A User Study for Comparing the Programming Efficiency of Modifying Executable Multimodal Interaction Descriptions

A Domain-Specific Language versus Equivalent Event-Callback Code

Fredy Cuenca Jan Van den Bergh Kris Luyten Karin Coninx

Hasselt University - tUL - iMinds
Expertise Centre for Digital Media, Diepenbeek, Belgium
{fredy.cuencalucero,jan.vandenbergh,kris.luyten,karin.coninx}@uhasselt.be

Abstract

The present paper describes an empirical user study intended to compare the programming efficiency of our proposed domain-specific language versus a mainstream event language when it comes to modify multimodal interactions. By concerted use of observations, interviews, and standardized questionnaires, we managed to measure the completion rates, completion time, code testing effort, and perceived difficulty of the programming tasks along with the perceived usability and perceived learnability of the tool supporting our proposed language. Based on this experience, we propose some guidelines for designing comparative user studies of programming languages. The paper also discusses the considerations we took into account when designing a multimodal interaction description language that intends to be well regarded by its users.

Categories and Subject Descriptors H.5.2. [*User Interfaces*]: User interface management systems (UIMS); Evaluation/methodology

Keywords multimodal systems, domain-specific languages, declarative languages, composite events

1. Introduction

For more than a decade, the HCI community has witnessed the proposal of several user interface tools aiming at simplifying and speeding up the prototyping of multimodal systems. These tools fall within the definition of User Interface Management Systems (UIMSs) (Beaudouin-Lafon 1994). UIMSs provide domain-specific languages with which multimodal interactions can be described at a high level of abstraction and separated from the application semantics, which must be written with a general-purpose language, without support of the UIMS (Figure 1).

Some examples of the aforementioned UIMSs include Mudra (Hoste et al. 2011), ICO (Navarre et al. 2009), OIDE (Serrano et al. 2008), HephaisTK (Dumas et al. 2014), and NiMMiT (De Boeck et al. 2007). They certainly accomplish their goal of facilitating the prototyping of multimodal systems, but they all share the same issue: Their domain-specific languages require the use of concepts that are unrelated with the event languages with which programmers used to implement interactive systems in real-world projects[1]. These contrasts may raise the resistance of their potential users, as it happened in the past.

In an effort to create a usable UIMS, we tried to stick our domain-specific language as close as possible to the languages and work practices followed when developing traditional WIMP systems (WIMP = Windows, Icons, Menus, Pointers) with commonly-used event languages.

With event languages, WIMP interactive systems are created by binding predefined user events to event-handling callbacks. Similarly, with Hasselt, our proposed language, multimodal interactions are described by binding composite events to event-handling callbacks. A composite event, the core concept of Hasselt, is a user-defined event pattern that is automatically detected at runtime.

Hasselt was evaluated in a comparative user study. We asked participants to perform equivalent modifications to a simple multimodal system with both Hasselt and a mainstream event language. By combining observations, interviews, and standardized questionnaires, we managed to measure the completion rate, completion time, code testing effort, and perceived difficulty of each programming task. We

PLATEAU'15, October 26, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3907-0/15/10...
http://dx.doi.org/10.1145/2846680.2846686

[1] Two important rankings of programming language popularity can be found at `http://spectrum.ieee.org/computing/software/top-10-programming-languages` and `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

also measured the perceived usability and perceived learnability of Hasselt UIMS, the supporting tool. The results of our empirical study along with the lessons learned from it are discussed at the end of this paper.

2. Problems with the Adoption of UIMSs and Domain-Specific Languages in the Past

2.1 Programmers Resistance to Unusual Concepts

After being involved in the development of four UIMSs, Olsen Jr. stated that the "success of a UIMS is directly related to the ease with which interface designs can be expressed" (Olsen Jr 1987). He illustrates his point by confessing that the difficulty for describing interfaces in terms of grammars caused the SYNGRAPH system (Olsen Jr and Dempsey 1983) to not be widely used despite that its users realized it improved productivity. A few years after, when discussing the Mickey UIMS, a tool proposed to tackle the problems engendered by MIKE (Olsen Jr 1986), Olsen Jr. reminded us once again of the risks of including unfamiliar languages within a UIMS: "By using interface specifications based on familiar terms to programmers we were able to overcome the programmer resistance that plagued our earlier UIMS" (Olsen Jr 1989).

Some years later, a similar observation was done by Myers et al., who mentioned that domain-specific languages along with their supporting UIMSs did not catch on in the past because, among other reasons, "programmers are required to use new programming concepts" to which "programmers are not adept at thinking in terms of" (Myers et al. 2000). There the authors claim that even in the cases where the improved power of the studied tools seemed to justify a steep learning curve, many potential users did not adopt them simply because they never got past initial difficulties.

2.2 The Warnings Seem to be Overlooked

The aforementioned cases referred to domain-specific languages developed in the 1980s and targeting unimodal interaction. However, the newer languages targeting multimodal interaction and proposed in the new millennia exhibit the same problems.

For instance, when using the visual languages provided by HephaisTK (Dumas et al. 2014) and OIDE (Serrano et al. 2008), one has to describe multimodal interactions in terms of the CARE properties (Coutaz et al. 1995). For the ICO (Navarre et al. 2009) language, depicting interaction models requires a solid command of a formalism called Petri nets. For Mudra (Hoste et al. 2011), one has to use the logic-based programming language CLIPS[2]. All these UIMSs along with their underlying languages have succeeded in their goal of simplifying the prototyping of multimodal interactions, but at the expense of requiring programmers to move away from commonly used programming languages. Concepts such as CARE properties, Petri nets, or logic-based constructs, are not required by mainstream languages, such as Java or C#, and consequently many programmers may ignore these concepts and offer resistance to learn them and use them.

[2] `http://clipsrules.sourceforge.net/`

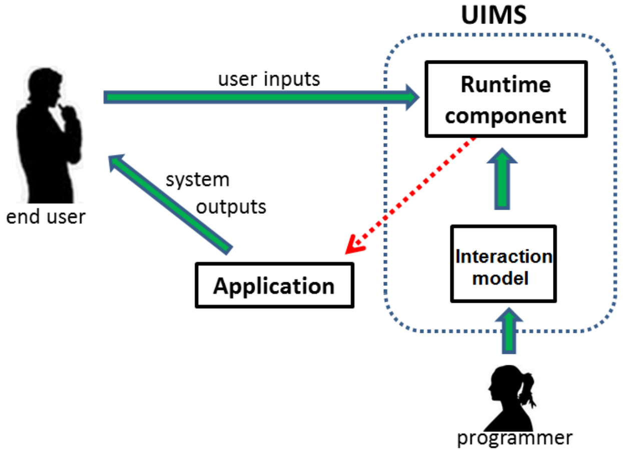


Figure 1: At runtime, the UIMS responds to its end users by launching the methods of an externally developed application. At design time, the programmer has already specified, through a domain-specific language, the methods that have to be launched for each set of user inputs.

The chances of the aforementioned languages to be adopted look even smaller when one notices that nearly all of them are visual languages in contrast to the textual nature of mainstream event languages.

Recent research reports that experienced developers show skepticism to use visual languages in practice since they still feel more comfortable with standard imperative code (Oney et al. 2014). The researchers believed, and we agree with them, that this preference may be "largely due to the relatively long-term exposure to standard code".

Hasselt does not require programmers to move away from their mental models and work practices; it maintains the textual and event-driven nature that are fundamental features of commonly used event languages to which, after decades of practice, programmers have become accustomed to, and naturally, they will not want to lose.

3. Hasselt, a Language for Rapid Prototyping Multimodal Systems

Hasselt is a declarative language aimed at describing multimodal interactions. It comes with a supporting tool, hereafter referred to as Hasselt UIMS, which includes the editors, runtime environment, and debugging tools required to code, run, and test Hasselt specifications.

3.1 Running Example

For illustrative purposes, we will use Hasselt to implement a desktop-version of the well-known *put-that-there* (Bolt 1980). This multimodal interaction enables end users to move virtual objects around a windows form through the concerted use of speech and mouse clicks. End users must utter the sentence *'put that there'* so that the pronouns *'that'* and *'there'* can be disambiguated with mouse clicks on the target object and on its new intended position respectively.

3.2 How to Use Hasselt?

The steps required to implement the *put-that-there* interaction are as follows.

3.2.1 Implementing the Back-End Application

Before using Hasselt, one typically creates an executable program implementing the front-end of the intended system and the methods to be invoked during the multimodal human-machine interaction. Such externally defined program will be referred to as back-end application.

For the *put-that-there* example, the back-end application consists of a windows form hosting several virtual objects, and a method $PutThatThere(x1, y1, x2, y2)$ for moving the object placed on $(x1, y1)$ to $(x2, y2)$.

3.2.2 Defining Multimodal Interactions

In Hasselt, a multimodal interaction is specified by defining a composite event and binding it with one or more event-handling callbacks.

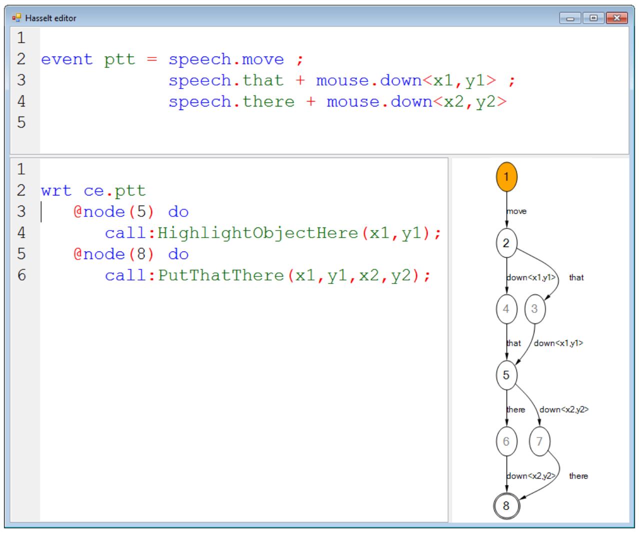
For the purposes of this work, a composite event is a user-defined combination of several events (e.g. mouse clicks, speech inputs, touch events, and body movement events) that can be combined through a series of event operators. The operator *FOLLOWED BY (;)* allows defining sequential events, the operator *AND (+)* serves to specify simultaneous events; *OR(|)* defines alternative events; and the Kleene start operator *ITERATION (*)* defines repetitive events (Cuenca et al. 2014, 2015).

Continuing with the running example, the *put-that-there* interaction is described by defining the composite event *ptt*, which must then be bound to the method $PutThatThere$, implemented in the back-end application. This is described with the following Hasselt code:

$$\begin{aligned}
&event\ \ ptt = speech.put\ ;\\
&\quad speech.that + mouse.down\langle x1, y1\rangle\ ;\\
&\quad speech.there + mouse.down\langle x2, y2\rangle\\
&wrt\ \ ce.ptt\\
&@node(8)\ do\\
&\quad call : PutThatThere(x1, y1, x2, y2);
\end{aligned} \tag{1}$$

3.2.3 Testing the Specified Multimodal Interactions

Upon entering runtime mode, Hasselt UIMS starts tracking the user-defined composite events. The (partial) detection of these composite events may cause Hasselt UIMS to invoke the methods of the back-end application.



```
event ptt = speech.move ;
            speech.that + mouse.down<x1,y1> ;
            speech.there + mouse.down<x2,y2>
```

```
wrt ce.ptt
   @node(5) do
      call:HighlightObjectHere(x1,y1);
   @node(8) do
      call:PutThatThere(x1,y1,x2,y2);
```

Figure 2: Hasselt UIMS editor. Composite events are declared in the top frame. The bindings between the composite events and the callback functions are specified in the bottom frame. The visual content of the non-editable right bottom frame is auto-generated; it displays the visual form of the composite event under edition. Callback functions are bound to the nodes/links of this graph.

Since this paper is focused on the evaluation of Hasselt, it is not possible to cover all technical details behind this language and its underlying concept of composite events. Interested readers may refer to (Cuenca et al. 2014, 2015) for more technical details, or to the publicly available videos[3,4] to watch a demonstration of the applicability of Hasselt for describing multimodal and touch interactions.

3.3 Difference Between Event Languages and Hasselt

Implementing the running example with event languages, such as Java or C#, is more complicated. Event-driven frameworks notify the speech inputs and the mouse clicks separately, as if they were independent of each other, ignoring that there is a relation between them. It is the task of programmers to write the code for determining whether these event notifications are related and whether they are arriving in the expected order of receipt. This code is difficult to maintain; it involves a multitude of state variables that have to be updated in a self-consistent manner and across different event handlers. Interested readers may refer to (Cuenca et al. 2014) for a discussion of the C# code required to implement the *put-that-there*.

The capability of Hasselt UIMS to detect patterns of events saves programmers from the difficult, error-prone

[3] `https://youtu.be/jC5EuBYWWRc`
[4] `https://youtu.be/ArX0UAHmio0`

task of implementing event pattern detection. As shown in Equation 1, with Hasselt, the stream of mouse clicks and speech inputs characterizing the *put-that-there* interaction is defined, in a declarative fashion, as the composite event ptt, which is bound with an externally defined method named $PutThatThere(x1, y1, x2, y2)$. This method will be called right after the occurrence of the composite event ptt, which will be automatically detected by Hasselt UIMS at runtime.

As an additional example, Figure 2 shows a slight variation of the interaction shown in Equation 1. Here the object to be moved is highlighted right after being selected. This requires binding ptt with two event-handling callbacks (one for highlighting, and another for moving the selected object), which are to be launched at two different moments of the interaction.

4. User Study

To the best of our knowledge, none of the aforementioned UIMSs has been evaluated in user studies. Outside the multimodal domain, we found two user studies that guided us in the design of our experiment.

4.1 Related Studies. Comparing Domain-Specific Languages with Equivalent Event-Callback Code

Oney et al. recruited 20 developers to evaluate the understandability of the Interstate's visual notation. Each participant had to modify two systems (drag-and-drop and a thumbnail viewer) implemented in both RaphaelJS[5] and InterState. It was verified that InterState models are faster-to-modify than equivalent event-callback code written in RaphaelJS (Oney et al. 2014).

The creators of Proton++ carried out two experiments with 12 programmers. Each participant was shown a gesture specification and set of videos of a user performing gestures. Gestures may be specified as a regular expression, tablature, or with event-callback code and the participant had to match the specification with the video showing the described gesture. The results showed that the tablatures of Proton++ are easier-to-comprehend than equivalent regular expressions and event-callback code (Kin et al. 2012).

Since real-world scenarios require programmers not only to comprehend but to write programming code, we followed the schema of Oney et al. We asked participants to modify an existing prototype with both Hasselt and equivalent event-callback code.

4.2 Hypotheses

Based on a pilot test, we hypothesized that the adaptation of a multimodal prototype requires (1) less time, (2) less code testing, and (3) is perceived as an easier task when using Hasselt than with C#.

[5] `http://raphaeljs.com/`

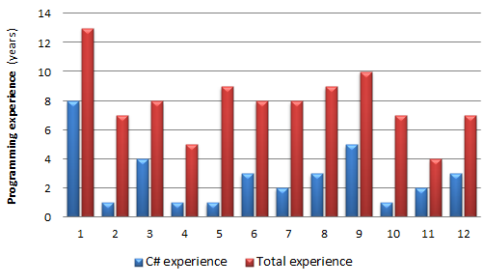


Figure 3: Programming experience of the 12 participants.

4.3 Method

4.3.1 Participants

We recruited 12 participants, all male. Their overall programming experience ranged from 4 to 13 years; and their C# experience, between 1 and 8 years (Figure 3).

4.3.2 Study Design

Participants were evaluated one by one. They were asked to modify a multimodal interaction that was described with both Hasselt and C#.

Right before the experiment, participants were given a 10-minutes tutorial about Hasselt. They were asked to describe a simple multimodal interaction by following step-by-step instructions. In this way, participants got acquainted with the code editors, debugging tools, and runtime environment of Hasselt UIMS. Since all participants had experience with C# and MS Visual Studio, there was no need for training in this respect.

The experiment starts by giving the participant a multimodal prototype with which he had to interact according to the indications of the researcher. Once the participant was familiar with the functionality of the prototype, he was asked to make slight changes in this prototype. Each participant had to sequentially perform the changes with both Hasselt and C#. The order of the language to be used first was balanced over the participants so that the aggregated experience bias can be neutralized. The changes to be performed were explained orally, but also written in a sheet that the participant could check during the experiment. The changes had to be performed within a time limit of 30 minutes per language.

While the participant modified the code, the researcher was in front of a second monitor connected to the computer used by the participant so that both monitors showed the same information. This way, the researcher could measure the completion time of the task, count how many times the partial changes were tested in the runtime environment, and watch how the participant navigated trough the code.

After the participant performed the requested changes with a certain language, he was asked to fill a post-task questionnaire for measuring the perceived difficulty of the task. Overall, each participant filled two of these questionnaires, one for Hasselt and one for C#. At the end of the user study, i.e. after both languages were used, the participant was asked to fill a usability questionnaire and immediately interviewed by the researcher.

4.3.3 Programming Task

The prototype to be modified allows end users to create and move virtual objects around a windows form. New objects can be created, in random positions, through the voice command *'create object'*. Existing objects can be moved by saying *'put that there'* while clicking on both the target object and its new position.

Participants were asked to change the command for creating objects. The new command had to be multimodal: instead of creating objects in random positions, the end user had to be able to select, through a mouse click, the position where the new object had to be placed. In the new prototype, the adverb of the new voice command 'create object here' must be disambiguated with a mouse click on the window form.

To make the comparison as fair as possible, participants were restrained to navigate the interaction code only. This is the only part of the code that can be seen with Hasselt. The code for configuring input recognizers and the code of the back-end functions, e.g. $PutThatThere(x1, y1, x2, y2)$, are not visible. The former is enclosed into Hasselt UIMS, the latter into a canned, externally developed EXE file. To confer similar complexity to the C# code, we had to divide the C# source code into regions. Right before using C#, participants were warned that the regions containing the code for configuring the input recognizers and the back-end methods must remain collapsed during the experiment. Note that this warning is not a training session: participants were not explained anything about C# or MS Visual Studio, but warned about the importance of focusing on the interaction code only.

The tutorial and the instructions sheet used in the user study are available on the web[6].

4.4 Measures

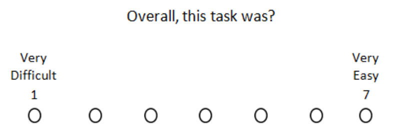
4.4.1 Observations

While the participant performs the required modifications with a certain language, the researcher monitors his working time, counts the number of times the code is tested, and watch how the participant navigates trough the code.

4.4.2 Single Ease Question (SEQ) Questionnaire

Right after completing the changes with each language, participants were asked to complete the Single Ease Question (SEQ) questionnaire (Figure 4), a rating scale ranging from 1 (anchored with "Very difficult") to 7 (anchored with "Very easy"). It aimed to assess the perceived difficulty (or perceived ease, depending on one's perspective) of a task. The

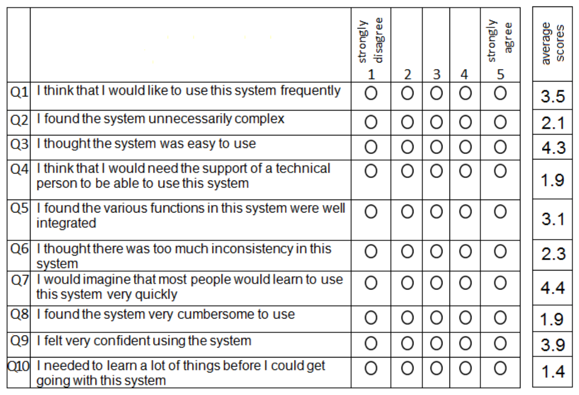
[6] `https://www.academia.edu/15725787/Supplemental_materials`



Overall, this task was?

Very Difficult 1						Very Easy 7
○	○	○	○	○	○	○

Figure 4: Single ease question (SEQ) questionnaire.



		strongly disagree 1	2	3	4	strongly agree 5	average scores
Q1	I think that I would like to use this system frequently	○	○	○	○	○	3.5
Q2	I found the system unnecessarily complex	○	○	○	○	○	2.1
Q3	I thought the system was easy to use	○	○	○	○	○	4.3
Q4	I think that I would need the support of a technical person to be able to use this system	○	○	○	○	○	1.9
Q5	I found the various functions in this system were well integrated	○	○	○	○	○	3.1
Q6	I thought there was too much inconsistency in this system	○	○	○	○	○	2.3
Q7	I would imagine that most people would learn to use this system very quickly	○	○	○	○	○	4.4
Q8	I found the system very cumbersome to use	○	○	○	○	○	1.9
Q9	I felt very confident using the system	○	○	○	○	○	3.9
Q10	I needed to learn a lot of things before I could get going with this system	○	○	○	○	○	1.4

Figure 5: System Usability Scale (SUS) questionnaire and average scores per question obtained for Hasselt UIMS.

SEQ has been proven to be reliable, sensitive, and valid while also being easy to respond (Sauro and Dumas 2009).

4.4.3 System Usability Scale (SUS) Questionnaire

After completing the experiment, participants filled the System Usability Scale (SUS) questionnaire (Brooke 1996), a well-known questionnaire for end-of-test subjective assessments of usability (Lewis and Sauro 2009).

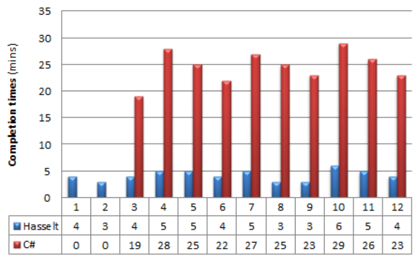
The SUS questionnaire (Figure 5) consists of 10 items with 5-point scales numbered from 1 (anchored with "Strongly disagree") to 5 (anchored with "Strongly agree"). SUS tests are quantified from 0 to 100.

To have a benchmark to which one can compare SUS scores with, Lewis et al. shared historical information showing that the average and third quartile of 324 usability evaluations performed with SUS are 62.1 and 75.0 respectively (Lewis and Sauro 2009).

Finally, according to a factor analysis performed by Lewis et al., the SUS questionnaire does not only measure usability. It also measures learnability, being $Q4$ and $Q10$ the questions that allow estimating the perceived learnability of the system under evaluation (Lewis and Sauro 2009).

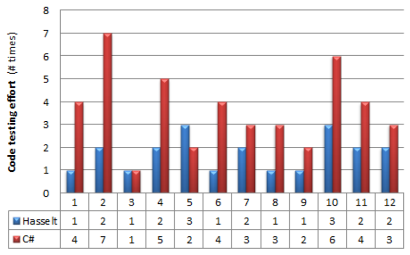
4.5 Results of the Experiment

All 12 participants completed the experiment when using Hasselt; but only 10 succeeded with C# –the others exceeded their allotted time. The raw data is shown in Figure 6.



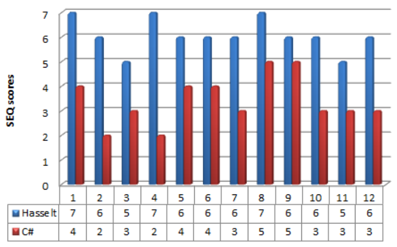
	1	2	3	4	5	6	7	8	9	10	11	12
🟦 Hasselt	4	3	4	5	5	4	5	3	3	6	5	4
🟥 C#	0	0	19	28	25	22	27	25	23	29	26	23

(a) Completion time



	1	2	3	4	5	6	7	8	9	10	11	12
🟦 Hasselt	1	2	1	2	3	1	2	1	1	3	2	2
🟥 C#	4	7	1	5	2	4	3	3	2	6	4	3

(b) Code testing effort



	1	2	3	4	5	6	7	8	9	10	11	12
🟦 Hasselt	7	6	5	7	6	6	6	7	6	6	5	6
🟥 C#	4	2	3	2	4	4	3	5	5	3	3	3

(c) Perceived ease

Figure 6: Raw data collected in the experiment. The two zeroes in the data shown in (a) correspond to those two participants who could not finish the programming task with C#.

4.5.1 Completion Time

The following results are based on those 10 participants who completed the required changes with both languages.

On average, changes made with Hasselt took 4.4 minutes in comparison with the 24.7 minutes when using C#. This difference in favor of Hasselt was statistically significant. A Wilcoxon signed-rank test rejected the null hypothesis in favor of the alternative hypothesis that Hasselt completion times are shorter (p-value = 0.0009766, $W = 0$, $Z = -2.8$).

4.5.2 Code Testing Effort

The following results are based on those 10 participants who completed the required changes with both languages.

On average, programmers tested their code 1.8 times when using Hasselt and 3.3 times when using C#. Once again, this difference in favor of Hasselt was statistically significant. A Wilcoxon signed-rank test rejected the null hypothesis in favor of the alternative hypothesis that Hasselt code requires to be tested a fewer number of times (p-value = 0.009766, $W = 2.5$, $Z = -2.4$).

4.5.3 Perceived Ease of the Task

The results on perceived ease include all the participants. Even those who could not complete the changes with C# have a clear idea of the difficulty of the task.

The average SEQ scores obtained for Hasselt and C# were 6.08 and 3.42 respectively. The perception that the changes are easier when performed with Hasselt was statistically significant. A Wilcoxon signed-rank test rejected the null hypothesis in favor of the alternative hypothesis that the SEQ scores obtained by Hasselt are higher (p-value = 0.0002441, $W = 78$, $Z = 3.1$).

4.5.4 SUS Scores of Hasselt UIMS

Comparing with the data repository provided by Lewis et al., the average SUS score of 73.96 obtained by Hasselt UIMS indicates that its perceived usability is well above average but not higher than 75% of the 324 systems reported in (Lewis and Sauro 2009).

4.6 Interview Highlights

There was unanimous consent that the required changes were easier when using Hasselt than when using C#.

When participants were asked "Why do you think it is more difficult with C#?", they refer to the fact that with event languages the human-machine interaction has to be implemented by splitting code across multiple event handlers. Concretely, Participant 11 said "It is harder with C# because it requires modifying the code in multiple places." Similarly but with his own words, Participant 3 added: "With C#, you have to check multiple variables and multiple handlers simultaneously to identify the right state of the system ... and you also have to reset the variables". Finally, Participant 2, one of the two participants who could not complete the changes with C#, confessed: "at the beginning the problem seemed quite simple but eventually you get lost while trying to maintain all the variables".

Participant 1 mentioned he had no previous experience with finite state automata (FSA), the auto-generated graphs displayed in the right bottom frame of Figure 2. Other participants had at least pen-and-paper experience with FSA. However, his total lack of knowledge about FSA did not affect his performance; Participant 1 successfully completed the required changes with Hasselt in 4 minutes. This may be an indication that the FSA auto-generated by Hasselt UIMS are so intuitive-to-read that no background may be needed. The use of the FSA is mandatory when writing Hasselt code: their nodes have to be referred to indicate the moment when the event-handling callbacks must be launched (Figure 2). The FSA generated from Hasselt code are slightly different from canonical FSA. The nodes of the former may have timers to control the temporal proximity between simultaneous inputs. The way how Hasselt UIMS handles parallel inputs was described in (Cuenca et al. 2014).

4.7 Threats to Validity

There are always threats to the validity of any empirical study, and we have tried to identify the threats to the presented user study.

First, we cannot completely guarantee that the observed differences were due solely to Hasselt and not to the specific interaction that participants had to modify. This threat could have been mitigated by performing not only one, but many experiments with different interactions. However, this was unfeasible in our case. Getting programmers who can volunteer to participate in a relatively stressful experiment that lasts more than an hour was already a difficult task. Asking them to stay for several consecutive experiments or to come in different sessions was simply unfeasible.

Second, the scores obtained with the subjective SEQ and SUS questionnaires may have been affected by the response bias. The participants were colleagues, former colleagues, former classmates, or former students of the researcher.

Finally, the SUS questionnaire may have been measured only certain aspects of the usability of Hasselt UIMS. An expert in empirical studies made us notice that usability also includes the long-term experience of using a software system, which is not considered in our study: all participants used Hasselt for the first time during the study. However, the initial learnability, which is another dimension of the SUS questionnaire, was correctly measured by $Q4$ and $Q10$, according to the same expert.

5. Discussion

5.1 Design Decisions About Hasselt. Considerations About Programming Language Adoption

We presented Hasselt, a language that provides notations for modeling multimodal interactions in a declarative manner. In order to give Hasselt real chances to be adopted, we gave it the textual and event-driven nature typical of mainstream event languages. With Hasselt, the same as with mainstream event languages, the interactions are defined by binding events with event-callback functions. The difference is that, with Hasselt, the events are defined by the programmers as combinations of other more fine-grained events.

5.2 Results. Hasselt versus C#

Hasselt clearly outperformed C# when it comes to modify multimodal interactions. The use of Hasselt led to higher completion rates, lower completion times and less code testing. Furthermore, both the SEQ questionnaires and the interviews indicate that programmers perceive that, when using Hasselt, the task can be solved more easily than when using C#. Hasselt code is simpler because it allows linking the event handlers with patterns of events.

5.3 Perceived Usability and Perceived Learnability of Hasselt UIMS

The average scores obtained by Hasselt UIMS for each of the 10 items of the SUS questionnaire are shown in Figure 5.

Considering that odd-numbered questions are positively-worded, scores higher than 3 in these items reflect that participants agree (to a certain degree) that the evaluated system presents some good aspect/feature. In our study, all odd-numbered questions were scored with more than 3 points on average. From this group, $Q3$, i.e. "I thought the system was easy to use" and $Q7$, i.e. "I would imagine that most people would learn to use this system very quickly", received the highest scores.

Similarly, since even-numbered items are negatively-worded, scores lower than 3 would indicate that participants are disagreeing (to a certain degree) with some negative comment about the system. In our studies, all even-numbered questions were scored with less than 3 points on average. From this group, $Q10$, "I needed to learn a lot of things before I could get going with this system", $Q4$, i.e. "I think I would need support of technical person to use this system", and $Q8$, i.e. "I found the system very cumbersome to use" received the lowest scores (which in this case it is something positive).

The salient scores obtained for $Q4$ and $Q10$, the questions defining learnability (Lewis and Sauro 2009), may indicate that Hasselt is perceived as easy-to-learn. This matches with the fact that all participants completed the experiment with Hasselt even though they received little training.

5.4 Lessons for the Future

Here we provide some lessons learned with respect to designing comparative studies of programming languages.

1. Careful selection of participants for the pilot test. Our pilot test consisted of one participant, who was handpicked for being one of the most experienced C# developers of our research lab. He took around 16 minutes to complete the test with C# and we considered that a time interval of 30 minutes (almost double) would be enough for all participants. This was not an optimal decision. We lost valuable data: we could not include those two participants who could not finish the experiment with C#; otherwise, the results would have been even more favorable to Hasselt.

   Future researchers may want to consider carrying out pilot tests with several randomly chosen participants. An alternative option consists of using more complex, formal mathematical models for estimating the maximum acceptable task completion time (Sauro and Kindlund 2005).

2. The number of lines of code is not a good metric to use when the programming task consists of modifying existing code. Initially, we wanted to measure the difference of lines between the original program and the modified version produced by the participant. However, we noticed that this number was going to be meaningless. First, the code added to existing lines (e.g. to the condition of an if clause) is not counted although the programming logic has changed. At the opposite side, some programmers used to break long statements into two lines and

vice versa, add or remove blank lines, comments, and region directives, etc. These actions alter the number of lines although the complexity of the programming logic remains the same.

3. The use of standardized questionnaires provides two advantages over ad-hoc questionnaires. First, the reliability and validity of the former are already proved, as in the case of SEQ and SUS. Second, since standardized tests are widely used, it may be possible to get historical data with which to compare our results.

4. We quantified code testing effort as the number of times when the user executes his program into runtime mode. In future studies, this can be complemented by measuring the time that the user spent in runtime mode. To do this, video recording the computer screen during the experiment is essential.

5. It may not be a good idea to search for participants in your research lab. Some may feel that one colleague is going to evaluate their programming skills. From a research lab with more than 50 people, we could only recruit 5 participants. The remaining 7 participants were recruited from external institutions. An alternative option would have been to ask a person from an external institution to play the role of researcher so that participants do not feel observed by an acquaintance or colleague.

6. Conclusion

This paper presented a user study that compared the programming efficiency of Hasselt versus a mainstream event language when it comes to modify multimodal interactions. The completion rates, completion time, code testing effort, and perceived difficulty of the programming tasks along with the perceived usability and perceived learnability of Hasselt UIMS were measured by means of observations, interviews, and standardized questionnaires. The paper provided some guidelines for designing comparative user studies of programming languages, and for designing a programming language that intends to be adopted.

References

M. Beaudouin-Lafon. User interface management systems: Present and future. In *From object modelling to advanced visual communication*, pages 197–223. Springer, 1994.

R. Bolt. Put-that-there: Voice and gesture at the graphics interface. In *Proc. of SIGGRAPH' 80*. ACM, 1980.

J. Brooke. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

J. Coutaz, L. Nigay, D. Salber, A. Blandford, J. May, and R. M. Young. Four easy pieces for assessing the usability of multimodal interaction: the care properties. In *InterAct*, volume 95, pages 115–120, 1995.

F. Cuenca, J. Van der Bergh, K. Luyten, and K. Coninx. A domain-specific textual language for rapid prototyping of multimodal interactive systems. In *Proc. of EICS'14*. ACM, 2014.

F. Cuenca, J. Van den Bergh, K. Luyten, and K. Coninx. Hasselt uims: a tool for describing multimodal interactions with composite events. In *Proc. of EICS'15*. ACM, 2015.

J. De Boeck, D. Vanacken, C. Raymaekers, and K. Coninx. High level modeling of multimodal interaction techniques using NiMMiT. *Journal of Virtual Reality and Broadcasting*, 4(2), 2007.

B. Dumas, B. Signer, and D. Lalanne. A graphical editor for the smuiml multimodal user interaction description language. *Science of Computer Programming*, 86:30–42, 2014.

L. Hoste, B. Dumas, and B. Signer. Mudra: a unified multimodal interaction framework. In *Proc. of ICMI'11*, pages 97–104. ACM, 2011.

K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. Proton++: a customizable declarative multitouch framework. In *Proceedings of the 25th annual ACM symposium on User interface software and technology (UIST'12)*, pages 477–486, 2012.

J. R. Lewis and J. Sauro. The factor structure of the system usability scale. In *Human Centered Design*, pages 94–103. Springer, 2009.

B. Myers, S. E. Hudson, and R. Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28, 2000.

D. Navarre, P. Palanque, J.-F. Ladry, and E. Barboni. ICOs: A Model-Based User Interface Description Technique dedicated to Interactive Systems Addressing Usability, Reliability and Scalability. *ACM Transactions on Computer-Human Interaction*, 16 (4), 2009.

D. R. Olsen Jr. Mike: the menu interaction kontrol environment. *ACM Transactions on Graphics (TOG)*, 5(4):318–344, 1986.

D. R. Olsen Jr. Larger issues in user interface management. *ACM SIGGRAPH Computer Graphics*, 21(2):134–137, 1987.

D. R. Olsen Jr. A programming language basis for user interface. In *ACM SIGCHI Bulletin*, volume 20, pages 171–176. ACM, 1989.

D. R. Olsen Jr and E. P. Dempsey. Syngraph: A graphical user interface generator. In *ACM SIGGRAPH Computer Graphics*, volume 17, pages 43–50. ACM, 1983.

S. Oney, B. Myers, and J. Brandt. Interstate: Interaction-oriented language primitives for expressing gui behavior. In *Proc. of UIST'14*. ACM, 2014.

J. Sauro and J. S. Dumas. Comparison of three one-question, post-task usability questionnaires. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1599–1608. ACM, 2009.

J. Sauro and E. Kindlund. How long should a task take? identifying specification limits for task times in usability tests. In *Proceeding of the Human Computer Interaction International Conference (HCII 2005), Las Vegas, USA*, 2005.

M. Serrano, D. Juras, and L. Nigay. A three-dimensional characterization space of software components for rapidly developing multimodal interfaces. In *Proc. of ICMI'08*, pages 149–156. ACM, 2008.