Table of Contents

International Journal of People-Oriented Programming

Volume 5 • Issue 1 • January-June-2016 • ISSN: 2156-1796 • eISSN: 2156-1788
An official publication of the Information Resources Management Association

Editorial Preface

Research Articles

Book Review

The *International Journal of People-Oriented Programming* is indexed or listed in the following: Bacon's Media Directory; Cabell's Directories; DBLP; Google Scholar; INSPEC; JournalTOCs; MediaFinder; ProQuest Advanced Technologies & Aerospace Journals; ProQuest Computer Science Journals; ProQuest Illustrata: Technology; ProQuest SciTech Journals; ProQuest Technology Journals; The Index of Information Systems Journals; The Standard Periodical Directory; Ulrich's Periodicals Directory

International Journal of People-Oriented Programming
Volume 5 • Issue 1 • January-June 2016

Hasselt:
Rapid Prototyping of Multimodal Interactions with Composite Event-Driven Programming

Fredy Cuenca, School of Mathematical Sciences and Information Technology, Yachay Tech, San Miguel de Urcuquí, Ecuador & Expertise Centre for Digital Media, Hasselt University – tUL – imec, Diepenbeek, Belgium

Jan Van den Bergh, Expertise Centre for Digital Media, Hasselt University – tUL – imec, Diepenbeek, Belgium

Kris Luyten, Expertise Centre for Digital Media, Hasselt University – tUL – imec, Diepenbeek, Belgium

Karin Coninx, Expertise Centre for Digital Media, Hasselt University – tUL – imec, Diepenbeek, Belgium

ABSTRACT

Implementing multimodal interactions with event-driven languages results in a 'callback soup', a source code littered with a multitude of flags that have to be maintained in a self-consistent manner and across different event handlers. Prototyping multimodal interactions adds to the complexity and error sensitivity, since the program code has to be refined iteratively as developers explore different possibilities and solutions. The authors present a declarative language for rapid prototyping multimodal interactions: Hasselt permits declaring composite events, sets of events that are logically related because of the interaction they support, that can be easily bound to dedicated event handlers for separate interactions. The authors' approach allows the description of multimodal interactions at a higher level of abstraction than event languages, which saves developers from dealing with the typical 'callback soup' thereby resulting in a gain in programming efficiency and a reduction in errors when writing event handling code. They compared Hasselt with using a traditional programming language with strong support for events in a study with 12 participants each having a solid background in software development. When performing equivalent modifications to a multimodal interaction, the use of Hasselt leads to higher completion rates, lower completion times, and less code testing than when using a mainstream event-driven language.

KEYWORDS

Composite Events, Declarative Languages, Event Languages, Event-Driven Programming, Interactive Systems, Multimodal Systems, Rapid Prototyping

INTRODUCTION

Rapid prototyping multimodal interactive systems consists of implementing, evaluating, and refining different types of multimodal interactions in an iterative fashion. These progressive refinements enable developers to gain a proper understanding of the strengths and weaknesses of different possible solutions. They arrive at a set of interactions that need to be supported by the final system. Rapid

DOI: 10.4018/IJPOP.2016010102

prototyping must be inexpensive in effort, since the goal is to quickly explore a wide variety of possible types of interaction. This involves building, evaluating, and throwing away many prototypes without remorse (Beaudouin-Lafon, 2003). In the remainder of this article we use the term developers to indicate developers of multimodal interactive systems that participate in rapid prototyping activities.

It is commonly accepted that the event-driven paradigm is a good match for realizing the implementation of interactive systems (Lewis & Rieman, 1993). However, in the case of multimodal interactive systems, the use of this paradigm may adversely affect the speed and cost of the rapid prototyping phase significantly. When implementing multimodal interactions, the usage of event-driven languages results in code that is dedicated in large part to the management of the interaction state. This code is then plagued with a multitude of flags that developers have to update in a self-consistent manner and across different event handlers (Spano, Cisternino, Paternò, & Fenu, 2013; Kin, Hartmann, DeRose, & Agrawala, 2012; Cuenca, Van den Bergh, Luyten, & Coninx, 2014). The resulting 'callback soup' makes it difficult to understand and to change the multimodal system source code. This complexity has to be faced for each iteration of the prototyping phase.

Several (mostly visual) languages have been proposed with the aims of facilitating the creation of multimodal prototypes (Bourguet, 2002; Dragicevic & Fekete, 2004; De Boeck, Vanacken, Raymaekers, & Coninx, 2007; Lawson, Al-Akkad, Vanderdonckt, & Macq, 2009; Navarre, Palanque, Ladry, & Barboni, 2009; König, Rädle, & Reiterer, 2010; Hoste, Dumas, & Signer, 2011; Dumas, Signer, & Lalanne, 2014). These languages allow the developer to describe multimodal interactions at a high-level of abstraction bypassing the need to manually maintain the interaction state, as it is needed with event-driven languages. To a greater or lesser extent, the aforementioned languages have accomplished their main goal of simplifying the creation of multimodal prototypes. Despite this, for many of these languages abstraction also means giving up the fine-grained control when dealing with events directly. In other words, these approaches dismiss the programming experience of developers and replace this with some formalism that hides details and introduces a more abstract terminology. Abstraction by means of visual models may not be the method of choice for many developers, who, instead, use textual languages or at least access and modify the code that drives the interactive system. Since familiarity with a language is an important factor that has a strong, positive influence in programming language adoption (Meyerovich & Rabkin, 2013), we created a language that saves developers from dealing with the 'callback soup' problem, while building upon familiar concepts and well-known programming practices.

Hasselt is a textual, declarative language that allows the description of executable multimodal interaction models. The core concept of Hasselt is a composite event, which is essentially a user-defined sequence of events that are logically related (for example, because these are part of the same interaction). Within Hasselt, developers define composite events by connecting several primitive events (e.g. touch events or speech inputs) by means of specialized operators. Each operator represents a specific relation between their operands. The overall composite event can then be bound to one or more event handlers, which specify the behavior the system should expose when the composite event occurs. At runtime, the event handlers are executed every time their associated composite events occur. For event detection, Hasselt relies on existing recognizers to process the low-level input (like speech, mid-air gestures or mouse movements) and does not replace existing recognition-based fusion engines (D'Ulizia, 2009; Nigay & Coutaz, 1995; Bouchet, Nigay, & Ganille, 2004).

One can implement the "put-that-there" interaction (Bolt,1980) —probably the best known example of multimodal interaction— in Hasselt with a composite event, ptt (Figure 2) that combines speech events and pointing events and specifies their temporal constraints (e.g. the pointing gestures must be synchronized with the spoken pronouns 'that' and 'there' to avoid ambiguities). Such a composite event can be bound to a function, putThatThere(), which will put the selected object at the specified position once the interaction is completed (i.e. once ptt occurs). When desired, one can also bind additional functions that are called before the interaction is completed (i.e. in response to the partial detection of ptt), e.g. to highlight the object identified as 'that'.

Hasselt includes a mechanism for tracking event sequences. By delegating the tracking of event sequences to its supporting tool, Hasselt developers can focus on specifying the interaction, rather than encoding and decoding the ever-changing interaction state. This dismisses a significant portion of the flags and global variables that would be required to implement the same task with traditional event-driven languages.

Hasselt was evaluated in a comparative user study for which a set of participants were asked to modify a multimodal interaction in both a mainstream event-driven language and Hasselt. Participants were developers familiar with the event-driven programming languages. The study was designed to reflect one iteration of the prototyping phase: instead of implementing interactions from scratch, developers have to read, understand, and modify existing code. The results show that, when using Hasselt, participants achieve higher completion rates, lower completion times, and the code that they produced required less validation. Despite their strong affinity with traditional programming languages, the participants expressed their appreciation for our approach with respect to the traditional approach.

RELATED WORK

Almost all tools that allow rapid prototyping of multimodal interactions provide visual languages whose models are variations of block diagrams, state machines, and Petri nets (Cuenca, Coninx, Luyten, & Vanacken, 2015).

Block Diagrams to Model Multimodal Interaction

The visual languages provided by ICon (Dragicevic & Fekete, 2004), Squidy (König, Rädle, & Reiterer, 2010), and OpenInterface (Lawson, Al-Akkad, Vanderdonckt, & Macq, 2009) allow the representation of multimodal interactions as block diagrams. Block diagrams are directed graphs whose links allow input data to flow in the direction of their arrowheads towards an externally developed application. The nodes of a block diagram can represent (1) input hardware, (2) output devices, (3) an external application that will eventually receive data, and (4) transformations to be applied to the data (e.g. data filters).

As to the particular characteristics of each tool, it can be mentioned that ICon and OpenInterface provide a set of predefined transformation nodes whereas Squidy allows users to customize the transformation nodes by writing fine-grained code. Moreover, ICon and Squidy models can only include one external application while OpenInterface can feed data into multiple applications developed in different languages. For these three tools, multimodal applications have to store the input data coming from different modalities and identify when a meaningful set of events has occurred so that an adequate system response can be conveyed. Other approaches, including Hasselt, are able to identify these meaningful sets of events directly from the user-defined declarative specifications.

Finite State Machines to Model Multimodal Interaction

When using finite state machines (FSM) to model multimodal interactions, the nodes of the FSM represent the possible states of the interaction, while the arcs represent the transitions in the interaction state caused by events. Several approaches use FSMs to model multimodal interaction but differ in how events are linked to transitions.

In particular, with MEngine (Bourguet, 2002), each arch can be annotated with only one event name. This causes MEngine models to grow too quickly when simultaneous inputs are modelled. For instance, it is known that spoken deictic terms can precede pointing inputs or vice versa during speak-and-point selection (Oviatt, 1999) When using MEngine, these two possibilities have to be explicitly specified by the user. Obviously, this gets more tedious if one has to describe interactions involving not only two, but several simultaneous inputs –in general, N inputs can arrived in N! different ways. Hasselt UIMS protects its users from this state explosion: Hasselt developers only have to specify

which inputs are to be simultaneous (by using the AND operator) and, at runtime, its supporting tool will internally make all of the necessary arrangements to deal with all possible orders of arrival.

Compared to MEngine in NiMMiT (De Boeck, Vanacken, Raymaekers, & Coninx, 2007), one can annotate several event names to one single arc of a model. Such arcs will be traversed only if all its associated events occur simultaneously. Furthermore, one does not have to explicitly specify all possible orders of arrival in which the inputs can be sensed; NiMMiT hides the order of arrival of simultaneous events. One limitation of NiMMiT is that its events cannot carry parameters, which increases the number of function calls needed to compensate. E.g. every time one needs to refer to the cursor position during a mouse click, a function that returns this information has to be invoked. Instead, Hasselt allows events to carry parameters; the values of which are automatically set by its supporting tool.

In HephaisTK (Dumas, Signer, & Lalanne, 2014) models, there is a clear separation between the specifications of events and the dialog model, which, in our opinion, enhances their readability. In HephaisTK, each arc of its FSM-based models is annotated with a user-defined event pattern and an event-handling callback. Callbacks are launched when predefined event patterns occur, thus causing the system to switch to a new state. To define an event pattern, HephaisTK users have to specify the relation among its constituent events using the CARE properties. The CARE framework (Coutaz et al., 1995) defines the possible combinations of modalities in multimodal interaction: Complementary (two or more modalities are combined synergistically during an interaction), Assigned (one modality used for one interaction), Redundant (two or more equivalent commands are issued simultaneously through multiple modalities), and Equivalent (one out of several modalities can be chosen to issue a command). A limitation of HephaisTK is its inability to provide partial feedback. Unlike HephaisTK, Hasselt allows binding event-handling callbacks at very specific moments during detection of the multimodal command thus enabling partial feedback.

Petri Nets to Model Multimodal Interaction

ICO is a language intended for formal descriptions of multimodal interactive systems (Navarre, Palanque, Ladry, & Barboni, 2009). It has been successfully applied in the field of safety- critical systems. With ICO, one can describe a wide variety of interactions by depicting them in Petri nets-based models. By exploiting the well-studied mathematical apparatus behind Petri nets, some properties about ICO models can be predicted in static time, before running the model. But the use of a general-purpose mathematical modeling language has disadvantages too: Petri nets were not specifically created for modelling computerized systems, much less for multimodal systems. Not surprisingly, it does not have the notations for describing the special characteristics of multimodal interaction in a straightforward way. Other languages, with higher domain-specificity map closer to the multimodal domain than does ICO. In Hasselt, for instance, the modalities involved in the interaction are explicitly specified and each possible relation between modalities can be represented with one designated symbol. Empirical studies have shown that the more domain-specific a language is, the more accurate and more efficient developers are in program comprehension (Kosar, Mernik, & Carver, 2012). This efficiency is desirable in the prototyping phase, where the interaction descriptions have to go through multiple design-implement-test loops.

Logic Rules to Model Multimodal Interaction

Mudra (Hoste, Dumas, & Signer, 2011) allows the description of multimodal interactions with a textual notation. When comparing different models of the interaction put- that-there, we observed that the specification obtained with Mudra was more concise (in space) than other equivalent visual specifications. This conciseness is significantly beneficial for its users: the less material to be scanned, the higher is the proportion that can be held in working memory, and the lower the disruption caused by frequent searches through the model (Green & Petre, 1996). Mudra strongly influenced our decision to create Hasselt as a textual language. Mudra specifications have to be written in CLIPS,

which was specifically designed for expert systems. Therefore, Mudra does not map as closely to the multimodal domain as other domain-specific languages such as Hasselt and HephaisTK. Mudra requires viewing multimodal interactions by using the logic-based paradigm: In Mudra, the events are not viewed as notifications that have to be handled as they occur, as is the case with Hasselt and mainstream event-driven languages. Instead, Mudra events have to be viewed as information that is accumulated in a database that will be queried by the CLIPS engine from time to time. This type of approach fails when patterns need to be detected as soon as they really occur (Anicic, Fodor, Stuhmer, & Stojanovic, 2009).

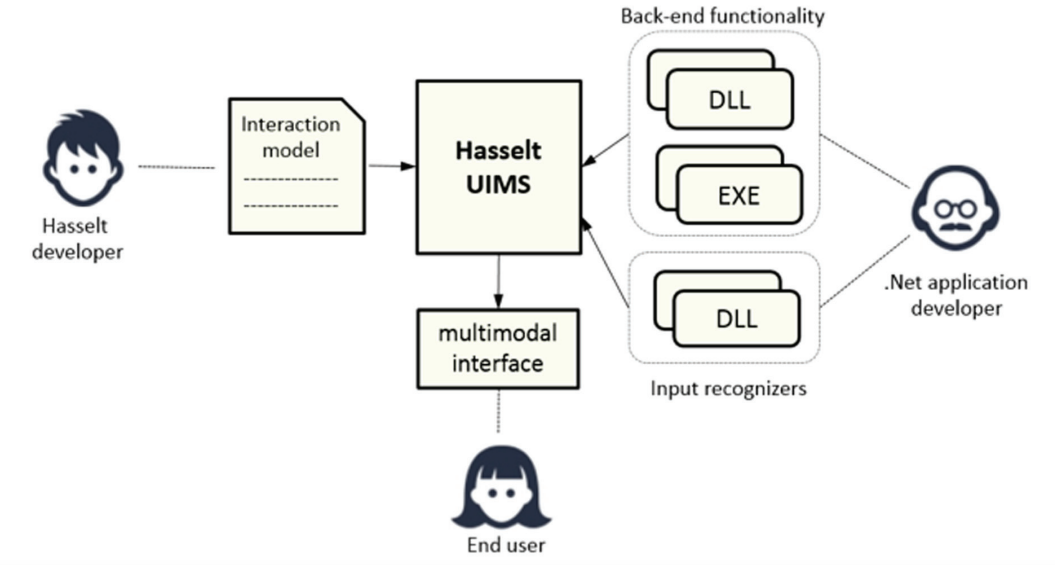
HASSELT'S PROTOTYPING ENVIRONMENT

Hasselt is part of a User Interface Management System (UIMS) suite, hereafter called Hasselt UIMS. It includes a code editor, runtime environment and debugging tools for writing, running, and evaluating Hasselt programs. In order to realize a multimodal interface, Hasselt UIMS requires an interaction model and back-end functionality (Figure 1).

The interaction model describes the interplay between the end user and the multimodal prototype, while the back-end functionality includes a set of callback functions that will be launched, at runtime, in response to user actions. Whereas the interaction model can be specified with Hasselt, the back-end functionality is encoded in .Net compatible libraries, without support from Hasselt UIMS. The Hasselt runtime environment allows for linking with .Net libraries and import the required functionality this way.

At runtime, the Hasselt code is 'glued' with the back-end functionality. This results in an executable multimodal prototype that the end user can interact with. In Hasselt, multimodal interactions are described as mappings of composite events to event handlers. The composite events represent coordinated sets of user actions; the event handlers encode all potential system responses. Hasselt

Figure 1. Hasselt UIMS, the tool supporting Hasselt, links people with different roles in the prototyping process; a Hasselt developer specifies an interaction model that calls upon .Net libraries and executables provided by a .Net developer, to create a prototype for an end user



developers can quickly explore a variety of multimodal interactions by defining and redefining these mappings in a declarative fashion.

The runtime environment of Hasselt UIMS incorporates a set of input recognizers (abstractions of input hardware) for managing events from various input devices ranging from mouse and keyboard to multi-touch screens, microphone input and depth cameras (like the MS Kinect and the SoftKinetic DS325 and DS326). In addition, new custom recognizers can be created and added for those projects that need to manage additional input devices.

HASSELT: A LANGUAGE TO SPECIFY MULTIMODAL INTERACTION

Hasselt is a domain-specific, declarative language that essentially allows for (1) declaration of composite events, and for (2) binding these composite events to event handlers.

Declaring Composite Events

An *atomic event* is an abstraction used to represent a signal generated by input hardware (like a voice signal or a frame generated by Kinect). It is called atomic because it cannot be defined as a combination of other more fine-grained events within Hasselt.

A *composite event* is a combination of several events with associated constraints. The events to be combined can be atomic events or previously defined composite events. Unlike atomic events, which occur in an instant, composite events occur over a significant time interval. To define a composite event, its constituent events are interconnected with event operators. Table 1 shows them in increasing order of precedence. Explicit use of parentheses is allowed to force the evaluation order of the terms. For instance, the composite events A;B|C and (A;B)|C are treated differently: the former will be triggered upon the detection of event A followed by either B or C whereas the latter will be triggered after the consecutive occurrence of A and B or, alternatively, upon the detection of C.

Binding Composite Events to Event Handlers

The Hasselt UIMS runtime can call event handlers both during and at the end of the detection of a composite event. This is possible because, at design time, Hasselt UIMS generates a Finite State Machine (FSM) for each composite event. Hasselt developers can attach function call statements to each node of the FSM, thus specifying the moment when the event handlers have to be called. Aside from launching the functions of the back-end applications, Hasselt also permits other types of system responses, as shown in Table 2.

Put-That-There Example

This section illustrates how one can implement a variation of the multimodal interaction *put-that-there* (Bolt, 1980) with Hasselt. In contrast to the original put-that-there in this case, a mouse is used for

Table 1. Event operators supported by Hasselt

Event Operator	Example	Semantics
NEGATION (−)	A-B	During event A, event B cannot occur
FOLLOWED BY (;)	A;B	B occurs after A
OR (|)	A|B	A or B occur
AND (+)	A+B	A and B occur simultaneously, meaning both occur within a pre-defined timeframe
ITERATION (*)	A*	A occur zero or more times

Table 2. Available types of system responses in Hasselt

System Response Type	Description
call:ns.cls.subName	Call routine subName of the namespace ns and class cls of the back-end application
raise:eventName	Generate an event that can be captured by other composite event
assign:lstVarAssign	Assign values to weakly typed variables
speak:expression	Speak sentence through text-to-speech
play:filePath	Play an audio file

pointing. The prototype permits users to move virtual objects around a windows form by saying the sentence *'put that there'* in conjunction with the mouse. While saying the pronouns *'that'* and *'there'*, the user has to simultaneously click on the target object and then its intended position, respectively.

The Hasselt code (Figure 2) required for this interaction has three distinguishable parts: (1) composite event declarations, (2) finite state machines (FSMs), and (3) composite event binding code. The non-editable FSMs appear automatically right after a composite event is declared. These FSMs are the linking element that allows binding composite events to event handlers.

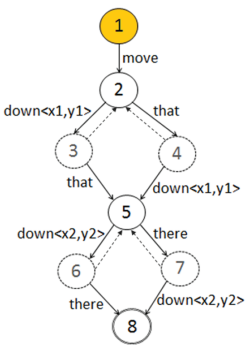
Atomic event names consist of two parts separated by a dot: the first part refers to the input modality (e.g. speech); the second part refers to the event itself (e.g. put). According to the code, the composite event ptt will be triggered upon the detection of the speech input *'put'* followed by the co-occurrence of *'that'* and a mouse click, and this, in turn, followed by the co-occurrence of the input *'there'* and another mouse click. The triggering of ptt will cause the execution of the function putThatThere() of the .Net class ppt.Ptt, which was encoded for moving virtual objects over a windows form.

The FSM next to it is used to link user inputs and system responses. Every time a callback function is attached to a node (or link) of a FSM, one is implicitly declaring the moment when such a function has to be called. In the Figure, the two alternative paths from node 2 to node 5 (and the same is true for paths from node 5 to node 8) shown cater for two possible situations: although the speech input and mouse clicks are expected to occur simultaneously, one will always proceed the other by some minuscule amount of time.

The event binding code starts with the statement wrt ce.{eventName}, which stands for: *with respect to composite event {eventName}* . In Figure 2, the code is binding the method putThatThere() to the final state of ptt, i.e. node(8), which means that putThatThere() will be called when the

Figure 2. Composite event ptt defined with Hasselt. The upper part of the code contains the declaration of ptt; the lower part is the code for binding ptt to two event handlers. Event handlers can be launched at different stages of the composite event lifespan; all these stages are represented in the auto-generated FSM on the right side.

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

wrt ce.ptt
  @node(5) do
    call:  ptt.Ptt.highlightObject(x1,y1);
    speak: 'where to move the object?';
  @node(8) do
    call: ptt.Ptt.putThatThere(x1,y1,x2,y2);
```



interaction is completed, that is, once the event ptt is fully detected. Node 5 is also voice annotated with a statement via the built-in speech synthesizer of Hasselt UIMS (speak) as well as a function that highlights an object on screen, so that end users receive acknowledgement that the prototype is correctly interpreting their inputs.

Variables in Hasselt

Variables in Hasselt are not declared, but are implicitly created and are scoped to a composite event. How this happens differs according to their initialization:

- **Event parameters:** The information carried by atomic events comes encapsulated in variables called event parameters (e.g. tscreen.up<x,y,t,id>). Such parameters can be passed to event handlers or be used to define conditional expressions. Once a composite event is fully detected (i.e. the final state of its reciprocal FSM is reached), the parameter values of all its constituent atomic events are cleared. Event parameters are defined in external DLLs, called input recognizers (Figure 1);
- **Local variables:** Hasselt local variables can be created at any stage of the composite event lifespan. Hasselt local variables are declared and maintained with the keyword assign (Table 2), e.g. assign: count=0, sum=sum + 1. There is no need to specify the datatype of Hasselt local variables; these will be treated as if they had the same datatype of their initial value;
- **Callback-generated variables:** These variables contain the returning values of the functions implemented in the back-end applications. Callback-generated variables do not have to be defined explicitly. Hasselt UIMS automatically creates a variable with the same name of the function and sets it with the return value. Callback-generated variables can be used, for example, to process the output generated by an externally developed gesture recognition library;
- **Properties:** Hasselt offers properties (i.e. auto-maintained variables) that simplify the description of interactions. Some properties, e.g. _lastNode and _lastEvent, allow reference to past interaction states; these two properties, for instance, can be used to conditionally execute rollback functions. Other properties (e.g. Now.TotalMilliSeconds) help ease the specification of time constraints.

PROOF-OF-CONCEPT APPLICATION

In this section, we discuss the features of Hasselt UIMS using a proof-of-concept multimodal application, called Couch Potato. This is a multimodal application that allows wireless and remote control of a media player. Users can choose, play, pause, and stop their favorite movies through the coordinated use of touch screen, body posture, and speech.

Couch Potato

First, Couch Potato displays an enumerated list of movie names, which can be scrolled through by voice commands or touch gestures. By saying ‘next’, ‘previous’, or more flexible commands like ‘four steps forward’ or ‘ten steps backward’, users can navigate the list to select a video. Alternatively, a user can draw a number on the touch-sensitive screen of his smartphone; this number is interpreted as the index of the video to be selected. Both selection methods can be used alternatively.

Once a video is selected, one can play it by flicking right on the smartphone while pointing it towards the screen where a Kinect sensor is positioned. For this multimodal command, Couch Potato combines two input modalities: (a) full body input to detect pointing towards the screen with Kinect and (b) flicking to the right on the smartphone’s touch screen. Similarly, as the video plays, one can point to the screen and flick to the left or tap on the smartphone in order to stop or pause the playback, respectively. When the video stops, the video list is shown again. The playback volume can also be increased/decreased by flicking up/down when in playback mode. Couch Potato is closed down when the user says ‘goodbye’ while waving his right hand in front of the Kinect sensor.

Back-End Applications

Couch Potato uses three back-end applications: a Windows application, a generic dynamic link library (DLL) for gesture stroke recognition, and a generic mobile application.

The Windows application presents a form hosting a video player, with a list box containing the names of the video files located in a specific directory. This Windows application implements both the presentation part and the functions for controlling the media player. The DLL contains a function that receives a series of (x-y)-points and returns a string with the name of the 0-9 digit encoded in those points, or the string 'none' when no match is possible. Both the Windows application and the DLL were imported into Hasselt UIMS. The mobile application that translates touch events into TUIO messages is the open-source TUIOdroid.

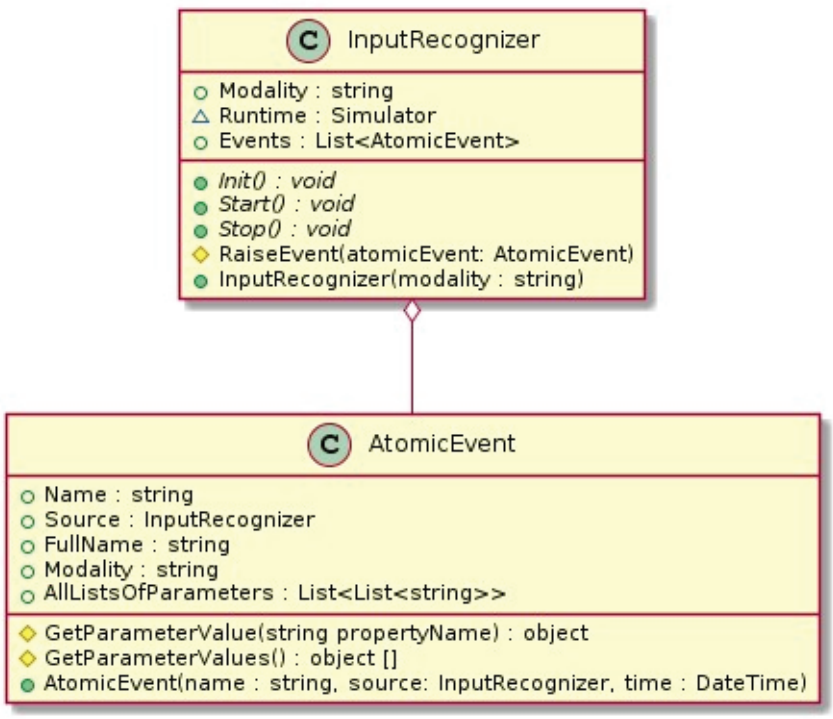
Input Recognizers

The Couch Potato prototype uses three predefined recognizers incorporated into Hasselt UIMS. These enable skeleton tracking via Microsoft Kinect API, speech recognition via Microsoft Speech API, and touch event detection via TUIO.

Each recognizer implements a subclass of the class InputRecognizer, which includes methods for configuring, starting, and stopping the operation of input hardware and raising atomic events. In addition to those, one or more subclasses of the class AtomicEvent (Figure 3) are needed. For each atomic event, the event name and possible parameters need to be specified. For each input recognizer used, about 200 lines C# code are required.

Importing input recognizers into Hasselt UIMS has two results: (1) At design time, the Hasselt grammar is internally updated so that a new set of atomic events are available to Hasselt developers, who can thereafter describe multimodal interactions that involve more sensors. (2) At runtime, new

Figure 3. Classes to be implemented to let Hasselt UIMS support input hardware



types of hardware are automatically activated (deactivated) upon entering (leaving) runtime mode, and their signals are encoded as events through the whole runtime.

Specification of Multimodal Interactions with Hasselt

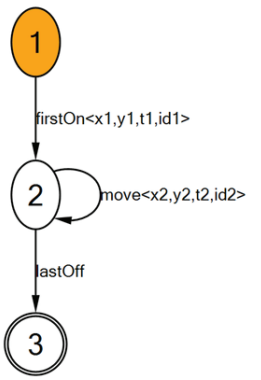
Hierarchical Interaction Specification

Starting a video is achieved by combining multimodal events: flicking to the right and extending the right hand forward are two events that when occurring simultaneously cause the selected video to play (Figure 4). The composite event flickRight occurs when the user flicks towards the right on his smartphone screen. This event is declared as a sequence of touch events: one initial touch.down event followed by an arbitrary number of touch.move's and one final touch.up event. Two constraints are imposed to guarantee that the touch moved to the right, i.e. x2 > x1, and that this movement was horizontal, i.e. abs (y2 - y1) < 0.05. The composite event handFront occurs when the user is pointing forward: when his right hand is at least 35 cm in front of his body. The parameter skl carried by the

Figure 4. Couch potato enters into playback mode when a user flicks to the right on a smartphone (*flickRight,* a) while pointing forward (*handFront,* b) at the same time (*playVideo*, c)

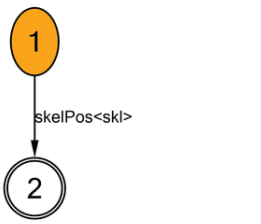
(a)

```
event flickRight =
  tscreen.firstOn<x1,y1,t1,id1>;
  tscreen.move<x2,y2,t2,id2>*;
  tscreen.lastOff

wrt ce.flickRight
  @link(2, tscreen.move<x2,y2,t2,id2>) do
  when id1 = id2;
triggers when x2 > x1
  and abs(y2 - y1) < 0.05
```



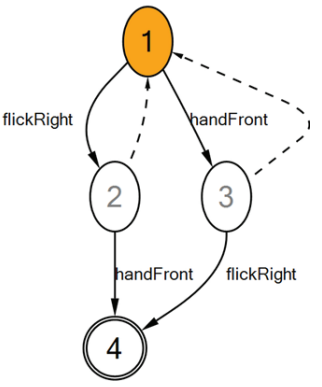
(b)

```
event handFront = kinect.skelPos<skl>

wrt ce.handFront
triggers when
  skl.handRight.Z < skl.Head.Z-0.35
```



(c)

```
event playVideo = ce.flickRight +
  ce.handFront


wrt ce.playVideo
  @node(4) do
    call:mediaPlayer.Form1.play();
```



atomic event kinect.skelPos, generated by the Kinect recognizer, is a data structure containing the (x,y,z)-positions of skeleton joints. The prefix ce, preceding flickRight and handFront, indicates that these two events have already been defined as composite events. The function play(), contained in the media player application, will be launched when playVideo is detected, i.e. when flickRight and handFront co-occur. This interaction illustrates how one can reduce the complexity of defining complex interactions by using composite events without associated event handler, such as flickRight and handFront. These composite events can be reused to achieve different interactions. E.g. handFront is reused in the definitions of stop and pause interactions.

Handling Simultaneous Inputs with Time-Out Transitions

In the FSM of Figure 4c, the dashed links outgoing from node 2 and node 3 towards node 1 represent time-out transitions that will be automatically executed if the events handFront and flickRight do not arrive within a time interval (whose length is predefined in the configuration file of Hasselt UIMS). Time-out transitions (dashed links) appear when a composite event contains simultaneous events, i.e. when the operator AND (+) is used. Time-out transitions thus guard correct execution of the operator AND (+): the interaction moves to its final state (in this case node 4) only if the two involved inputs (flick gesture and body pose) co-occur within a time interval. Otherwise, i.e. if the time interval expires when only one input has been detected, the time-out transitions are executed, thus resetting the interaction to its initial state (node 1).

Use of Arrays: Free-Form Gesture Recognition

Hasselt allows the collection of event parameters (e.g. touch position) into arrays, which can then be passed to back-end applications (e.g. gesture recognition libraries). Couch Potato allows users to select the N^{th} element of the video list by drawing the number N on his smartphone screen. Numbers to be drawn can consist of one or many unistroke 0-9 digits drawn in a quick succession. This interaction is defined with the composite events digit and number. In the definition of the event digit (Figure 5), all the points of a stroke as well as their timestamps are collected into arrays that are passed to the function getBestMatch() once the stroke is finished. This function belongs to the DLL that was imported into Hasselt UIMS. As mentioned, a variable getBestMatch is created that contains the value returned by the function of the same name. In this case it is a string containing the name of the depicted digit or 'none' if the stroke did not match with any digit template. This enables expressions such as getBestMatch <> 'none'.

The event number (Figure 6) is composed out of a stream of digit events, e.g. digit<'two'>, digit<'six'>, that finishes after 2.5 seconds of 'silence'. The event number collects the parameters carried by the digit events into an array, e.g. d = ['two', 'six'], that is passed to the back-end method chooseVideo, which selects the video whose index is indicated in the input parameter, i.e. the 26^{th} video.

Figure 5. Composite event digit. Parameters carried by atomic (touch) events down and move are accumulated into arrays.

```
event digit<getBestMatch> =
  tscreen.down<xs[],ys[],ts[],ids[]>;
  tscreen.move<xs[],ys[],ts[],ids[]>*;
  tscreen.up<xn,yn,tn,idn>

wrt ce.digit <getBestMatch>
  @link(2, tscreen.move<xn,yn,tn,idn>) do
    call:g2d.utils.getBestMatch(xs,ys,ts);
triggers when getBestMatch <> 'none'
```

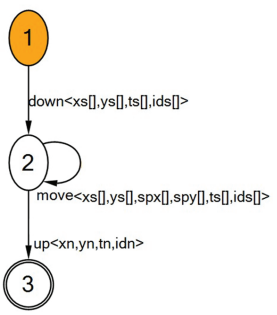
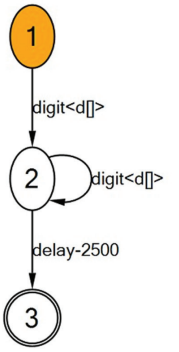


Figure 6. Composite event number. Parameters carried by digit are accumulated into arrays.

```
event number = ce.digit<d[]>;
  ce.digit<d[]>*; delay-2500


wrt ce.number
  @node(3) do
    call:mediaPlayer.Form1.chooseVideo(d);
```



Interruptibility: Cancelling Partially Entered Commands

End users may sometimes decide to interrupt a partially entered command to start issuing a new one. Hasselt UIMS facilitates the implementation of such a scenario by allowing developers to declare a reset command, which causes an immediate reset of Hasselt UIMS: local variables are destroyed and FSMs return to their initial state. The reset command (e.g. speech.reset) is declared in a configuration file that Hasselt UIMS reads at startup.

In some cases, aborting the tracking of composite events may leave the system in an inconsistent state. For instance, if the reset command is detected after the system has already performed some internal computation, it may be convenient to roll back the effects produced so far. Just as Hasselt UIMS resets its local variables, the back-end applications are expected to include roll back functions to reset their internal variables.

For the rollback functions to be launched in the right scenario, it is important to distinguish whether composite events return to their initial state after reaching the final state (normal termination), or else, after a reset command (abort termination). By using the property _lastNode, Hasselt developers can restrict the invocation of roll back functions only for those cases when the reset command was raised. More technically, roll back functions have to be attached to the initial nodes and their invocations have to be restricted to the case: when _lastNode <> N, where N is the index of the final node.

Passive Inputs

Whereas active inputs are intentionally generated by the end user to command a system (e.g. speech), passive inputs are unintentionally issued (e.g. facial expressions or incidental manual gestures) and can be exploited by the system to proactively help the end user.

Couch Potato can react to passive inputs. If the end user leaves the room, Couch Potato will automatically pause the video, which will be automatically played again once the end user is back. Such an interaction can be described by using the atomic events kinect.userOn (kinect.userOff), which are fired by the Kinect recognizer every time the end user appears (disappears) from the Kinect's field of view. These two events are not part of the Microsoft Kinect API; these were implemented by the application developer, thus hiding complexity from Hasselt developers.

EVALUATION OF HASSELT

We gathered 12 participants in order to evaluate whether programming with composite events brings about benefits when it comes to modify multimodal systems. Event-driven programming was used as the baseline paradigm. This section is a summary of the most relevant aspects of the experiment described in (Cuenca, F., Van den Bergh, J., Luyten, K., & Coninx, K.).

Hypothesis

Based on the results of a pilot test, we hypothesized that the adaptation of a multimodal interaction model requires (1) less time, (2) less code testing, and (3) less mental effort when using composite events than when using traditional event-callback code These hypotheses were tested by a within-subjects experiment in which participants are required to perform equivalent modifications with both Hasselt and C#.

The variables are operationalized as follows: The amount of time for performing the requested changes is counted from the moment the participant starts modifying the code until he informs the researcher about the completion of the task. The amount of testing involved during the experiment is measured as the number of times the participant enters into runtime mode. The mental effort required by a programming task was obtained with the subjective post-task Single Ease Question (SEQ) questionnaire. It uses a rating scale ranging from 1 (anchored with "Very difficult") to 7 (anchored with "Very easy") and is aimed to assess the perceived difficulty (or perceived ease, depending on one's perspective) of a task (Sauro & Dumas, 2009).

Participants

We recruited 12 participants, all of which are male. The overall programming experience of the participants ranged from 4 to 13 years ($M = 7.9$, $SD = 2.3$); and their C# experience, between 1 and 8 years ($M = 3.0$, $SD = 2.2$). The pool of participants included master and PhD students, post-docs, and industry developers, from different universities and countries, and with different backgrounds (computer science and engineering).

Procedure

The study was a within-subject experiment and was preceded by a short training session of 10-15 minutes. By following step-by-step instructions, one by one, the participants were able to describe a simple multimodal interaction with Hasselt. In this way, they got acquainted with Hasselt and Hasselt UIMS. Since all had experience with C# and MS Visual Studio, there was no need for training in that respect.

In the experiment, each participant was shown a multimodal prototype that he had to interact with, according to instructions from the researcher. Once he was familiar with the functionality of the prototype, he was asked to make some changes; these changes had to be performed with both Hasselt and C# within a time limit of 30 minutes per language. The order of the languages to be used is balanced over the participants so that the aggregated experience bias was neutralized overall. After the experiment the participants fill out a questionnaire and were interviewed.

The prototype they were required to modify 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 issuing *'put that there'* while clicking on both the target object and then its new position. Participants were asked to adapt the command for creating objects so that the end user is able to select using a mouse click, the position where the new object has to be placed. The changes only required modifying the interaction code, not the application-specific code.

Results

All 12 participants completed the experiment when using Hasselt; but only 10 succeeded with C# — the other two exceeded their allotted time. For completion time and code testing effort we analyzed the 10 participants that completed both conditions. For the other variable the analysis includes the results of all participants.

Completion Time

Changes made with Hasselt took on average 4.4 minutes (SD = 0.97) compared to 24.7 minutes (SD = 3.02) when using C#. 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 = −2.8085). Participant 2, who did not finish the C# task, mentioned in the interview that he eventually got lost in the maintenance of the state variables when using C#.

Code Testing Effort

On average, participants tested their code significantly less when using Hasselt (M = 1.8 times, SD = 0.75) than when using C# (M = 3.3 times, SD = 1.72). We reject the null hypothesis based on a Wilcoxon signed-rank test in favor of the alternative hypothesis "Hasselt code is tested less frequently" (p-value = 0.009766, W = 2.5, Z = −2.4233). Participants 1 and 2, who did not complete the experiment, showed the highest difference in code testing effort (300% and 250% additional tests, respectively); while other participants did between 50% and 150% additional tests with C#.

Perceived Ease of the Task

All participants gave higher SEQ scores to Hasselt. Participants rated the task with Hasselt to be easy (M = 6.08, SD = 0.67) while they rated the task with C# to be slightly difficult (M = 3.42, SD = 1.00). A Wilcoxon signed-rank test showed this difference to be significant in support of the alternative hypothesis that Hasselt's SEQ scores are higher (p-value = 0.0002441, W = 78, Z = 3.0953). In the interview the difference in rating was explained by the split over multiple event handlers. E.g. participant 11 mentioned: "It is harder with C# because it requires modifying the code in multiple places." Participant 3 mentioned: "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."

DISCUSSION

It has been reported that familiarity with a language has a strong, positive impact in programming language adoption, even more positive than performance, reliability, or language semantics (Meyerovich & Rabkin, 2013). Based on this report and the past experiences described below, Hasselt was designed so that interactions can be described by means of event binding, as with traditional event-driven languages.

Design Decisions About Hasselt: Why Textual? Why Event-Driven?

Almost all languages provided by the studied rapid prototyping tools are visual languages and/or require using concepts such as CARE properties, transition rules, or logic-based concepts, which are unrelated to event languages. These concepts may thus be unknown even to developers with experience in interactive systems. We conceded this may be a design issue since past experiences show that deviating developers from their 'native languages' brings about negative consequences.

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 in describing interfaces in terms of grammars caused the SYNGRAPH system (Olsen Jr. & Dempsey, 1983) to not be widely used despite the improved productivity realized by its users realized. A few years after, when discussing the Mickey UIMS, a tool proposed to tackle the problems engendered by MIKE (Olsen Jr., 1986), its author reminded us once again of the risks of including unfamiliar languages within a UIMS: "By using

interface specifications based on familiar terms to developers we were able to overcome the developer resistance that plagued our earlier UIMS" (Olsen Jr., 1989).

Influence of Previous Programming Experience

The previous cases highlighted the resistance of developers to use unfamiliar concepts. The present study warns about the potential consequences of adopting languages that are clearly different from the languages one is accustomed to. In another study, a group of master and doctoral students had to expand on a project consisting of describing a system with the language Live Sequence Charts (LSC), the syntax of which was unknown to the participants, as were the underlying concepts. Instead, they had experience with other programming languages, mainly C++ and Java (Alexandron, Armoni, Gordon, & Harel, 2012). The results showed that previous programming experience leads developers not only to misunderstand or misinterpret concepts that are new to them, but that it can also lead them to actively distort the new concepts in a way that enables them to use familiar programming patterns, rather than exploiting the new ones to good effect. Learners of the new language not only interpret the new models through the prism of the previous models they are familiar with — this is the straightforward implication of a theory called constructivism (Ben-Ari, 2001) —, but they actively try to force the new model to behave like the model they are familiar with, so they can use previously acquired programming solutions.

Skepticism Towards Visual Languages

Since many of the existing languages aimed at describing multimodal interactions are visual languages, it is also important to reflect on the experiment carried out by Oney et al. (Oney, Myers, & Brandt, 2014). They enlisted 20 developers to perform equivalent modifications with both InterState, a visual language, and RaphaelJS, a textual, event language.

These researchers reported that, during the interviews, the participants (experienced developers) showed skepticism about using visual languages in practice since they still felt more comfortable with standard imperative code. The authors hypothesized that this preference may be "largely due to the relatively long-term exposure to standard code". Not even the enhanced efficiency achieved with InterState in comparison with equivalent event-callback code could seduce the participants to consider using visual languages in real- world scenarios.

Based on these experiences, it is clear that when designing a new language, one cannot simply overlook the previous programming experience of its potential users. The rankings of programming language popularity published by IEEE ("IEEE Spectrum", 2016) and by TIOBE ("TIOBE Index", 2016) agree that most widely-used languages to date are textual, and a predominant proportion of them subscribe to the event- driven paradigm. Therefore, Hasselt was designed to retain the textual and event-driven nature that are fundamental features of commonly-used event languages to which, after decades of practice, developers have become accustomed to, and naturally, they will not want to give up.

Hasselt Simplifies the Creation of Multimodal Interactive Prototypes

Below we discuss how Hasselt helps reduce the "callback soup" obtained when prototyping multimodal interactions with event languages.

Updating Interaction State

When implementing multimodal interactions with event languages, developers have to update several state variables that altogether encode the interaction state. For the *put-that-there* interaction (Bolt, 1980), for instance, state variables have to be updated for every relevant speech input and pointing gesture until the whole interaction is completed. These updates have to be implemented manually, in a self-consistent manner, and across different event handlers. By contrast in Hasselt, developers are saved from the error-prone task of maintaining state variables, as demonstrated by

the actual experience of participants in the evaluation as well as the assessment in the questionnaire and interview. The interaction state is internally updated by Hasselt UIMS while tracking composite events: for each interaction, Hasselt UIMS 'knows' whether this is in its initial stage, (node 1), final stage, or somewhere in between.

Identifying Current Interaction State

When implementing multimodal interactions with event-driven languages, developers must write conditional clauses for distinguishing between interaction states, e.g. "if interaction state is X; system must respond with Y". These conditional clauses can be more or less complex depending on the number of state variables that need to be interrogated. By contrast in Hasselt, the interaction state can be referred to directly, in an explicit manner, e.g. "when in node(A), function B is called", without the need of conditional clauses for interrogating state variables.

Fusing Inputs from Different Event Handlers

With event languages, the event data (e.g. mouse cursor position) is carried by the parameters of the event handlers (e.g. MouseEventArgs), which can only be referred to from within the event handlers (local scope). Therefore, the event data may have to be saved in a wider scope (e.g. global variables) in order to make it visible to other event handlers needed to deal with the same multimodal interaction. This trick of saving event data in global variables for its subsequent fusion with the data carried by other related events is not needed in Hasselt. Hasselt variables can be referred to at any moment of the interaction, i.e. at any moment during the composite event lifespan. In existing event languages, local variables are alive within one event; global variables, throughout the whole runtime; but a new scope for maintaining variables across a particular sequence of events —as introduced by Hasselt— can be better tailored for describing multimodal interactions. Developers can then use such (scoped by composite events) variables and avoid littering the code with too many global variables whose only purpose is to make event data visible at the moment of fusion. Maintenance of variables in C# was also an important cause of complexity mentioned by participants in the evaluation presented before.

Limitations of Hasselt

The creation of multimodal prototypes may be hindered by the low range of fine-tuning allowed by Hasselt. Some functionalities offered by Hasselt UIMS are 'hermetically sealed' and cannot be tweaked, which restricts Hasselt developers to a subset of the interactions that can be implemented with event-callback code. Defining the tempo with which the voice messages are to be synthesized or invoking back-end functionality asynchronously exemplify two operations that cannot be defined with Hasselt. Therefore, such a fine tuning is not possible with the current version of Hasselt or done by the application developer. This is because the present work focused on evaluating the feasibility, pros and cons of extending the concepts of event and event binding with the aims of facilitating rapid prototyping. Augmenting Hasselt with additional notations to increase the level of fine tuning during prototyping, remains part of future envisaged work.

A further limitation involves the finite state machines (FSMs). The strong dependency between the event binding code and the FSMs implies that every time a FSM changes (because the corresponding composite event is redefined), the event binding code may have to be updated. E.g. some nodes of the original FSM may not exist in the new FSM or may have a different index. For future versions of Hasselt we will explore alternative ways to refer to the timeline points of the human-machine interaction, e.g. using after:speech.move instead of @node(2). However, this not trivial since many events can lead to the same node or the same event may occur in different nodes.

CONCLUSION

This article presents Hasselt, a declarative language that was designed as an alternative to event-driven languages in the rapid prototyping of multimodal interactions. One reason to look for an alternative to event-driven languages in that prototyping phase is the 'callback soup' problem associated with handling events. Such programs are plagued with a multitude of flags and state variables that have to be maintained in a self-consistent manner, across different event handlers, and for each iteration of the prototype. The ability to compose events that allow developers to describe multimodal interactions at a high level of abstraction, and thereby avoid the aforementioned 'callback soup', is the distinguishing feature of Hasselt. In doing so it reduces the risk of a project delivery going overtime. By taking into account the disadvantages of other proposed languages that push developers beyond familiar concepts and their programming practices, we designed Hasselt to maintain the textual and event-driven nature of well-known event languages, allowing them to describe multimodal interactions in familiar terms. We do this by binding (composite) events to event handlers.

The enhanced simplicity of Hasselt in comparison with event-driven languages was noticed in practice by twelve participants, who were asked to perform slight modifications to a mouse-and-speech interaction with both languages. They unanimously agree, in both interviews and SEQ questionnaires, that the required changes were more easily performed with Hasselt than with C#. This subjective perception is in line with the objective fact that, during the same study, Hasselt led to higher completion rate, lower completion times, and less code testing.

ACKNOWLEDGMENT

This research was partly funded by the ClaXon. ClaXon is a project co-funded by IMEC, a digital research institute founded by the Flemish Government. Project partners are Audi Brussels, Robovision, SoftKinetic, Melexis, and AMS.

REFERENCES

Alexandron, G., Armoni, M., Gordon, M., & Harel, D. (2012, November). The effect of previous programming experience on the learning of scenario-based programming. *Proceedings of the 12th Koli Calling International Conference on Computing Education Research* (pp. 151-159). ACM. doi:10.1145/2401796.2401821

Anicic, D., Fodor, P., Stuhmer, R., & Stojanovic, N. (2009, August). Event-driven approach for logic-based complex event processing. *Proceedings of the International Conference on Computational Science and Engineering CSE'09* (Vol. 1, pp. 56-63). IEEE. doi:10.1109/CSE.2009.402

Beaudouin-Lafon, M. (1994). User interface management systems: Present and future. In *From object modelling to advanced visual communication* (pp. 197–223). Springer Berlin Heidelberg. doi:10.1007/978-3-642-78291-6_7

Beaudouin-Lafon, M., & Mackay, W. (2003). Prototyping tools and techniques. In *Human Computer Interaction-Development Process*, 122-142.

Ben-Ari, M. (1998, March). Constructivism in computer science education. In ACM SIGCSE bulletin, 30(1), 257-261. doi:10.1145/273133.274308

Bolt, R. A. (1980). "Put-that-there": Voice and gesture at the graphics interface. ACM SIGGRAPH Computer Graphics, 14(3), 262–270.

Bouchet, J., Nigay, L., & Ganille, T. (2004, October). ICARE software components for rapidly developing multimodal interfaces. *Proceedings of the 6th international conference on Multimodal interfaces* (pp. 251-258). ACM. doi:10.1145/1027933.1027975

Bourguet, M. L. (2002). A toolkit for creating and testing multimodal interface designs. *companion proceedings of UIST* (Vol. 2, pp. 29-30).

Cass, S. (2016, December 31). The 2016 Top Programming Languages. *IEEE Spectrum*. Retrieved from http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

Coutaz, J., Nigay, L., Salber, D., Blandford, A., May, J., & Young, R. M. (1995). *Four easy pieces for assessing the usability of multimodal interaction: the CARE properties* (pp. 115–120). Springer, US: Human—Computer Interaction.

Cuenca, F., Coninx, K., Luyten, K., & Vanacken, D. (2015). Graphical Toolkits for Rapid Prototyping of Multimodal Systems: A Survey. *Interacting with Computers*, *27*(4), 470–488. doi:10.1093/iwc/iwu003

Cuenca, F., Van den Bergh, J., Luyten, K., & Coninx, K. (2014, June). A domain-specific textual language for rapid prototyping of multimodal interactive systems. *Proceedings of the 2014 ACM SIGCHI symposium on Engineering interactive computing systems* (pp. 97-106). ACM. doi:10.1145/2607023.2607036

Cuenca, F., Van den Bergh, J., Luyten, K., & Coninx, K. (2015). A user study for comparing the programming efficiency of modifying executable multimodal interaction descriptions: a domain-specific language versus equivalent event-callback code. *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools* (pp. 31-38). ACM. doi:10.1145/2846680.2846686

D'Ulizia, A. (2009). Exploring multimodal input fusion strategies. The Handbook of Research on Multimodal Human Computer Interaction and Pervasive Services: Evolutionary Techniques for Improving Accessibility, 34-57. doi:10.4018/978-1-60566-386-9.ch003

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

Dragicevic, P., & Fekete, J. D. (2004, October). Support for input adaptability in the ICON toolkit. *Proceedings of the 6th international conference on Multimodal interfaces* (pp. 212-219). ACM. doi:10.1145/1027933.1027969

Dumas, B., Lalanne, D., & Ingold, R. (2010). Description languages for multimodal interaction: a set of guidelines and its illustration with SMUIML. *Journal on multimodal user interfaces, 3*(3), 237-247.

Dumas, B., Lalanne, D., & Oviatt, S. (2009). Multimodal interfaces: A survey of principles, models and frameworks. In *Human machine interaction* (pp. 3–26). Springer Berlin Heidelberg. doi:10.1007/978-3-642-00437-7_1

Dumas, B., Signer, B., & Lalanne, D. (2014). A graphical editor for the SMUIML multimodal user interaction description language. *Science of Computer Programming*, *86*, 30–42. doi:10.1016/j.scico.2013.04.003

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: A cognitive dimensions framework. *Journal of Visual Languages and Computing*, *7*(2), 131–174. doi:10.1006/jvlc.1996.0009

Hoste, L., Dumas, B., & Signer, B. (2011, November). Mudra: a unified multimodal interaction framework. *Proceedings of the 13th international conference on multimodal interfaces* (pp. 97-104). ACM.

Kin, K., Hartmann, B., DeRose, T., & Agrawala, M. (2012, May). Proton: multitouch gestures as regular expressions. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 2885-2894). ACM.

König, W. A., Rädle, R., & Reiterer, H. (2010). Interactive design of multimodal user interfaces. *Journal on Multimodal User Interfaces*, *3*(3), 197–213. doi:10.1007/s12193-010-0044-2

Kosar, T., Mernik, M., & Carver, J. C. (2012). Program comprehension of domain-specific and general-purpose languages: Comparison using a family of experiments. *Empirical Software Engineering*, *17*(3), 276–304. doi:10.1007/s10664-011-9172-x

Lawson, J. Y. L., Al-Akkad, A. A., Vanderdonckt, J., & Macq, B. (2009, July). An open source workbench for prototyping multimodal interactions based on off-the-shelf heterogeneous components. *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems* (pp. 245-254). ACM. doi:10.1145/1570433.1570480

Lewis, C., & Rieman, J. (1993). Task-centered user interface design. *A Practical Introduction*.

Meyerovich, L. A., & Rabkin, A. S. (2013). Empirical analysis of programming language adoption. *ACM SIGPLAN Notices*, *48*(10), 1–18. doi:10.1145/2544173.2509515

Mitra, S., & Acharya, T. (2007). Gesture recognition: A survey. *IEEE Transactions on Systems, Man and Cybernetics. Part C, Applications and Reviews*, *37*(3), 311–324. doi:10.1109/TSMCC.2007.893280

Myers, B., Hudson, S. E., & Pausch, R. (2000). Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction*, *7*(1), 3–28. doi:10.1145/344949.344959

Navarre, D., Palanque, P., Ladry, J. F., & Barboni, E. (2009). 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), 18. doi:10.1145/1614390.1614393

Nigay, L., & Coutaz, J. (1995, May). A generic platform for addressing the multimodal challenge. *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 98-105). ACM Press/Addison-Wesley Publishing Co. doi:10.1145/223904.223917

Olsen, D. R. Jr. (1986). MIKE: The menu interaction kontrol environment. *ACM Transactions on Graphics*, *5*(4), 318–344. doi:10.1145/27623.28868

Olsen, D. R. Jr. (1987). Larger issues in user interface management. *Computer Graphics*, *21*(2), 134–137. doi:10.1145/24919.24932

Olsen, D. R., Jr. (1989, March). A programming language basis for user interface. In ACM SIGCHI Bulletin, 20(SI), 171-176. doi:10.1145/67449.67485

Olsen, D. R. Jr, & Dempsey, E. P. (1983). SYNGRAPH: A graphical user interface generator. *Computer Graphics*, *17*(3), 43–50. doi:10.1145/964967.801131

Oney, S., Myers, B., & Brandt, J. (2014). Interstate: Interaction-oriented language primitives for expressing gui behavior. *Proc. of UIST* (Vol. 14). doi:10.1145/2642918.2647358

Oviatt, S. (1999). Ten myths of multimodal interaction. *Communications of the ACM*, *42*(11), 74–81. doi:10.1145/319382.319398

Sauro, J., & Dumas, J. S. (2009, April). Comparison of three one-question, post-task usability questionnaires. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1599-1608). ACM. doi:10.1145/1518701.1518946

Spano, L. D., Cisternino, A., Paternò, F., & Fenu, G. (2013, June). GestIT: a declarative and compositional framework for multiplatform gesture definition. *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems* (pp. 187-196). ACM. doi:10.1145/2494603.2480307

TIOBE. (2016, December 31). Index for December 2016. Retrieved from http://www.tiobe.com/tiobe-index/

Turk, M. (2014). Multimodal interaction: A review. *Pattern Recognition Letters*, *36*, 189–195. doi:10.1016/j.patrec.2013.07.003

Fredy Cuenca is a professor in the School of Mathematical Sciences and Information Technology at Yachay Tech, Ecuador. Since he received his PhD degree, from Hasselt Universiteit, he is dedicated to investigate multimodal interaction description languages and programming environment tools.

Jan Van den Bergh is a senior researcher at Hasselt University and member of the Human-Computer Interaction lab of the research institute Expertise Centre for Digital Media. He has a PhD from Hasselt University and Maastricht University. His research focus is on user-centered engineering of context-sensitive or collaborative user interfaces. His current research includes programming and modeling languages for the creation or configuration of multimodal and collaborative interfaces for human-robot interaction as well as professional translation environments.

Kris Luyten is a professor in Computer Science at Hasselt University and member of the Human-Computer Interaction lab of the research institute Expertise Centre for Digital Media. His research interest is exploring new techniques and methods to engineer and use context-aware interactive systems. Since a few years, he is working on creating more accessible, usable and approachable ubiquitous systems, focusing on intelligibility in its various shapes and forms. More information on his endeavours in research and teaching can be found on his website www.krisluyten.net.

Karin Coninx is full professor at Hasselt University, Belgium. She leads the Human-Computer Interaction Group in the Expertise Centre for Digital Media at UHasselt and is engaged in a mandate as Vice-Rector Education. Her research interests include user-centred methodologies, (engineering approaches for) multimodal interaction, haptic feedback, virtual environments, technology-supported rehabilitation and CareTech, serious games, mobile and context-sensitive systems, and interactive work spaces.

Call for Articles

International Journal of People-Oriented Programming

Volume 5 • Issue 1 • January-June 2016 • ISSN: 2156-1796 • eISSN: 2156-1788
An official publication of the Information Resources Management Association

MISSION

The primary mission of the **International Journal of People-Oriented Programming (IJPOP)** is to be instrumental in the improvement and development of the people-oriented programming, appealing to both academics and practitioners. It also educates a wider audience discussing the conceptualization, design, programming, configuration and orchestration of self-fashioned tools and products that ultimately suit the user's own unique needs and aspirations. The journal publishes original material of high quality concerned with the theory, concepts, techniques, methodologies and the tools that service a market-of-one—the empowered user.

COVERAGE AND MAJOR TOPICS

The topics of interest in this journal include, but are not limited to:

Activity theory and modeling • Agent meta-models, mental models • Alert filter and notification software, automated task assistance • Augmented reality, augmented interaction • Automating personal ontologies, personalised content generation • Client-side conceptual modeling • Computational models from psychology • Context-aware systems, location-aware computing, ubiquitous computing • Cultural probes, self-ethnography • End-user composition, end-user multi-agent systems • Game development support tools • Game mods, game engines, open game engines • Home network applications • Human-centered software development • Interface generators, XML-based UI notation generators • Interface metaphors • Life logs, life blogs, feed aggregators • Mashups, mashup tools, cloud mashups • Model-driven design, didactic models, model-based design and implementation • New generation visual programming • People-Oriented Programming (POP) • People-Oriented Programming case studies • Personal interaction styles, touch and gestures • Personal ontologies and taxonomies • Personalisation, individualisation, market of one • Personalized Learning • Personas and actors • Real-time narrative generation engines • Role-based modeling • Service science for individuals • Situated computation, social proximity applications • Smart-phone mashups, home network mashups, home media mashups • Software analysis & design, software process modeling • Software component selection • Speech and natural language interfaces • Storyboarding, scenarios, picture scenarios • Task flow diagrams, Task-based design • Task models, task analysis, cognitive task models, concurrent task modeling • Use case models, user interface XML notations • User interface tools, XML-based UI notations • User modelling, end user programming, end user development • User-centered design, usage-centered design • Wearable Computing • Wearable computing, bodyware • Web-service orchestration, web-service co-ordination

ALL INQUIRIES REGARDING IJPOP SHOULD BE DIRECTED TO THE ATTENTION OF:

Steve Goschnick, Editor-in-Chief • IJPOP@igi-global.com

ALL MANUSCRIPT SUBMISSIONS TO IJPOP SHOULD BE SENT THROUGH THE ONLINE SUBMISSION SYSTEM:

http://www.igi-global.com/authorseditors/titlesubmission/newproject.aspx

IDEAS FOR SPECIAL THEME ISSUES MAY BE SUBMITTED TO THE EDITOR(S)-IN-CHIEF

PLEASE RECOMMEND THIS PUBLICATION TO YOUR LIBRARIAN

For a convenient easy-to-use library recommendation form, please visit:
http://www.igi-global.com/IJPOP