only after they had already reached that part of the program through normal execution. This again reduces the impact of liveness features. Remarkably, in SMACH we encountered a different strategy to save time testing this behavior. Three out of ten participants set the turning state as the initial state of the machine to test it, minimizing the time needed to test this particular behavior. Moreover, even when they did not do that, the program was not time consuming enough to lead to a significant performance impact.

On a positive note, the subjects did use the liveness features in Step 1 and Step 2 of the experiment in program writing, although on a smaller scale. We believe these programs are not big enough to notice a real impact of live programming, since code complexity is low and hence it is straightforward to build a mental model of this code. In our experiment we could not make the subjects build bigger programs because of time constraints. We did expect to see a significant impact of liveness in the last step, but, as we explained before, we encountered other obstacles.

7.3. How to improve our experiments?

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

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

8. Threats to Validity

As in any experiment, there are several threats to validity in this work. We analyze four kinds of threats, as classified by Wohlin *et al.* [20].

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

For the experiment in program writing we gave the subjects a base program to start each step (more on this in Section 6). This may benefit one system if the task of understanding the base program is more difficult on one of the systems. However, the results of the experiment in program understanding (explained in Section 5) show us there is no such difference.

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

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

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

Finally, we believe the ease of building a robot behavior is influenced by the robot API. In this case, our opinion is that the ROS API factor minimizes the effect of using live programming (as explained in Section 7.2).

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

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

The effect size is a threat to validity too. The lack of participants (10 per experiment) may lead to a lack of generalization of our results. However, even with the small amount of participants no difference is perceived, not even a trend. We do not believe increasing the amount of participants will change our conclusions. Live programing does not help with everything, nor everyone and it would have problems with more complex API (as explained in Section 7.2).

Also, the complexity of the settings may introduce a problem to other researches when trying to replicate or generalize these results. To minimize this, we use a simulator of a real robot to remove the bias of working with real hardware and to make the experiment easier to replicate by just using computers, without the need of a real robot.

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

9. Related Work

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

As part of the validation of Interstate Oney *et al.* [15] performed a comparative laboratory study where Interstate was tested against JavaScript. This study had 20 participants and consisted of 2 tasks where participants needed to make modifications and express new behaviors. There is however no description of how the experiment was conducted, nor how the tasks were divided amongst the participants. The conclusions of the experiment were that Interstate is faster than JavaScript to make modifications to already existing programs and to express new behaviors. There is however no report of a study that measures the accuracy or speed of Interstate in understanding existing code.

The work of Wilcox *et al.* [16] revealed that continuous visual feedback in direct manipulation of programs helps in the accuracy of debugging certain tasks. They compare a live version of Forms/3 [21], against another version with immediate feedback removed. In this experiment there are 29 subjects, where half of them work on two different tasks using first the live version and then the non-live version, while the other half do exactly the opposite. They use two completely different tasks for this study, one to emphasize a graphical program, and the other to emphasize a mathematical program. The degree of improvement, as the authors conclude, depends on the type of problem, the type of user, and the type of bug.

Kramer *et al.* [17] complemented the work of Wilcox *et al.* [16] by analyzing code creation, comparing a live version against a non-live version of JavaScript. In this work the authors analyzed 10 subjects where each subject should solve three different tasks: one to parse an RSS feed, one to convert between two object representations of a date, and one to implement Dijkstra's algorithm. The authors noticed that while live programming did not speed up the time to complete a task, it did significantly decrease the time for fixing bugs introduced while writing the task. They state that they found no indication that live programming speeds up the process of code creation because of the small sample size and/or that the data is overshadowed by inter-subject differences.

In addition to above small studies on whether live programming improves the speed in development, there are three studies about how developers behave in a live programming environment [16, 17, 18]. These show that developers interact more with the live systems by performing more changes in programs or by regularly checking the code for correctness. The rationale is that this is because these systems promote more interaction between developers and their programs. The recent work of Kubelka *et al.* [19] also shows how developers behave in a live programming environment, including more practical tasks of fixing bugs in a real-unknown system and extend a familiar system using liveness

features.

Lastly, Hundhausen and Brown [22] investigated the impact of continuous feedback on novice programmers. To do this each subject was exposed to three different systems: No feedback, self-select feedback and automatic feedback. In the first system the subjects mentally simulated the program, in the second the subject explicitly requested syntactic and semantic feedback, and the third system the subject had feedback with every keystroke. The subjects completed three tasks that involved creating, populating and iterating over arrays. The authors found that even when subjects did significantly better in the systems with feedback, there was no difference between both feedback treatments. Note that liveness implies continuous and meaningful feedback [18] and that is one of the conclusions of the work of Hundhausen and Brown [22]: *"rather than coming up with ways to facilitate liveness (in terms of feedback), programming environment designers ought to be putting their efforts into designing effective semantic feedback that benefits users"*.

None of the previous experiments focus on the impact of live programming on robotic behaviors nor on the use of a more complex system, except for the work of Hancock [18] that focuses on teaching children to program robot behaviors. However, Hancock only presents a study on how children behave using his system in robotics, with no comparative analysis on the impact of live programming in the development of robot behaviors. Moreover, the system that connects with the robot is much less complex than using ROS, which may explain why this work does not report on the problems we explain in Section 7.

10. Conclusion and Future Work

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

The quantitative results of the experiment show us that there is *no significant* difference in program understanding nor in program creation, both considering correctness and time taken. This novel result seems to contradict the previous works in live programming.

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

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

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

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

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

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

Our future work consists of further studying the performance of developers using a live programming system in practical settings, specially in robotic behaviors. We wish to confirm our hypothesis that a complex API reduces the positive impact of live programming, and investigate how we can reduce the amount of missed opportunities of liveness features. After these studies are done, we will have a more solid ground to design and conduct new experiments, with subjects with several backgrounds, and finally asset the impact of live programming on behavior-based robotics.

Moreover, liveness is said to help also in the task of debugging programs in practical settings. Part of our future work is to test if live programming helps to debug robotic behaviors written as state machines. However, we first need to research the impact of a complex API in a live programming setting. Maybe the complex API could also obscure, as with the experiments presented in this work, the advantages of live programming in debugging programs.

Acknowledgments

We thank Lam Research for partially sponsoring the work presented in this paper. Miguel Campusano is also funded by CONICYT-PCHA/Doctorado Nacional/2015-21151534.

References

[1] S. L. Tanimoto, Viva: A visual language for image processing, Journal of Visual Languages & Computing 1 (2) (1990) 127–139.

[2] 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.

[3] A. Topalidou-Kyniazopoulou, N. I. Spanoudakis, M. G. Lagoudakis, A case tool for robot behavior development, in: X. Chen, P. Stone, L. Sucar, T. Zant (Eds.), RoboCup 2012: Robot Soccer World Cup XVI, Vol. 7500 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013,

pp. 225–236.
URL http://dx.doi.org/10.1007/978-3-642-39250-4_21

[4] 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.

[5] J. Bohren, S. Cousins, The smach high-level executive [ros news], IEEE Robotics & Automation Magazine 17 (4) (2010) 18–20.

[6] M. Campusano, J. Fabry, Live robot programming: The language, its implementation, and robot API independence, Science of Computer Programming 133 (2016) 1 – 19.

[7] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, G. Wachsmuth, DSL engineering: Designing, implementing and using domain-specific languages, dslbook. org, 2013.

[8] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, M. Ziane, Robotml, a domain-specific language to design, simulate and deploy robotic applications, in: International Conference on Simulation, Modeling, and Programming for Autonomous Robots, Springer, 2012, pp. 149–160.

[9] U. P. Schultz, D. J. Christensen, K. Stoy, A domain-specific language for programming self-reconfigurable robots, in: Workshop on automatic program generation for embedded systems (APGES), 2007, pp. 28–36.

[10] A. Nordmann, S. Wrede, J. Steil, Modeling of movement control architectures based on motion primitives using domain-specific languages, in: Robotics and Automation (ICRA), 2015 IEEE International Conference on, IEEE, 2015, pp. 5032–5039.

[11] A. P. Black, O. Nierstrasz, S. Ducasse, D. Pollet, Pharo by example, Lulu. com, 2010.

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

[13] J. Fabry, M. Campusano, Live robot programming, in: A. Bazzan, K. Pichara (Eds.), Advances in Artificial Intelligence – IBERAMIA 2014, no. 8864 in LNCS, Springer-Verlag, 2014, pp. 445–456. doi:http://dx.doi.org/10.1007/978-3-319-12027-0_36.

[14] A. Jedlitschka, D. Pfahl, Reporting guidelines for controlled experiments in software engineering, in: Empirical Software Engineering, 2005. 2005 International Symposium on, IEEE, 2005, pp. 10–pp.

[15] 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.

[16] E. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, C. R. Cook, Does continuous visual feedback aid debugging in direct-manipulation programming systems?, in: Proceedings of the ACM SIGCHI Conference on Human factors in computing systems, ACM, 1997, pp. 258–265.

[17] J.-P. Kramer, J. Kurz, T. Karrer, J. Borchers, How live coding affects developers' coding behavior, in: 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE, 2014, pp. 5–8.

[18] C. M. Hancock, Real-time programming and the big ideas of computational literacy, Ph.D. thesis, Massachusetts Institute of Technology (2003).

[19] J. Kubelka, R. Robbes, A. Bergel, The road to live programming: insights from the practice, in: Proceedings of the 40th International Conference on Software Engineering, ACM, 2018, pp. 1090–1101.

[20] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.

[21] M. Burnett, R. Walpole Djang, J. Reichwein, H. Gottfried, S. Yang, Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm, Journal of Functional Programming 11 (2001) 155–206.

[22] C. D. Hundhausen, J. L. Brown, An experimental study of the impact of visual semantic feedback on novice programming, Journal of Visual Languages & Computing 18 (6) (2007) 537–559.