Does Live Programming Help Program Comprehension?

A user study with Live Robot Programming

Miguel Campusano
PLEIAD and RyCh labs,
Computer Science Department
(DCC), University of Chile, Chile

Alexandre Bergel
PLEIAD lab,
Computer Science Department
(DCC), University of Chile, Chile

Johan Fabry
PLEIAD and RyCh labs,
Computer Science Department
(DCC), University of Chile, Chile

Abstract

A tenet of Live Programming is that its tightening of the development feedback loop results in better program comprehension and hence higher developer productivity. There are however no extensive reports published on user studies that validate this claim when considering already existing code. In this paper we report on a controlled experiment that establishes whether our live programming language, LRP, helps in program understanding when compared to a non-live language and toolkit. We furthermore obtained qualitative feedback from the test subjects on their preferences between the two systems. Remarkably, while the users prefer the live system over a non-live system, the actual level and speed of program comprehension is the same for both systems.

1. Introduction

In Live Programming (Tanimoto 1990), software development is augmented by performing continuous real time modifications of running programs and by an always present visualization of the running program. The goal of live programming is to improve program comprehension 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 allows developers to immediately see the results of their changes, they will better understand the code, yielding higher productivity.

As part of our research on software engineering for robotics we aim to speed up the development of behaviors of robots. Using live programming to yield such a speedup was therefore an appealing idea and motivated us to build a live programming language for robotics: LRP (Fabry and Campusano 2014; Campusano and Fabry 2016). In LRP robot behaviors are written as state machines, with the main difference over existing solutions being the live nature of the language. In LRP the running state machines are continuously updated as the program is changed, and they are visualized in a purpose-built visualization.

As part of the validation of LRP we are performing a number of user studies to establish if there indeed is a productivity advantage of using it instead of the common programming languages and tools. In this paper we report on our user study that treats program comprehension of existing code. This user study is however relevant beyond simply evaluating LRP. This is because, to the best of our knowledge, there has been no extensive report published of a study on the productivity advantages of live programming for code comprehension of existing code.

Our user study is a controlled experiment using within-subjects and repeated measures. We implemented two different tasks, each in two different systems: LRP and the baseline system. We measured correctness of program understanding as well as time taken, and obtained qualitative results through a post-experiment questionnaire.

We find that the results of our experiment are noteworthy: the test subjects' performance is actually unchanged between the two systems, yet they nonetheless still express a clear preference for using LRP over the common programming language and toolset.

2. The Systems Under Comparison

2.1 Live Robot Programming

LRP is a live programming language for the specification of the behavior of robots (Fabry and Campusano 2014; Campusano and Fabry 2016). A LRP program is the textual definition of a tree of nested state machines. To the arguably well-known model of nested state machines, LRP adds lexically scoped variables and actions in states and transitions. Actions are blocks of Smalltalk (Goldberg and Robson 1980)

code. These can be guards for transitions, 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 actions and as a consequence LRP has no binding to a specific robotic API or middleware.

The live nature of LRP allows for the direct construction, visualization and manipulation of the program's run-time state. This direct manipulation 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.

While LRP is not coupled to a specific robot API (and not fundamentally restricted to the development of robot behaviors) it does provides out-of-the box support for three robot APIs. First and foremost, it provides support for ROS (Quigley et al. 2009), de-facto standard middleware for robotics. Second is support for the Lego Mindstorms EV3[1] robot, and third LRP also supports the Parrot AR.Drone[2].

A full introduction to LRP is outside of the scope of this paper. Instead we refer to the published literature (Fabry and Campusano 2014; Campusano and Fabry 2016) as well as the LRP website: `http://pleiad.cl/LRP` for videos.

2.2 Baseline: SMACH

In ROS (Quigley et al. 2009), a popular library for developing robot behaviors in Python is SMACH[3]. We focus on this library as a baseline to evaluate the understanding of robotic behaviors. This is because, beyond its popularity, Python and SMACH have key similarities to 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 wish to compare LRP and SMACH as complete systems and skip comparing individual features. This notwithstanding that LRP has features that are not present in SMACH (especially considering liveness). Comparing both systems is ambiguous in terms of which specific features are better, however both systems can be compared in terms of effectiveness and efficiency. If one system is better than the other we can not be sure which features are the most effective but we still can be sure about which system, as a whole, is better for program comprehension.

3. Experiment Design

We performed a controlled experiment using within-subjects and repeated measures to gauge whether code written in a live programming language is easier to understand than code using a conventional language and toolkit for robotic development. More specifically, we cross-evaluated LRP with Python and SMACH and measured the effectiveness and efficiency of understanding an existing complete program.

Put differently, our dependent variables are the correctness of the understanding of the program, *i.e.* the effectiveness in understanding programs, and the completion time, *i.e.* the efficiency in understanding programs. Our independent variables are the systems used in the experiment, LRP and SMACH, and the two different programs that subjects will try to understand.

3.1 Tasks to Evaluate

We performed a cross-evaluation experiment and implemented two different programs that subjects should understand. These programs specify two robot tasks. For the sake of the experiment, we did not perform the experiment on real hardware but we simulated a Turtlebot robot[4] in the Gazebo robotics simulator[5]. Note that simulation of robot behavior as part of the development process is standard procedure in the robotics community as it allows one to abstract from hardware issues and accelerates the development process. For both tasks we also implemented a user interface for the robot. This is a simple textual interface where the robot can ask the users to take certain actions and to provide them with information.

We now present a summary of the two different tasks:

Task A In this task the Turtlebot performs an object delivery service. First, the robot is idle waiting for instructions. The program defines 4 possible places where the Turtlebot can go. The user should input a place for the robot to move to. This movement can then be seen in the simulation. When the robot arrives, it asks for an object to the user that called it (the *caller*), this action is simulated by pressing *Enter* in the interface. After this, the robot asks for a destination to deliver the object. When the robot arrives to the desired destination, it asks the user who receives the object (the *receiver*) to take the object. Again, this action is simulated by pressing *Enter*. Then, the receiver can: send another object back, return the robot to the caller without an object or return the robot to its original position. If the robot returns to the caller, the caller can: send another object to the receiver or return the robot to its original position. Whenever the robot returns to its original position, it will wait for someone to call it again, repeating the behavior.

Task B In this task the Turtlebot broadcasts a message. First the robot asks the user for a message to be delivered. Then the robot distributes the message to 4 different locations. In each of these places, the receiver of the message has a possibility to tell the robot that he/she does not understand the message. If the receiver does not understand the message, then the robot goes back to the user who origi-

[1] `https://education.lego.com/mindstorms`

[2] `http://developer.parrot.com/`

[3] `http://wiki.ros.org/smach`

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

[5] `http://gazebosim.org/`

nally provided the message to the robot where it asks for a new message. The circuit then is resumed, but using the new message instead of the old one.

3.2 On Reducing Biases

Reducing bias of difficulties and feature use 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 fast 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 where 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 by just looking at meaningful names. We see this as a 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 move to. 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 may 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. We want the developers to experience this difference by, as before, not only looking at meaningful names, but looking at the program structure, further removing bias.

3.3 Work Session

We performed the controlled experiment one subject at a time, to better be able to analyze how each subject works with both systems. For every subject called, we assigned them a type of work session out of 4 possible types:

- **Work Session 1:** Task A in LRP - Task B in SMACH
- **Work Session 2:** Task A in SMACH - Task B in LRP
- **Work Session 3:** Task B in LRP - Task A in SMACH
- **Work Session 4:** Task B in SMACH - Task A in LRP

We assign a subject to a work session in a way that, in the end, we have a similar number of subjects for every type of work session. Every work session has an approximate duration of 4 hours, with each task an approximate duration of 2 hours. We included a 15 minutes break between the two tasks to help to avoid that the subject was too tired at the end of the experiment. To make the experience more comfortable for the subject, we offered beverages and candies. In addition, to motivate subjects to participate in this long experiment, we give a monetary incentive of approximately 7 EUR and announced that we will gather all subjects and offer them pizza when all experiments are completed.

The activity of each work session is structured as follows:

- Answer a questionnaire with personal information and background. We call this phase *Pre-Questionnaire* phase
- Read description of the system used in the first task. We call this phase *Warm-up* phase for the first task
- The first system is evaluated where the subject should answer a questionnaire. This is the *Evaluation* phase for the first task
- 15 minutes break
- Warm-up phase for the second task
- Evaluation phase for the second task
- Answer a questionnaire with qualitative information about the subject and the experiment itself. We call this phase *Post-Questionnaire* phase

Pre-Questionnaire Phase In this phase we ask the subject to fill out a form with personal information. This form is anonymous and we do not collect information that allow us to identify the subject completely. For example, we asked for the age and the education level of the subject. Besides, we also asked for his/her previous knowledge of several tools used in the experiment. This information is important because we need to know if people with similar backgrounds have similar results in the experiment.

Warm-up Phase The subjects may not be used to program robotic behaviors using nested state machines. To achieve

a sufficient level of knowledge such that the participants are prepared for the study, before starting a task we give the subject reference material of the system to be used. 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. For both kinds of reference materials this example is the same. The reference material can be used by the subject in the experiment at any time. The materials also contain an explanation of the communication API to the robot used by the program.

At the end of every material there are two exercises for the subject to resolve. The exercises are about extending the example presented in the reference material. The exercises are focused on several features that are mandatory for the subject 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

With these exercises we make sure that the subject understands. The reference material is optional to read, but the resolving of the exercises is mandatory.

Evaluation Phase 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 [6] 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 it is opened, it is also displayed in the Development Space.

For both tasks we give a questionnaire to the subject. The subject should try to understand the program to answer the different questions of the questionnaires. At the end of each task, we collect the questionnaire to see how many correct answers the subject has, and also note how much time the subject takes to finish the questionnaire. The contents of the questionnaires are described in Section 3.4.

There may be a learning effect in this experiment since subjects may do better in the second task because they already answered the first one, and have experience with what is shown in the robot space. To reduce this possible bias, the work sessions were designed to use a cross-evaluation system, where a group of subjects, for example, first work in Task A in LRP, then Task B in SMACH. This group is complemented by another group of subjects that first work in Task B in SMACH, then Task A in LRP.

Post-Questionnaire Phase After both tasks are completed, *i.e.* at the end of the experiment, the subject is asked to fill a final form. In this form we ask for qualitative data about the experiment, to be able to establish the *feeling* of the subjects about the experiment, the different systems used and their results. This includes, amongst others, the following kinds of questions:

- **Time pressure:** We ask which if the subject felt any time pressure. Even though the experiment did not have an explicit time limit, subjects may feel pressure because of the duration of the experiment. This also gives us an insight on how comparable the tasks are.
- **Difficulty of tasks**: For each task, we ask how difficult the subjects find solving the tasks. We use a five-point Likert scale where 1 means the task is Extremely easy and 5 means the task is Extremely Hard.
- **How easy it is to understand programs written in a system**: For LRP and SMACH we asked the subject how easy it was to understand the program in that particular system, using the same scale as the previous question. This gives us insight about how the subjects feel using the systems, without comparing the systems themselves.
- **Comparing systems:** We ask whether the first tool is better than the second tool for that task (phrased in those terms to reduce acquiescence bias). This is the only question that explicitly compares both systems and asks the subjects for a preference. Here we also use a five-point Likert scale: when subjects mark 1 they Strongly Disagree with the statement (the first tool is better than the second tool), and subjects mark 5 when they Strongly Agree with that statement.
- **Use the system again:** For each system, we ask if the subject would use the system again to work with robotic behaviors. These are two separate questions since the same subject may want to strongly use both systems in the future, for example. We use the same five point Likert scale of agreement as above.
- **Extra comments**: At the end we give the subjects a space where they can give us comments and feedback about the experiment, the systems (LRP and SMACH), the tasks and anything the subject may want to share.

3.4 Questionnaire

For each of the tasks the subjects answered a questionnaire that serves to measure how well they understand the

[6] `https://www.sublimetext.com`

program. Each questionnaire has 17 questions, designed to cover the different strategies we expected the subject would use in a program comprehension task. The questions for the different strategies are as follows:

- Easy to solve using only the visualization.
- Focusing on the states of the machine and the nested machine, *i.e.* focusing in the behavior of the robot.
- Focusing on the transitions of the machine and the nested machine:
  - A specific question about how much time the program waits before or after something happens.
  - Questions about what should happen in the program to make the program go from one state to another.
- A specific question about a behavior of the program that involves a specific execution path.
- A specific question about looking at program variables.
- Questions only answerable by running the program.

Note that most of the questions can also be answered by using other strategies. For example, since the visualization is a graphical representation of the source code, the questions that are easy to answer using the visualization may be answered by reading the source code as well. We however did design 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. If we did not have these questions, the questionnaire may be answered by looking at a static representation of the visualization and the source code only, where it does not matter if the system is a live programming environment or not.

There may be a bias where, by grouping questions according to the solution strategy, subjects may notice the common solution strategy and stick to using that strategy. To reduce this possible bias, we randomized the order of questions in the questionnaire.

When evaluating the answers, we simply counted how many right answers the subject had. For each right answer we add one point and a wrong answer does not add any point. There are some questions of the questionnaire where subjects had to justify their answers. Whenever the justification of an answer is correct, but they still mark a wrong answer, we add half a point.

3.5 Pilot Studies

As part of the design of the experiment we first performed two small pilot studies.

The first pilot was performed on two subjects. As a result, we improved the original tasks since they were considered too easy by the test subjects. Moreover, this pilot revealed the importance to have obfuscated names in the programs. After we improved the tasks and the questionnaire we invited 2 other subjects to participate in a second pilot. This instance allowed us to improve some questions of the experiment and made them more clear. We also measured the time and checked the overall duration of the experiment.

The pilot of the experiment also gave the necessary practice and confidence to the supervisor of the experiment. He learned how to carry out the experiment and to behave as an impartial being, only answering technical questions without preferring one system over the other.

4. Results

4.1 Participant Profile

We have 8 test subjects, all students from two careers at the engineering faculty of the University of Chile. All subjects are at least in their fourth year, have an interest in robotics and some have taken elective courses in the area of robotics. Six students are undergraduate students, 2 of them are graduate students. There are 5 students from Computer Engineering and 3 students from Electrical Engineering.

2 of the subjects state that they have a Beginner level of SMACH, 3 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. 2 of the subjects are female, one is from Computer Engineering and one is from Electrical Engineering.

4.2 Quantitative Results

We present the raw data of the experiment in Table 1. The LRP and SMACH columns are the score obtained on the questionnaire for the tasks done in LRP and SMACH respectively. The table then presents the time to complete the tasks done by LRP and SMACH. The last column shows the type of work session done by the subject (as explained in Section 3.3).

To analyze these results, we consider four data combinations: score between Task A and Task B, time between Tasks A and Task B, score between SMACH and LRP, and time between SMACH and LRP. We first 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 Man-Whitney test, which is non-parametric and therefore able to cope with non-normal data sets. None of the four data combinations however indicates a significant difference, the results are not even close as the smallest P value is 0.4.

There is no significant difference in score nor time to complete the tasks using LRP and SMACH. As a consequence, the observed difference are due to the experiment setting, and cannot be related to the treatment (SMACH or LRP). Moreover, the average and median of each of the four

Part.	LRP Correctness	SMACH Correctness	LRP Time (min)	SMACH Time (min)	Work Session
1	16	16	100	76	B in LRP, A in SMACH
2	14	13	49	96	B in SMACH, A in LRP
3	16,5	15	47	33	A in LRP, B in SMACH
4	15,5	16	48	54	A in SMACH, B in LRP
5	16	15	55	60	B in SMACH, A in LRP
6	10	12	47	34	A in LRP, B in SMACH
7	15	10	45	55	A in SMACH, B in LRP
8	14	15	53	50	B in LRP, A in SMACH

Table 1. Results of the controlled experiment

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.

Lastly, we notice how similar the tasks are in the controlled experiment. The time and the number of right answers are similar. This shows that the two tasks are comparable, reducing a bias where a task could be easier or harder than the other.

4.3 Qualitative Results

The questionnaire answered by the subjects after finishing the experiment gives us qualitative information about how the subjects feel with the systems. Most of the subjects did not feel any time pressure, with 5 subjects answering that they did not feel any pressure at all. Only 1 subject felt pressure about time, however the results of this subject were similar to the other subjects.

An important qualitative result is that every subject in the experiment thinks that is easy to understand programs written in LRP. In contrast, with SMACH there are mixed opinions: 5 subjects think that is easy to understand programs using this platform and 3 subjects think that it is hard. Moreover, the subjects also think that LRP is better that SMACH with an average result of 4,12. 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 for SMACH.

Some positive observations of the subjects (translated from Spanish) are: “In LRP there are less concepts, so it is easier to understand”, “LRP was easier to understand”, “In LRP the code was easier to undersand”. 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”.

4.4 Conclusions

In this experiment we measured how easy it is to understand complete programs for the behavior layer of robots using 2 different platforms: LRP and SMACH. To do this, the subjects of the experiment studied 2 different programs, one in LRP and one in SMACH. To measure the understanding of the programs we gave the subjects a questionnaire to answer.

The quantitative results of the experiment show us that 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 of the experiment thought that it is easier to understand programs written in LRP than in SMACH. They also stated that they prefer to use LRP over SMACH.

In summary, while the users’ opinions about the different systems confirm the tenet of live programming yielding a better developer experience, the qualitative data show that in this experiment the developers’ performance is actually unchanged.

5. Threats to Validity

As in any experiment, there are several threats to validity in this work. We present the most important ones here.

Inexperience The inexperience of the researcher supervising the experiments may be a threat for the experiment itself. To minimize this, the researcher had several opportunities in the pilot to test not only the treatment of the experiment, but also to test how he interacts with the subjects.

Time Extension The experiment itself has a duration of approximately 4 hours. In this time the subject may get tired, leading to worse results in the second task. While the subject were performing the tasks we were constantly supervising to check if they were performing worse over time, which was not the case. Moreover, we provided them with beverages and food (candies), to lessen fatigue. At the end of the experiment we asked how the subject felt, if he/she was tired or not. All subjects stated that they did not feel tired at the end of the long experiment, even more, they stated that they found the experiment fun.

Type of Participants All participants in the experiment are students studying a topic related to robotics. The use of students may lead to a bias in different types of computer science experiments, but this is however not so in robotics. 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 improved by calling subjects of other areas. We claim that, even while the experiment can be improved, we believe the results of using students is a good measure of developers in the robotics world.

Previous Knowledge of Participants We had multiple participants with different backgrounds. While we invited people that are involved with robotics they may not have had the necessary knowledge to complete the experiment, especially given that one of the systems (LRP) is quite new. We however made sure that every participant has the required level of base knowledge by first performing the warm-up phase.

Effect size An effect size is a quantitative value associated to a statistical measurement. The higher the number of subjects considered, the stronger the correlation effect is between two or more variables. We have considered 8 subjects in our experiment and no significant difference was identified both in time and score between SMACH and LRP. Although our measurements indicates a very slight difference (*e.g.* the mean score of SMACH exercises is 13.85 and 14.80 for LRP), no conclusion can be drawn. Significantly increasing the number of subjects may lead to a representative difference. However, no difference is perceived with 8 participants, so the alternative hypothesis of our control experiment is not verified.

Definition and Artificially of Tasks The tasks defined for the experiments may benefit one system over the other. To avoid this, when we defined the tasks we made sure to not use specific features of one system that may improve its performance. Moreover, the tasks were defined to use several features of both systems so the program were non-trivial. For example, the tasks use nested machines and several paths per states. Even while the tasks are artificially created, they were designed to solve possible real life problems, such a delivering an object or a message sending robot.

6. Related Work

To the best of our knowledge, there has yet been no extensive report of a study of the productivity advantages of live programming for understanding existing code. 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. 2014) the authors 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 taks were divided amongst the participants. In this study participants were significantly faster using Interstate than JavaScript. 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 efficiency of Interstate in understanding already existing code.

The work of Wilcox *et al.* (Wilcox et al. 1997) revealed that continuous visual feedback in direct manipulation of programs helps in the accuracy of debugging certain tasks. They compare a version of Forms/3 (Burnett et al. 2001), which is live, against another version with immediate feedback removed. In this experiment there are 29 subjects, where half of them work in 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 the bug.

Kramer *et al.* (Kramer et al. 2014) also analyzed code creation, complementing the work of Wilcox *et al.* (Wilcox et al. 1997). In this experiment they compare a live version of JavaScript with a non-live version of JavaScript, both versions using Brackets, an IDE for 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. This experiment uses a between-groups design, *i.e.* every group works with a different treatment. The authors noticed that while live programming did not speed up the time to complete a task, it did significantly decrease the time 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 the above small studies on whether live programming improves the efficiency in development, there are three studies about how developers behave in a live programming environment (Wilcox et al. 1997)(Kramer et al. 2014)(Hancock 2003). 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.

Lastly, Hundhausen and Brown (Hundhausen and Brown 2007) investigated the impact of continuous feedback on novice programmers. To do this each subject was exposed

to three different system: 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 does not only implies continuous feedback, but this feedback needs to be meaningful (Hancock 2003), and that is one of the conclusions of this work: "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".

7. Conclusion and Future Work

In this paper we reported on a controlled experiment using within-subjects and repeated measures to gauge whether code written in the live programming language LRP is easier to understand than code using the SMACH conventional language and toolkit for robotic development. To the best of our knowledge, this is the first in-depth study of code comprehension of existing code in a live programming system.

We implemented two different robotics tasks in each system, which were run under simulation. Work sessions consisted of a pre questionnaire, the two tasks and a post questionnaire. Each task contained a warmup and evaluation phase. The warmup phases provided a description of the system that is used in the evaluation phase. In the evaluation phase the subjects answered a questionnaire with 17 code comprehension questions. Work sessions lasted approximately four hours, with a 15 minutes break between each task.

We performed the study with 8 subjects, all of them engineering students with six of them in their fourth year of undergrad or later and two graduate students. The quantitative results of the experiment show us that there is no difference between using LRP or SMACH to understand complete programs for robotic behaviors, both considering correctness and time taken. Moreover, the average and median of the data indicates that there is no tendency that could be increased by including more participants. As a result, even though the number of participants is low, we can say that there is no significant difference in score nor time to complete the tasks using LRP and SMACH. Considering the qualitative results, every subject in the experiment states that is easy to understand programs written in LRP whereas with SMACH 3 subjects find it hard. Also, the subjects think that LRP is better than SMACH and would prefer using it over SMACH.

These results expose an apparent contradiction: the users' opinions about the different systems confirms the tenet that live programming yields a better developer experience but the hard data shows that this does not result in more effective or efficient code understanding. Instead, the quantitative data show that the developers' performance is actually unchanged when performing code comprehension of existing code. This apparent contradiction is remarkable and merits further study.

Our future work hence consists of further studying the performance of developers with LRP versus SMACH. Our next experiment will treat code development, and subsequent experiments will focus on the different features of LRP versus SMACH to establish what their effect is on developer performance and experience.

References

M. Burnett, R. Walpole Djang, J. Reichwein, H. Gottfried, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11:155–206, 3 2001. ISSN 1469-7653.

M. Campusano and J. Fabry. Live robot programming: The language, its implementation, and robot API independence. *Science of Computer Programming*, 2016. To appear.

J. Fabry and M. Campusano. Live robot programming. In A. Bazzan and K. Pichara, editors, *Advances in Artificial Intelligence – IBERAMIA 2014*, number 8864 in LNCS, pages 445–456. Springer-Verlag, 2014. doi: http://dx.doi.org/10.1007/978-3-319-12027-0_36.

A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1980.

C. M. Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.

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

J.-P. Kramer, J. Kurz, T. Karrer, and J. Borchers. How live coding affects developers' coding behavior. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 5–8. IEEE, 2014.

S. Oney, B. Myers, and 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*, pages 263–272. ACM, 2014.

M. Quigley, K. Conley, B. 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*, volume 3, page 5, 2009.

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

E. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and 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*, pages 258–265. ACM, 1997.