An Empirical Study on Program Comprehension with Reactive Programming

Guido Salvaneschi, Sven Amann, Sebastian Proksch, Mira Mezini
Software Technology Group
Technische Universität Darmstadt
lastname@informatik.tu-darmstadt.de

ABSTRACT
Starting from the first investigations with strictly functional languages, reactive programming has been proposed as THE programming paradigm for reactive applications. The advantages of designs based on this style over designs based on the Observer design pattern have been studied for a long time. Over the years, researchers have enriched reactive languages with more powerful abstractions, embedded these abstractions into mainstream languages – including object-oriented languages – and applied reactive programming to several domains, like GUIs, animations, Web applications, robotics, and sensor networks.

However, an important assumption behind this line of research – that, beside other advantages, reactive programming makes a wide class of otherwise cumbersome applications more comprehensible – has never been evaluated. In this paper, we present the design and the results of the first empirical study that evaluates the effect of reactive programming on comprehensibility compared to the traditional object-oriented style with the Observer design pattern. Results confirm the conjecture that comprehensibility is enhanced by reactive programming. In the experiment, the reactive programming group significantly outperforms the other group.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features

Keywords
Reactive Applications, Reactive Programming

1. INTRODUCTION

Reactive applications are a wide class of software that needs to respond to internal or external stimuli with a proper action. Examples of such applications include user-interactive software, like GUIs and Web applications, graphical animations, data acquisition from sensors and distributed event-based systems.

Over the last few years, reactive programming (RP) has gained the attention of researchers and practitioners for the potential to express otherwise complex reactive behavior in intuitive and declarative way. RP has been firstly introduced in Haskell. Influenced by these approaches, implementations of RP have been proposed in several widespread languages, including Scheme [3], Javascript [29] and Scala[40, 27]. Recently, concepts inspired by RP have been applied to production frameworks like Microsoft Reactive Extensions (Rx) [25], which received great attention after the success story of the Netflix streaming media provider. This growing interest around RP is also witnessed by the success of the Coursera online class “Principles of Reactive Programming”, in winter semester 2013-14. Finally, a lot of attention in the front-end developers community is revealed by the increasing number of libraries, inspired by the Flapjax reactive language [29], that implement RP principles, among the others React.js, Bacon.js, Knockout, Meteor and Reactive.coffee.

The relevance of RP comes from the well-known complexity of reactive applications, which are hard to develop and understand because of the mixed combination of data and control flow. The Observer design pattern [14] is widely used for such applications. It has the advantage of decoupling observers from observables. But, when it comes to program readability, it does not make things easier, because of dynamic registration, side effects in callbacks, and inversion of control.

In contrast, RP supports a design based in data flows and time-changing values: the programmer states which relations should be enforced among the variables that compose a reactive program and the RP runtime takes care of performing all the required updates. Dependencies are defined explicitly instead of being hidden in the control flow, combination can be guided by types opposite to callbacks that return void, contrarily to the Observer pattern control is not inverted and less boilerplate is required since collecting dependencies and performing the updates is automatized by the framework. Based on these arguments, it has indeed been argued that RP greatly improves over the traditional Observer pattern used in OO programming both (i) from the software design perspective as well as (ii) from the perspective of facilitating the comprehensibility of the software [26, 3, 29, 2].

Yet, not enough empirical evidence has been ever provided in favor of the claimed advantages of RP. Preliminary empirical results seem to confirm the claimed design benefits (e.g., more comprehensibility) of RP [40]. However, even preliminary evidence is missing regarding the claim that RP enhances comprehensibility. Despite the intuition about its potential, the reactive style is not obviously more comprehensible than the Observer design pattern. For example, in the Flapjax paper [29], a Javascript application based on Observer is compared against a functionally equivalent RP version. The authors argue that the RP version is much easier to comprehend. However, the reader is warned that: “Obviously, the Flapjax
code may not appear 'easier' to a first-time reader”. Doubt-
ing, at this point, is legitimate: does RP really make reactive applications easier to read? Also, it is unclear how much expertise is required to find a RP program "easier" – if ever.

To fill the gap, this paper provides the first empirical evaluation of the impact of RP on program comprehension compared to the traditional technique based on the Observer design pattern. The experiment involves 38 subjects that where divided into a RP group and an OO group. They were shown a reactive application and their understanding of the reactive functionalities was measured. The experiment considers several aspects including (i) correctness of comprehension, (ii) required time and (iii) programming skills needed for correct comprehension. To the best of our knowledge, such a study has never been attempted before. All experimental data and the artifacts developed for this work are available for download\(^1\), including the tasks that compose the experiment, the raw results and the complete statistical analysis not included in this paper for space reasons.

The paper is organized as follows: Section 2 motivates the work. Section 3 introduces the design of the study. Section 4 describes the results. Section 5 discusses threats to validity. Section 6 presents related work. Finally, Section 7 outlines open research challenges and concludes the paper.

2. MOTIVATION

In this section, we introduce reactive programming and discuss the weaknesses of the Observer pattern that can potentially reduce program understandability.

Reactive Programming in a Nutshell.

Reactive programming supports first-class constraints among program values. These constraints are automatically enforced by the runtime. In the rest, we adopt the terminology of REScala [40] and Scala.react [27]. Constraints are expressed as signals, a language concept for expressing functional dependencies among values in a declarative way. A signal can depend on variables without further dependencies (i.e., vars) or on other signals. When any of the dependency sources changes, the expression defining the reactive value is automatically recomputed by the language runtime to keep the reactive value up-to-date.

The general form of a signal \(s\) is \(\text{Signal}(\text{expr})\), where \(\text{expr}\) is a standard expression. When \(\text{expr}\) is evaluated, all \(\text{Signal}\) and \(\text{Var}\) values it refers to are registered as dependents of \(s\); any subsequent change of them triggers a reevaluation of \(s\).

Consider the example in Figure 1a. The code snippet defines two vars \(a\) and \(b\). When \(a\) is declared (Line 3), the runtime collects the values it depends on (i.e., \(a\) and \(b\)). When \(a\) is updated (Line 6), all signals that (even indirectly) depend on it – \(s\) in this case – are automatically updated.

As the reader has probably already noticed, the syntax of the example is a bit cluttered because of the implementation of RP as an embedded Scala DSL. Assigning a var requires \(\text{}\langle\text{}\rangle\) which is translated into a call to the \(\text{app1y}\) method on the var object. Similarly, vars and signals must appear with the method call notation \(\text{}\langle\text{}\rangle\) in signal expressions. More details can be found in [40, 27].

Reactivity the Old School: The Observer Pattern.

In OO languages, reactive applications are usually developed using the Observer design pattern. This solution has gained wide popularity because it decouples observers from observables, i.e., observables do not know (hold a static reference to) observers in

\[^1\]http://goo.gl/8Qn4cx

```scala
1 val a = Var(1)
2 val b = Var(2)
3 val s = Signal(a() + b())
4 println(s.getVal()) // 3
5 a()=4
6 println(s.getVal()) // 6
```

Figure 1: Functional dependencies using signals in RP (a) and the Observer design pattern in OO programming (b).

advance. Instead, observers register themselves to the observable they are interested in at runtime [14]. Detractors argue that programs are hard to reason about. Below, we summarize the main points that support this argumentation. For convenience, we refer to the example in Figure 1b, which implements the same functionality as Figure 1a but with the Observer design pattern.

(i) The natural dependency among program entities is inverted. Conceptually, change flows from observables to observers, but in code, observers call the observable to register. For example, \(c\) is registered calling \(\text{addObserver}\) on a (Line 5).

(ii) Programmers need to inspect a lot of code to figure out the reactive behavior because functional dependencies are implicit. To define the dependency from \(a\) to \(b\), programmers register a handler to \(a\) that updates \(c\). When readers encounter \(c\) in the code for the first time, there is no sign that the value permanently depends on another one, since the update is performed by a side effect in the handler potentially anywhere in the program.

(iii) Code is cluttered. Reactive applications are more verbose and the application logic is hidden behind the machinery required by the Observer design pattern.

Executive Summary: Call to Arms.

It is, however, not clear, what the impact of the previous issues is. For example, (i) contributes to make OO applications too complex to read at first sight, but, with experience, programmers are likely to get used to inversion of control. For (ii), the handler still expresses the functional dependency, even if indirectly. Concerning (iii), there is no evidence that the Observer design pattern clutters programs up to the point that they are significantly harder to read than with an alternative design. In summary, the claims that RP addresses the aforementioned issues and improves program comprehension should be evaluated empirically.

3. STUDY DEFINITION

The claimed advantages of RP include increased composability, abstraction over state, automatic memory management, enforcement of consistency guarantees during change propagation and ease of comprehension [40, 26, 3, 29, 2]. In this work, we limit the scope to program comprehension. We argue that this aspect is crucial because programs are written once but read many times: “Programs must be written for people to read, and only incidentally for machines to execute” [1]. We also consider a single alternative to RP,
i.e., OO programming and the Observer design patterns, because it is the most common solution for reactive applications.

Object of the Experiment.

The experiment focuses on 10 reactive programs. Each program is implemented in two versions. The RP version is based on reactive programming, i.e., signal and, when needed, events. The OO version adopts the Observer pattern to implement reactivity.

The applications we developed belong to three categories. The first category consists of synthetic applications (applications 1-4) that define functional dependencies among values and propagate changes when certain values are updated. These applications look similar to Figure 1 except that the functional dependencies are more complex. The second category consists of graphical animations (applications 5-7), where shapes are displayed on a canvas and move according to regular patterns. Graphical animations is a traditional domain for reactive programming [9, 8]. The last category consists of interactive applications (8-10) that require the user, e.g., to click buttons or drag the mouse over a shape. These functionalities are common in GUIs, another traditional domain for reactive programming [29, 3].

An example of the applications we used is the Squares application in Figure 2a (RP version) and Figure 2b (OO version). The example belongs to the animations category. Code is reduced for presentation purposes. The Squares application draws two moving squares on a canvas. To give an intuition of the execution, a screenshot is provided in Figure 3 (note that subjects did not have access to it). The upper and the lower square move horizontally from left to right at constant speed – respectively, at increasing speed.

In the RP version (Figure 2a) the position signal in Line 5 models a time-changing value of type Point. The x coordinate of the point depends on the time signal, the y coordinate is fixed. Every time the time signal changes (defined elsewhere), a new point is generated and assigned to the content of the position signal. The position signal for the second square (Line 13) works similarly, except that the x coordinate of the point (Line 14) depends on both time and a speed signal v defined in Line 10. Line 19 triggers a repaint in the asynchronous Swing events loop every time either position of square1 or the position of square2 changes. Lines 24-39 setup the canvas and display the initial squares.

In the OO version (Figure 2b) the same functionalities are implemented using the Observer pattern. Line 6 register a handler...
to the observable time (defined elsewhere). The handler updates the position variable in Line 5. The position variable is observable and the handler in Lines 27-28 repaints the view every time position changes. The definition of square2 follows the same principles; the GUI setup in Lines 29-48 is unchanged from the RP case.

As the reader may have noticed, some functionalities, like time, are defined elsewhere and accessed via import (not shown). This is also the case for the reactivity machinery (signals and observers). This choice helps to keep the application code short and is consistent across the RP and the OO version.

Methodology.

The experiment is composed of 10 tasks. In each task, a subject is shown a reactive application and asked a question about the application behavior. Crucially, questions and alternative choices are formulated in a way that finding the correct answer requires to understand the (whole) reactive logic of the application. An example taken from the synthetic applications group is to ask the sequence formulated in a way that finding the correct answer requires to understand the variables that depend on other values in the program once such values are updated. Answering the question requires to inspect the application to detect functional dependencies among values, i.e., which values are affected by a change and in which order. An example from the interactive group comprises a quick description of the application in the question (a canvas with a box drawn on it) and asks which combination of actions from the user produces a change of the color of the GUI. The correct answer is “crossing two borders of the box while dragging the mouse”. An example for the animations is shown in Figure 4 and refers to Figure 2.

To practically run the experiment, we developed WebCompr, a Web application for experiments on program comprehension that allows users to complete the tasks via a Web browser. Existing tools for controlled experiments are aware of are not Web-based (e.g., Biscuit [32] and Purity [15]) which considerably increases the complexity of running the experiment. We relinquished using existing Web platforms for in-browser homework and exams (e.g., WebLab\(^2\)) because they lack fine-grained time management, such as setting an upper bound to each single task and recording the task completion time – which are important for our experiment (more on this later). WebCompr is highly customizable, collects events from the user in a generic way and supports different kinds of tasks, including some not used here, for example, blind tasks where subjects can inspect an application for a limited amount of time and are asked to complete a task afterward, without access to the code anymore. WebCompr is available to the community to run experiments similar to ours.

We use tasks as described above as a research instrument, because they provide objective results and scale well (most controlled studies hardly include more than 15 participants [12, 22] – we have 38 subjects). Yet, the experiment requires a controlled environment with on-site execution and staff supervision to make sure that developers perform the task without external help. Also, we wanted to control the training of the subjects, which prevented us from making the experiment publicly available on the Internet and ask for volunteers.

We briefly discuss the alternatives we evaluated. In the think loudly approach subjects comment the actions they are performing [12, 22]. A subsequent interview can clarify the motivations behind each action [38]. This approach, however, does not allow to collect objective measures and apply statistical tests. Other studies measure software comprehension by artificially introducing a bug in an application. Then, they measure the time subjects need to fix a bug or perform a modification task [22]. However, bug fixing requires not only to understand the application, but also to write code that solves the problem. Thus, measurements would include code-writing skills, not only comprehension (i.e., code-writing skills is an additional factor of the experiment). This approach is suitable, if the factor is balanced for the groups. In such a case, it doesn’t influence the main factor we want to observe. In our experiment however, the programming skills of the subjects in the RP and in the OO style may be very diverse and there would be no way to separate the influence of this factor from the main factor we want to observe – program comprehension.

We decided to measure both time and correctness of the results. Subjects were encouraged to provide an answer as quickly as possible, but it was also clarified that time becomes relevant only in case the answer is correct. The reason for this design choice is to simulate a realistic coding session in which developers inspect a large project and spontaneously force themselves to spend on each portion class only a limited amount of time to keep the analysis of the whole software feasible.

For practical reasons, we needed to define an upper bound to the time subjects can spend on each task. Also, it has been shown that, without constraints, subjects may spend the entire time of the experiment on a single task [36]. We fixed the available time to 5 minutes for synthetic applications and to 10 minutes for the animations and interactive applications, because the latter are slightly longer and require to inspect more code. In practice, our estimations turned out to be rather conservative. None of the subjects required the full amount of time and most subjects provided that answer way earlier. In any case, to avoid that attention decreases too much, the experiment was designed to require no more than 2 hours (preliminary tasks + experiment tasks).

Controlled experiments can be conducted with between-subjects design or within-subject design. In between-subjects designs two versions of the same application are proposed to different subjects. In within-subjects designs each subject is given both versions [20]. In most empirical studies in software engineering, within-subject design is usually preferred to balance the effect of the individual skills factor, i.e., to reduce the variability due to heterogeneous skills among the subjects. On the other hand, within-subject designs introduce learning effects, because subjects can apply the knowledge gained when solving a task with a factor to the solution of the same task with the other factor – a problem solved by between-subjects designs. It has been shown [16, 28] that, if the learning effect is small enough, proper experiment design still allows a within-subject approach. In our case, however, this was no option. The learning effect is likely to dominate the effect we want to measure, because it is extremely easy for subjects to remember previous findings in the experiment – a known challenge in controlled experiments on program comprehension [36]. Therefore, we chose to design the experiment as a between-subjects.
The following application draws two squares. Which of the following sentences is true?

- a - The squares are moving at the same speed
- b - The first square is moving at constant speed, the second at increasing speed
- c - The squares have a fixed position
- d - The first square is fixed, the second is moving

Figure 4: Example question.

Study Context.

The subjects of our study are students from a Software Engineering course, held in the 4th year of study in Computer Science. Subjects have similar academic background. They have been exposed to Scala programming for at least the semester of the course. All subjects learned Java since CS101 in their first semester and used Java as their primary language in the rest of their studies. They learned the Observer pattern since their Java first year course on programming. Subjects were taught RP for an amount of two lectures (1.5h+1.5h) and were assigned the homework (8h+8h estimated) that require to redevelop a reactive application from a given specification.

We used WebCompr also to collect information about the subjects, such as their programming experience, and required them to self-evaluate as programmers. Since self-evaluation cannot be considered reliable [21], subjects were also given 18 preliminary tasks to solve as a measure of their programming knowledge. To test a variety of different topics we necessarily kept each task very short – the maximum amount of time available is a minute. Questions include concepts of OO programming, like inheritance and polymorphism, functional programming, e.g., high-order functions and pattern matching, and also RP, including events and signals. A secondary goal of these tasks was to train students to correctly use the Web application we used for the experiment.

Overall, the questions we asked were quite advanced and not necessarily covered by the course. Subjects were given multiple answers and a Don’t know option to avoid guessing. The assumption behind this approach is that good programmers are likely to master advanced features of the language. For example, a subject that knows the behavior of the zip function on lists is likely to be a better programmer than a subject that has never heard about it.

Research Questions and Hypotheses.

We want to investigate the impact of RP on the comprehension of reactive applications. Our research questions can be formulated as follows. The fist research question concerns correctness of program comprehension in the RP and in the OO style:

RQ1: Does reactive programming increase correctness of program comprehension?

The second research question investigates the role of time (besides correctness) in comprehending RP programs on OO programs:

RQ2: Is comprehending reactive applications in the RP style more or less time-consuming than in the OO style?

The third question concerns the relation between programming skills and the level of comprehension of RP programs compared to OO programs. The impact of programming expertise on the use or RP is important to know whether RP is only useful for advanced programmers or exhibits significant benefits also for beginners.

RQ3: Does comprehending RP programs require more advanced programming skills than the OO style?

While previous research has speculated on RP overperforming OO programming w.r.t. program comprehensibility, we take a neutral approach. Consistently, in Section 4 we define 2-tailed statistical tests to analyze our results.

4. STUDY RESULTS

In this section, we present the main contribution of our work, the analysis of the statistical difference between the subjects in the RP group and the subjects in the OO group. For the analysis we used the SPSS statistical tool.

As a preliminary step, the tasks on programming skills allow us to validate the hypothesis of equal distribution among the RP group and the OO group. This assumption is needed to make sure that the groups are not unbalanced for what programming skills concern. A first inspection of the data shows similarity between the groups (Figure 5). We performed a Mann-Whitney 2-tailed U test on the subjects in the RP group and in the OO group. The plots seem to support the claim that equality of the groups cannot be rejected (p=0.17).

4.1 Correctness

First, we analyze the data to provide an answer for RQ1. Correctness analysis concerns the amount of tasks for which the subject correctly understood the behavior of the application and provided the right answer. As the measure of correctness, we consider the score (i.e., the cumulative number of correct answers) provided by each student in all tasks.

We initially provide an overview of the results with descriptive statistic. The mean score for the OO group and the RP group are \( \mu_{\text{score,OO}} = 6.65 \) respectively \( \mu_{\text{score,RP}} = 8.17 \). Figure 6 shows the cumulative scores obtained by the subjects in the RP group and in the OO group. The plots seem to support the claim that the RP group obtains a better result than the OO group. However, to conclude that the difference is significant, it is necessary to apply an appropriate statistical test.

To check if the difference between the scores of the RP group and the OO group is significant, we formulate the following hypothesis to be tested:

The zip function takes two lists and returns a list of pairs. Given the input lists \( [l_i, i \in (0..n)] \) and \( [r_i, i \in (0..n)] \), zip returns the list \( [(l_i, r_i), i \in (0..n)] \).
In contrast to the correctness case, where binary data require a cumulative score over all the tasks, for time data are continuous and we can analyze each task separately. Similar to the previous case, the underlying distribution is unknown and we perform a non-parametric Mann-Whitney U test. The results are in Figure 9. For exercises 1-2-3-4-7, \( H_0 \) can be rejected with high significance \((p<0.05)\). In all those cases, the rank sum indicates that OO times are higher than RP times (Figure 9, column “Ranks sum”), i.e., the RP group is faster. In exercises 5-6-8-9-10, \( H_0 \) cannot be rejected \((p>0.05)\).

One may argue that the last result measures the time it takes a programmer to give the answer she believes is correct. We further inspected our results to restrict the analysis to only the times that the subjects needed to provide the correct answer. The results of the Mann-Whitney U test are in Figure 10. For exercises 1-2-3-7-10, the difference is significant \((p<0.05)\) which rejects \( H_0 \). The rank sum indicates that OO dominates RP. In the other exercises \( H_0 \) cannot be rejected. These observations confirm that the previous conclusion still holds even when the analysis considers only the correct answers. In summary, we answer RQ2 concluding that comprehending programs in the RP style does not require more time than comprehending their OO equivalent.

### 4.3 Programming Skills

In this section, we inspect the experimental data to answer RQ3 – were more advanced programming skills required for comprehending RP applications. To this end, we correlate the programming skills we measured for each subject with the correctness results in the experiment. Figure 11 shows a scatter plot of the programming skills and the cumulative score in the tasks for both the RP group and the OO group (there are less data points than subjects because some data points are identical for different subjects).

The figure suggests that in the OO group lower-skilled subjects perform poorly in the tasks. Subjects that performs better are also advanced programmers. In the RP group, instead, subjects do not need advanced programming skills to reach high scores. In other terms, the figure suggests absence of correlation for the RP group and a positive correlation for the OO group, which we further investigate in the rest. At this stage of the analysis, the inspection of the plots in Figure 11 shows a significant difference between the RP group and the OO group.

We use a statistical test to estimate the significance of the correlation of subjects’ score and skills. We define the following hypotheses:

\( H_{0(RP)} \): There is no correlation between the score of the subjects in the RP group in the experiment and the measured programming skills.

\( H_{0(OO)} \): There is no correlation between the score of the subjects in the OO group in the experiment and the measured programming skills.

The skills results cannot be safely assumed normally distributed. We use the Kendall’s \( \tau \) test and Spearman’s rank correlation \( \rho \) test. Even if this result is unlikely, we do not exclude negative correlation and perform 2-tailed tests. The results are presented in Figure 12, where we show the correlation coefficient \( \tau \) and the \( p \)-value \( p \) for each case. For the RP group, we detected no significant correlation, i.e., \( H_0 \) cannot be rejected. Surprisingly, in this case, the correlation coefficient is even negative. For the OO group, instead, both tests are statistically significant \((p=0.039, p=0.045)\) and \( H_0 \) can be rejected. We conclude that in the OO group there is ev-
Figure 8: Box plots for time in RP and OO Tasks.

<table>
<thead>
<tr>
<th>Group</th>
<th>N</th>
<th>Rank avg</th>
<th>Rank sum</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>RP 18</td>
<td>13.17</td>
<td>237</td>
<td>0.001</td>
</tr>
<tr>
<td>OO 20</td>
<td>25.2</td>
<td>504</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 2</td>
<td>RP 18</td>
<td>11.39</td>
<td>205</td>
<td>0.023</td>
</tr>
<tr>
<td>OO 20</td>
<td>26.8</td>
<td>536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 3</td>
<td>RP 18</td>
<td>15.19</td>
<td>273.5</td>
<td>0.003</td>
</tr>
<tr>
<td>OO 20</td>
<td>23.38</td>
<td>467.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 4</td>
<td>RP 18</td>
<td>14.03</td>
<td>252.5</td>
<td>0.011</td>
</tr>
<tr>
<td>OO 20</td>
<td>24.43</td>
<td>488.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 5</td>
<td>RP 18</td>
<td>18.14</td>
<td>326.5</td>
<td>0.483</td>
</tr>
<tr>
<td>OO 20</td>
<td>20.73</td>
<td>414.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 6</td>
<td>RP 18</td>
<td>18.11</td>
<td>326</td>
<td>0.474</td>
</tr>
<tr>
<td>OO 20</td>
<td>20.75</td>
<td>415</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 7</td>
<td>RP 18</td>
<td>14.75</td>
<td>265.5</td>
<td>0.745</td>
</tr>
<tr>
<td>OO 20</td>
<td>23.78</td>
<td>475.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 8</td>
<td>RP 18</td>
<td>13.64</td>
<td>245.5</td>
<td>0.028</td>
</tr>
<tr>
<td>OO 20</td>
<td>16.64</td>
<td>335.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 9</td>
<td>RP 18</td>
<td>18.12</td>
<td>367.5</td>
<td>0.638</td>
</tr>
<tr>
<td>OO 20</td>
<td>18.68</td>
<td>373.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 10</td>
<td>RP 18</td>
<td>19.11</td>
<td>344</td>
<td>0.845</td>
</tr>
<tr>
<td>OO 20</td>
<td>19.85</td>
<td>397</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9: Mann-Whitney U test for Answer Times.

Figure 10: Mann-Whitney U test for Correct Answers Times.

Figure 11: Scatter plots of Score and programming Skills.

Figure 8: Box plots for time in RP and OO Tasks.

Figure 9: Mann-Whitney U test for Answer Times.

Figure 10: Mann-Whitney U test for Correct Answers Times.

Figure 11: Scatter plots of Score and programming Skills.

Figure 8: Box plots for time in RP and OO Tasks.

Figure 9: Mann-Whitney U test for Answer Times.

Figure 10: Mann-Whitney U test for Correct Answers Times.

Figure 11: Scatter plots of Score and programming Skills.

The analysis provides an answer for RQ3. We conclude that, in contrast to OO where score results are correlated to programming skills, with RP (advanced) programming skills are not needed to understand reactive applications. This result suggests that RP lowers the entrance barrier required to understand reactive applications.

5. DISCUSSION

5.1 Threats to Validity

In this section, we discuss factors that menace the validity of our result and which countermeasures we adopted to reduce this risk.

Construct Validity.

Threats to construct validity refer to the extent to which the experiment does actually measure what the theory says it does.

Our approach to measure program understanding requires careful formulation of questions and candidate answers. Questions that are too specific may not require a significant understanding of the program. For example, the correct answer "3 clicks" for the question “How many clicks are needed to activate functionality X” can be found (with low confidence, admittedly) just by spotting in the program code a line like if(numberOfClicks == 3) [...] . We tackled this problem in two ways. First, we for-
mulated questions and candidate answers in a way that requires a broad understanding of the reactive behavior of the program. The questions we formulated are basically equivalent to “what does this reactive program do?”, as discussed in Section 3 and shown in Figure 3. We made sure that subjects cannot simply spot the correct answer by “pattern matching” over the code. Second, we performed renamings to change too meaningful names into more neutral ones. For example, in the application in Figure 2a the variables position in Line 5 and position in Line 13 were originally named constantSpeedPosition and increasingSpeedPosition, which immediately provides an answer to the question in Figure 4.

Another issue concerns the use of a Web-based WebCompr application to complete the tasks. Such a platform can be not immediately intuitive for some subjects, which may potentially affect the results, especially timing. To mitigate this effect, we presented the interface of WebCompr in detail before the experiment started and showed example screenshots to give the subjects a clear feeling of what to expect. Also, the experiment starts with warm up tasks to help subjects getting confidence with the platform.

**Internal Validity.**

Threats to internal validity relate to factors – other than independent variables under control – that can affect dependent variables, i.e., influence the results.

While IDEs play a relevant roles in programmer’s activity, the influence of the IDE on different programming paradigms is unclear and should be subject of a different study. Other studies facing the same problem [11, 16, 18] adopted a minimal programming environment to minimize the effect of the IDE. This problem is especially relevant for program comprehension, since advanced search tools and outlines of the program structure can significantly speed up understanding of the application behavior. It must also be considered that there is currently no dedicated tool support for RP which would unfairly disadvantage this paradigm in a comparison that takes the IDE into account. Our approach based on the WebCompr platform has the advantage that the impact of tool support is a removed factor. Text search and syntax highlighting are the only help subjects receive.

**External Validity.**

Threats to external validity relate to what extent our findings are generalizable.

A first issue concerns the training subjects received. The comparison of two programming techniques is definitely influenced by the skills of the participants in each. For example, a previous study on parallel programming in Scala concluded that programmers are no more productive in Scala than in Java, but subjects trained in both Scala and Java multcore programming for the study had 4 years experience with Java and no previous knowledge of Scala. In our case training on RP was minimal (cf. Section 3), yet leads to observable differences in favor of RP.

Another issue concerns the types and size of the applications we adopted in the experiment. The type of applications we selected are representative for typical domains for RP. We argue that these are also representative for a wide class of reactive applications. Also, synthetic applications capture the issues of reactivity in general and are not bound to a specific domain. Concerning size, small-size applications are necessary to keep the experiment feasible. However, for what reactivity concerns, we tried to reflect the structure of bigger applications. For example, in Figure 2 the signal in line Line 10 could be removed collapsing its signal expression with the one in Line 13. Similar considerations apply for the OO counterpart. The correctness of such a design is disputable, yet, it is functionally equivalent to the presented solution. More importantly, however, intermediate observables are likely to appear in larger applications because programmers are more likely to use these intermediate values in multiple places and therefore introduce them in the first place. To make sure that both the RP version and the OO version of the applications are representative of the respective style, we asked the members of our research group to review our code and modified it according to the feedback we obtained.

The subjects of our experiment are students. Although using students for empirical studies is common practice, this can affect the result of the experiment [7]. Professional developers can have more expertise in a programming technique after applying it on daily bases for years. A final consideration concerns subjects skills. Kleinenschmager and Hanenberg, in a preliminary study [21], presented a negative result on using pretests as a valid criteria for measuring programming skills. However, we use 18 preliminary tasks to increase statistical validity, while in [21] a single task is used to predict the outcome of other 14 tasks which significantly increases our confidence on the methodology we adopt.

### 5.2 Preliminary Interpretation and Outlook

The results of our empirical study show that RP overcomes OO for program comprehension and, hence, should encourage researchers to further explore RP. However, the work presented in this paper does not test the root causes of the difference between RP and OO w.r.t program comprehension. Further systematic investigation of this issue should proceed along two lines.

First, more evidence should be collected on how developers reason about RP programs. Exploratory studies, e.g., using either the think loudly approach or interviews [24, 22] help understanding the causes of the issues developers face. Far form being systematic, we collected informal feedback from the subjects in the experiment. With different wording, subjects reported that, with OO versions, one has to follow the flow of the whole application to infer the reactive behavior. On the other hand, with RP, the flow associated to the reactive behavior is explicit (in practice a signal is defined together with its expression). Based on this feedback and on our experience, we believe that the main effect we observed comes from RP making it easier to reason about data flow i.e., points (i) and (ii) in Section 2. This conjecture is supported by previous findings on program comprehension showing that developers understand programs by building a mental model based on control flow [34] and data and control flow play a fundamental role in the way programmers develop a mental representation of programs [6]. Data flow determines whether a variable depends on another, a fundamental aspect of comprehending reactive applications. Detecting dependencies among variables is also a reachability question because it requires to check whether a change to a variable propagates to the

<table>
<thead>
<tr>
<th></th>
<th>RP</th>
<th>OO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Score</td>
<td>Skills</td>
</tr>
<tr>
<td>Kendall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Score</td>
<td>r 1</td>
<td>1</td>
</tr>
<tr>
<td>p</td>
<td>0</td>
<td>0.597</td>
</tr>
<tr>
<td>Skills</td>
<td>r -0.102</td>
<td>1 *0.358</td>
</tr>
<tr>
<td>p</td>
<td>0.597</td>
<td>0.039</td>
</tr>
<tr>
<td>Spearman</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Score</td>
<td>r 1</td>
<td>0.139</td>
</tr>
<tr>
<td>p</td>
<td>0</td>
<td>0.584</td>
</tr>
<tr>
<td>Skills</td>
<td>r -0.139</td>
<td>1 *0.452</td>
</tr>
<tr>
<td>p</td>
<td>0.584</td>
<td>0.045</td>
</tr>
</tbody>
</table>

Figure 12: Correlation of Score and Skills with Kendall’s τ test and Spearman’s ρ test. Starred correlations are significant.
other. Previous research [23] noticed that reachability questions are extremely common for developers, they are (i) hard to answer and time-consuming and (ii) error prone. These findings also provide an explanation of our results.

Second, further studies are needed to investigate the use and effect of RP for certain domains in a more systematic way. In our case, this point concerns the relation between programming tasks and the effect of RP on program comprehension. In our experiment design, correctness is a binary variable. Hence, it is unlikely that splitting the analysis for correctness provides significant results for each task. For this reason, in Section 4 we analyzed the aggregated data. However, a preliminary comparison of the correctness in the OO group and in the RP group for each task is depicted in Figure 13. Since the hypothesis on the size of expected frequencies for a Chi-squared test is not met, we used the Likelihood Ratio and Fisher’s exact test. The tests are still able to detect a significant advantage of RP for tasks 3, 5 and 6. The difference in the other tasks is not significant. Interestingly, two out of three significant cases belong to animations, which has been the domain for which functional-reactive programming has been originally proposed as a suitable programming paradigm [9].

6. RELATED WORK

We organize related work as follows. First, we outline recent research on RP. Second, we provide an overview of existing empirical studies on program comprehension. Last, we enlarge the scope to other studies and controlled experiments on programming techniques. We are not aware of any controlled experiment on RP.

Reactive Programming.

Reactive extensions of existing languages include FrTime [3] (Scheme), FlapJax [29] (Javascript), Scala.react [27] and REScala [40] (Scala). An overview of the available solutions and of the advanced features each language adds to those presented in Section 2 can be found in the survey [2].

Current reactive languages have been influenced by several approaches, often from quite different domains. Functional-reactive programming was proposed in the strictly functional language Haskell and applied to graphic animations [8], robotics [19] and sensor networks [31]. Graphical libraries, like Garnet and Amulet (Lisp) [30] applied concepts of dataflow programming to relieve the user from manually updating dependent values. Other researchers proposed languages where developers can specify constraints. Bidirectional constraints are allowed and in case not all constraints can be satisfied, a priority rank is applied [13]. Current research directions include RP in the distributed setting [39] and integration with the OO paradigm [41].

Studies on Program Comprehension.

Pennington [34] shows that different language designs can influence whether control flow or data flow questions are easier to understand for programmers. Ramalingam and Wiedenbeck [37] organize an empirical study on program comprehension in the OO and in the imperative style. They find that novice developers achieve better program comprehension in the imperative style. However, in contrast to the mental representation of imperative programs, which focuses on program-level knowledge, the mental representation of OO programs focuses on domain-level knowledge. Corritore and Wiedenbeck substantially confirm these results [5].

Other researchers organized controlled experiments to investigate the effect of tools on program comprehension. Quante [36] study the impact of dynamic object graphs on program understanding. Wettel et al. [45] show that the CodeCity 3D software visualization tools significantly increase task correctness and reduce task completion time. Similarly, Cornelissen et al. [4] evaluate the enhancement of program comprehension by visualizing execution traces with tools.

From a methodological perspective, Di Penta et al. discuss the issues of designing studies on program comprehension [7]. Storey [42] reviews the theories that have been formulated for program comprehension and their implications on tool design.

Empirical Studies on Programming Techniques.

Pankratius et al. [33] organize an empirical study to compare the use of Scala and Java to develop multicore software. Contrarily to a common belief, they find that Scala does neither reduce development effort nor debugging effort. Prechelt presents an empirical study that compares seven programming languages along directions that include working time to complete a task and productivity [35]. In [17] researchers investigate variation of development time using Aspect-oriented Programming. Hanenberg organized a series of controlled experiments to evaluate the effect of types in programming languages. Among the other results, these experiments do not find a positive effects of static type systems on development time [16] and show that similar uncertainties hold for the influence of static type systems on the usability of undocumented software [28]. Also, it was found that generic types improve documentation, do not significantly change development time and reduce extensibility [18].

Beside comparing languages or programming paradigms, researchers focused on specific abstractions and API design, including requiring parameters to objects constructors [43], effects of method placement on API learnability [44] and the effect on usability of the Factory desing pattern [10].

7. CONCLUSION

RP is a paradigm that specifically addresses reactive software. The advantages of RP over the traditional OO paradigm have been advocated for some time now, but little evidence has been provided in practice. In this paper, we presented a controlled experiment to evaluate the impact of RP on program comprehension. Results suggest that RP outperforms OO. The RP group provided more correct results, while not requiring more time to complete the tasks. In addition, we found evidence that understanding RP program requires less programming skills than OO programs.

In the future, we will continue working on the evaluation of RP. First, we plan to repeat the experiment with other subjects to further strengthen statistical significance. Second, we want to further investigate which aspects of RP are crucial for improving comprehension extending the analysis to the domain of the applications.

Acknowledgment

This research has been funded by the European Research Council, grant No. 321217 and by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under grant No. 01I12S01V and under grant No. 01IS12054.

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
<th>T10</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.Ratio</td>
<td>0.038</td>
<td>0.527</td>
<td>0.036</td>
<td>0.042</td>
<td>0.008</td>
<td>0.019</td>
<td>0.586</td>
<td>0.253</td>
<td>0.573</td>
<td>0.88</td>
</tr>
<tr>
<td>Fisher</td>
<td>0.067</td>
<td>0.697</td>
<td>0.052</td>
<td>0.093</td>
<td>0.048</td>
<td>0.045</td>
<td>0.719</td>
<td>1</td>
<td>0.709</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 13: Likelihood Ratio and Fisher’s Exact Test.
### Table 8: Descriptive statistics

<table>
<thead>
<tr>
<th>Subject</th>
<th>TP</th>
<th>TO</th>
<th>Correct</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median</td>
<td>73.00</td>
<td>46.00</td>
<td>143.00</td>
<td>46.00</td>
</tr>
<tr>
<td>Average</td>
<td>72.56</td>
<td>58.87</td>
<td>144.36</td>
<td>56.82</td>
</tr>
<tr>
<td>MedIAN</td>
<td>73.00</td>
<td>46.00</td>
<td>143.00</td>
<td>46.00</td>
</tr>
<tr>
<td>Average</td>
<td>72.56</td>
<td>58.87</td>
<td>144.36</td>
<td>56.82</td>
</tr>
</tbody>
</table>


### Table 9: Main metrics for the tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>LOCs</th>
<th>LOCs sig/vars</th>
<th>events</th>
<th>hdlr</th>
<th>obs</th>
<th>hdlr</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>9</td>
<td>26</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>T2</td>
<td>10</td>
<td>28</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>T3</td>
<td>9</td>
<td>38</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>T4</td>
<td>12</td>
<td>31</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>T5</td>
<td>46</td>
<td>54</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>T6</td>
<td>41</td>
<td>49</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>T7</td>
<td>38</td>
<td>47</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>T8</td>
<td>32</td>
<td>43</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>T9</td>
<td>42</td>
<td>49</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>T10</td>
<td>61</td>
<td>82</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

**CLOC**: For the RP version: Number of signals/vars, number of events and handlers. For the OO version: Number of handles and number of observables.
9. REFERENCES


[27] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper,


