

Towards a composite event-based language for
describing multimodal interactions

Fredy Cuenca

Promotor: Prof. Dr. Karin Coninx
Copromotor: Prof. Dr. Kris Luyten

January 27, 2016

I am weary of my wisdom, like the bee that has gathered too much honey; I need hands outstretched to take it from me.

—Friedrich Nietzsche, *Thus Spoke Zarathustra*

Abstract

Implementing interactive systems with event languages requires writing sub-routines, called event handlers, which, at runtime, are automatically called when external events occur. When implementing multimodal systems with event languages, the interaction code gets split into several event handlers within which a multitude of flags and state variables have to be manually maintained in a self-consistent manner, thus complicating programmers' work.

The present research aims at simplifying this complexity with a language that empowers programmers to define event sequences, herein called composite events, each of which can be bound to one or more event handlers. At runtime, these event handlers will be called automatically at different stages of the composite event detection process. Hasselt, the proposed language, comes accompanied with a supporting tool that includes the editors, compilers, runtime environment, and debugging tools required to write, syntax-check, run, and test Hasselt programs, respectively.

Using an event language as a baseline, Hasselt was evaluated by both static and dynamic testing. First, code inspection was used to evaluate the complexity of equivalent source codes written with both Hasselt and an event language. Among many results, the inspection showed that Hasselt code is shorter and simpler since it releases programmers from manually tracking sequences of events. Second, a user study was conducted to compare programming efficiency. After modifying an interaction model with both Hasselt and the baseline language, it was revealed that the former leads to higher completion rates, lower completion times, and less code testing.

The results obtained during this research imply that augmenting existing event languages with notations for defining composing events may be one way to reduce the accidental complexity of implementing multimodal systems.

Dedication

This thesis is dedicated to the people whose unconditional friendship gave me the strength to overcome the always present problems of a longstanding project. To Jan and Sean, my dearest friends, my partners in crime, for filling the first two years of my Ph.D. studies with a wealth of anecdotes that will be forever engraved in my memory. And to Yelena, who, with her warm smile and radiant personality, managed to melt the frost of loneliness in which I was about to freeze.

Acknowledgments

The completion of this thesis could not have been accomplished without the support of many people for whom I feel the moral duty to acknowledge.

Firstly, I would like to express my sincere gratitude to my advisor Prof. Karin Coninx for giving me the opportunity to do my Ph.D. study in a competitive environment as I always wanted. Her constant enthusiasm, tolerance, and understanding, typical of a trustworthy leader, were the motivation that impelled me in the search for excellence at each stage of the Ph.D. research.

Besides my advisor, I would like to thank Prof. Kris Luyten, Dr. Jan Van den Bergh, Dr. Davy Vanacken, and Dr. Mieke Haesen for their insightful comments and encouragement, but also for the constant hard questions which incited me to widen my research from various perspectives. I am also appreciative with Philippe Palanque, for his suggestions to my work and for hosting my PhD internship at Université Paul Sabatier de Toulouse, France.

Finally, I cannot let this opportunity pass without expressing my admiration for the work of a handful of researchers, the giants on whose shoulders I stood up, namely, Robert J. K. Jacob, Michel Beaudouin-Lafon, Brad Myers, Sharon Oviatt, Miro Samek, and Bruno Dumas. In the large amount of scientific literature, these authors inspired me with stimulating papers, full of beautiful style and perspicuous content.

The present research was financially supported by the Special Research Fund (BOF) of Hasselt University.

Contents

Abstract	iii
Dedication	v
Acknowledgments	vii
Contents	xiv
List of Figures	xx
List of Tables	xxii
1 Introduction	1
1.1 Motivation	2
1.2 Research goals	4
1.3 Research approach	4
1.4 Contributions	6
1.5 Supporting publications	6
1.6 Thesis outline	9
1.7 Summary	10
2 Background and Related Work	13
2.1 Background	13
2.1.1 Interaction styles: multimodal, multitouch, cross-device	13
2.1.2 Important programming paradigms	17
2.1.3 Prototyping and Rapid Prototyping	18
2.1.4 User Interface Management System (UIMS)	19
2.1.5 Accidental complexity and essential complexity	20
2.1.6 Finite State Automaton and Finite State Machine	22

2.2	Related Work	23
2.2.1	Multimodal interaction description languages	23
2.2.2	Gesture Description Languages	29
2.2.3	Human-machine dialog modeling languages	34
2.3	Summary	34
I	Hasselt, a family of languages	37
3	Hasselt UIMS, a composite event-based tool	39
3.1	Hasselt UIMS overview	39
3.1.1	Workflow	40
3.1.2	People and roles involved	42
3.1.3	Startup configuration	43
3.2	Lifecycle of Hasselt programs	43
3.2.1	Design time	43
3.2.2	Compile time	44
3.2.3	Runtime	48
3.3	Algorithms used at compile time	52
3.3.1	CEDL compiler. From composite events to FSA	52
3.3.2	SRDL compiler. From FSA to FSM	53
3.4	Summary	56
4	CEDL: Composing user events	57
4.1	Composite Event Definition Language (CEDL)	57
4.1.1	Atomic events	57
4.1.2	Composite events	63
4.2	Put-That-There in Hasselt UIMS	65
4.2.1	Implementing back-end applications	65
4.2.2	Declaring composite events	65
4.2.3	Binding composite event with event handlers	66
4.2.4	Testing the multimodal interactions	66
4.3	CEDL advanced features	67
4.3.1	Arbitrary speech input	67
4.3.2	Arrays of variables	68
4.3.3	Timeout events	68
4.3.4	Compositional definitions	69
4.4	Limitations of the CEDL	70
4.5	Summary	71

5	SRDL: Responding to composite events	73
5.1	System Response Definition Language (SRDL)	74
5.1.1	Multiple system responses at different times	74
5.1.2	Hasselt variables	75
5.1.3	Hasselt properties	77
5.1.4	Hasselt guard and triggering conditions	77
5.1.5	Hasselt user-defined events	78
5.1.6	Types of constraints describable by Hasselt	79
5.2	Enhancing put-that-there with SRDL	79
5.3	Describing touch and body gestures	82
5.3.1	Single-stroke touch gestures	82
5.3.2	Multi-stroke touch gestures of arbitrary length	83
5.3.3	Multitouch gestures	85
5.3.4	Free-form hand gestures	86
5.3.5	Body movements	88
5.4	Technical details	91
5.4.1	Management of parallel inputs	91
5.4.2	Interruptibility and rolling-back	92
5.4.3	Evaluation of expressions	93
5.4.4	Speech recognition grammars	94
5.5	Expressiveness of CEDL/SRDL	95
5.5.1	Negation of events	95
5.5.2	About the CARE properties	96
5.5.3	Types of feedback	99
5.6	Limitations of SRDL	100
5.7	Summary	101
6	HMD2L: Separating events from dialog model	103
6.1	Hasselt's visual language: The Human-Machine Dialog Defini- tion Language (HMD2L)	104
6.1.1	HMD2L within Hasselt	104
6.1.2	HMD2L models	105
6.1.3	Differences between auto-generated FSMs and HMD2L models	106
6.2	Proof-of-concept application	108
6.2.1	Couch Potato. A Multimodal Video Player	108
6.2.2	Implementation	109
6.2.3	Passive inputs	115
6.3	Limitations of HMD2L	117

6.4	Summary	117
II	Assessment of Hasselt	119
7	Code comparison of two different paradigms	121
7.1	Cognitive Dimensions	122
7.2	Interaction models	123
7.2.1	Code inspecting a multimodal interaction	123
7.2.2	Code inspecting a multitouch interaction	129
7.3	Dialog models	134
7.3.1	Implementation of the baseline system	134
7.4	Wrapping up the results	139
7.4.1	About interaction models	139
7.4.2	About dialog models	140
7.5	Threats to validity	140
7.6	Summary	141
8	User Study	143
8.1	Hypotheses	144
8.2	Method	144
8.2.1	Study Design	144
8.2.2	Participants	145
8.2.3	First Experiment. CEDL/SRDL versus C#	145
8.2.4	Second Experiment. HMD2L versus C#	148
8.3	Measures	148
8.3.1	Observations	148
8.3.2	Single Ease Question (SEQ) questionnaire	148
8.3.3	System Usability Scale (SUS) questionnaire	149
8.4	Results	150
8.4.1	Modifying an interaction model: CEDL/SRDL vs. C#	150
8.4.2	Modifying a dialog model: HMD2L vs. C#	151
8.5	Observations	154
8.6	Usability and learnability of Hasselt UIMS	156
8.7	Interview highlights	156
8.8	Threats to validity	157
8.8.1	Construct validity	157
8.8.2	Internal validity	158
8.8.3	External validity	159

8.9	Lessons for the future	160
8.10	Summary	161
III Discussion, Conclusions, and Future Work		163
9	Discussion	165
9.1	Design of a composite-event based language	165
9.1.1	Why textual? Why event-driven?	165
9.1.2	Why these notations?	168
9.2	Evaluation of a composite event-based language	169
9.2.1	Interaction models	169
9.2.2	Dialog models	171
9.3	Engineering problems tackled by CEDL/SRDL	172
9.3.1	The selection ambiguity problem	173
9.3.2	The problem of dual-faced gestures	176
9.4	Contributions of the thesis	177
9.4.1	Contributions in the tooling	177
9.4.2	Contributed algorithms	179
9.4.3	Contributions in engineering	180
9.4.4	Contributions in user study design	181
9.5	Limitations of Hasselt UIMS	182
9.5.1	Fixed set of atomic events	182
9.5.2	Inability to describe two-handed multitouch gestures	183
9.5.3	Negative consequences of separating interaction code from application code	183
9.6	Summary	185
10	Conclusions and Future Work	187
10.1	Conclusions	187
10.2	Future Work	189
10.2.1	Evaluation methods	189
10.2.2	Applied research	189
10.2.3	Exploring alternative directions	190
10.2.4	Towards a composite event-based language	191
10.3	Long term vision	193
10.3.1	The need of a guiding star. Stopping the Babel-like confusion of languages	193
10.3.2	Extrapolating the past for envisioning the future	193

10.3.3 Composite event-based programming. The guiding star	194
10.4 Summary	196
Appendices	199
A Theoretical background	199
A.1 Exponential Smoothing Filter	199
A.2 Augmented Transition Network (ATN)	200
B Composite events in other domains	201
B.1 Active databases as composite event processors	201
B.2 Complex Event Processing (CEP) as a service	202
C Source code	205
C.1 Feedback about the error recognition inputs	205
C.2 Rolling back	206
C.3 Management of of redundant inputs	208
C.4 Ambiguity of gestures <i>plus</i> and <i>equal</i>	209
C.5 Dual-faced gestures problem	211
C.6 Couch Potato	213
C.7 Back-end application for the put-that-there	218
D User study	221
D.1 Tutorial for user studies	221
D.2 User study tasks	226
D.3 Statistical tools	226
D.3.1 Boxplot	226
D.3.2 Q-Q normality plots	226
D.3.3 Wilcoxon signed-rank test	229
D.4 Raw data	229
D.5 Other standardized questionnaires	229
D.5.1 Standardized post-task tests	229
D.5.2 Standardized usability tests	232
Bibliography	247

List of Figures

2.1	At runtime, different types of user inputs, such as mouse clicks, speech inputs, or touch inputs, are received by the runtime component ①, which is in charge of launching the methods of an externally developed application ③, according to the specifications of an interaction model ②. The externally developed application can combine different types of outputs, such as text, audio, images, animation, or video, to respond the user. The interaction model is elaborated with a declarative language that can be textual or visual; it specifies which methods of the application have to be called in response to which user inputs. . .	21
2.2	Multimodal interaction represented with a (a) Flow-based model, (b) State-based model.	27
2.3	(a) Petri nets-based model for multimodal interaction, (b) Logic-based code for describing a gesture.	32
2.4	Languages for describing context-of-use-dependent interactions. CoGenIVE shows how the human-machine dialog changes; each context-of-use represents a set of interaction techniques available to the end user.	33
3.1	Artifacts and roles involved in a Hasselt project. At runtime, Hasselt UIMS senses and responds to the end user actions by launching the methods of the back-end applications according to the specifications of the interaction model and dialog model. Whereas the interaction model and dialog model are specified with the languages of the Hasselt family, the back-end applications (EXE applications and/or DLL libraries) are externally developed with .NET languages.	40

-
- | | | |
|-----|--|----|
| 3.2 | The three tabs of the References window allows declaring the (1) EXE applications, (2) DLL libraries, and (3) previously defined Hasselt programs that are to be imported into the current project. Hasselt UIMS allows extensibility and modularization of code. | 42 |
| 3.3 | Editors for the three languages comprising the Hasselt family, namely Composite Event definition Language (CEDL), System response Definition Language (SRDL), and Human-Machine Dialog Definition Language (HMD2L). | 45 |
| 3.4 | Design time and compile time architectures. A composite event <i>ce</i> is transformed into a finite state automaton <i>fsa</i> , which is to be annotated with system responses, <i>sr</i> , thus resulting a finite state machine, <i> fsm</i> . The system responses may refer to the methods of externally defined back-end applications. Both CEDL editor and SRDL editor are integrated into the same form, as shown in Figure 3.3a. The FSM created in the HMD2L editor, if exists, will be treated in the same way as the FSMs auto-generated from CEDL and SRDL at runtime. | 46 |
| 3.5 | Typical flow of compile time activities [Wu 10]. The compilers incorporated in Hasselt UIMS followed the same flow except that they do not generate binary files, but finite state machines (FSMs). | 46 |
| 3.6 | Chain of transformations undergone by the composite event drag-and-drop. (1) The CEDL code is transformed into a parse tree by a third-party component. (2) This parse tree is converted into a finite state automaton (FSA) by Algorithm 1. (3) The SRDL code is parsed, once again, by the Irony library. (4) The nodes and links of the FSA are augmented with system outputs, according to the SRDL specifications (Algorithm 2), thus resulting in a finite state machine (FSM). Algorithm 1 and Algorithm 2 are shown at the end of this chapter. | 49 |
| 3.7 | Runtime architecture. The finite state machines (FSMs) are fed with user events detected by the input recognizers and internally-generated events. Its transitions may activate the back-end applications and/or the synthesizers. | 50 |
| 3.8 | Hasselt runtime environment. The back-end application (BE) the end user is interacting with was built in C# and imported through a designated window. | 52 |

3.9	The CEDL provides four types of operators that one can use to compose events (Section 4.1.2). Each of these operators implies different operations in the internal process of transforming the CEDL code into FSA. This figure illustrates the effect of each event operator when applied to state machines sd1 and sd2 , shown in (a). (b) CONCATENATE(sd1, sd2) . (c) OVERLAY(sd1, sd2) . (d) PERMUTE(sd1, sd2) and (e) LOOP(sd1)	55
4.1	Intellisense technology is used by the CEDL editor of Hasselt UIMS.	58
4.2	The pink-colored circle represents the first of three touches placed on a touchscreen device. The figure shows how the angles with respect to the first touch are calculated and, in particular, what the value of the field <i>avgAngle2First</i> will be for this case.	62
4.3	Artifacts involved in the implementation of the <i>put-that-there</i> interaction. (a) Windows-form application developed in C# without support of Hasselt UIMS. (b) Binding the composite event <i>moveObject</i> to the method <i>PutThatThere</i> implemented the back-end application.	66
4.4	CEDL editor of the first, deprecated version of Hasselt UIMS, presented in [Cuenca 14b]. Each composite event could be bound with one event-handling callback only. Noticed that this editor was much simpler than the current one, shown in Figure 4.3.	70
5.1	Implementing the <i>put-that-there</i> command.	81
5.2	(a) Event composed of an arbitrary number of flick downs. (b) FSA-based representation of event <i>flickdown</i> (Equation 5.4), which describes one of the strokes shown in (a). (c) FSA-based representation of event <i>repeflicks</i> (Equation 5.6, which describes the whole multi-stroke gesture shown in (a)). (d) FSA-based representation of the multi-touch gesture <i>zoom</i> (Equation 5.8)	84
5.3	(a) The end user must put his left hand in front to start drawing a digit. (b) The points covered by his stroke, and their timestamps are accumulated in <i>x[]</i> and <i>y[]</i> , and <i>t[]</i> respectively. (c) Common structure for the events <i>lhandfront</i> and <i>lhandback</i> used in the definition of the event <i>digit</i>	87

-
- 5.4 Description of a prototype that supervises the end user during his training session. A demo of this system is shown in <https://youtu.be/rKBNi4VEaKM> 89
- 5.5 (a) To avoid interleavings of *mouse.move* events during a mouse click, *perfectClick* must only be triggered when *x* is null. (b) Any system response annotated in *node(2)* will be conveyed if the command for deletion is detected either via speech or via keyboard. (c) Any system response annotated in both unstable nodes, *node(2)* and *node(3)*, will be conveyed only once even if the command for deletion arrives redundantly via voice and via keyboard. 96
- 98figure.caption.63
- 6.1 On the left side of the diagram, one can see the different levels of abstraction used by Dumas et al. [Dumas 10]. On the right side, one can see how our proposed tool follows the same framework: our HMD2L is at the dialog level whereas the CEDL and SRDL are at the events level. 105
- 6.2 HMD2L editor. A property window is displayed when programmers click on an arrow of the user-defined FSM. The drop-down list contains all composite events previously defined with CEDL and SRDL. The textboxes can be optionally written with a guard condition (top) or with a list of assignments statements (bottom). 106
- 6.3 At the top, one can see Couch Potato operating in its three different contexts-of-use: (a) Initial mode, (b) Selection mode, and (c) Playback mode. These contexts-of-use as well as their interrelations are modeled in (d). 110
- 6.4 FSA-based representation of the events (a) *left2right* to be triggered when the user flicks to the right, (b) *rhandfront*, to be triggered when the user put his right hand forward, and (c) *playvideo* to be triggered when (a) and (b) co-occur, in which case Couch Potato will enter into playback mode. . . . 111
- 6.5 Coach Potato setup. The Z-axis of the Kinect's coordinate space extends in the direction in which the sensor points. When the user extends his right hand, the z-coordinates of the head and the hand differs in at least 35 cm. 112

6.6	FSA-based representation of composite events (a) <i>digit</i> , triggered when a digit has been drawn, and (b) <i>searchByTouch</i> , triggered when several digits, whose names are accumulated in the array <i>d</i> , have been drawn.	113
6.7	FSA-based representation of the composite events (a) <i>xrighthand</i> , which filters the information provided by MS Kinect, (b) <i>fromR2L</i> , to be triggered when the user's right hand moves to the left in three consecutive frames, and (c) <i>wave</i> , to be triggered when the wave gesture occurs.	116
7.1	HMD2L model representing the dialog supported by the Put-That-There system described in Section 7.3.	137
8.1	(a) Sequence of steps followed by each participant in the user study. The first and second experiments referred to in this sequence are carried out in an analogous fashion. (b) Common flow of activities followed during the first and second experiment. Here Hasselt would mean CEDL and SRDL for the first experiment; and HMD2L for the second experiment.	146
8.2	Programming experience of the 12 participants.	147
8.3	Single ease question (SEQ) questionnaire. Each participant filled the SEQ four times –namely after using C# in experiment 1, after using Hasselt in experiment 1, after using C# in experiment 2, and after using Hasselt in experiment 2.	149
8.4	System Usability Scale (SUS) questionnaire and average scores per question obtained for Hasselt UIMS. Raw data can be seen in Appendix D.4.	150
8.5	First experiment (CEDL/SRDL versus C#). Boxplots (a) and (b) summarize the measurements for the 10 participants who succeeded with both languages. Boxplot (c) includes all 12 participants. Raw data can be seen in Appendix D.4.	152
8.6	First experiment. Q-Q normality plots for the differences in (a) completion times, (b) code testing effort, and (c) SEQ scores. Plots (a) and (b) involve data of the 10 participants who succeeded with both languages; plot (c) includes all 12 participants. Those many points falling far from the straight line, depicted in red, indicate that the normality of the pair differences cannot be guaranteed, i.e. it may not be safe to apply paired t-tests.	152

8.7	Second experiment (HMD2L versus C#). Boxplots showing the data collected from the 12 participants. Raw data can be seen in Appendix D.4.	153
8.8	Second experiment. Normal Q-Q plots for the differences in (a) completion times, (b) code testing effort, and (c) SEQ scores, calculated for the 11 participants whose data was analyzed. Those many points falling far from the straight line, depicted in red, indicate that the normality of the pair differences cannot be guaranteed, i.e. it may not be safe to apply paired t-tests. .	154
10.1	Functionalities that can be delegated to a UIMS [Cuenca 14a]. Hasselt UIMS would fit in the state-based group.	191
10.2	(a) Each user event can be bound to one event handler. (b) Each composite event, which is a combination of many events, can be bound to multiple event handlers.	195
C.1	Unambiguous specification of the gestures <i>plus</i> and <i>equal</i>	210
C.2	Unambiguous specifications of three conflicting gestures, namely vertical flick, horizontal flick, and plus symbol	212
D.1	Raw data collected in the first experiment.	230
D.2	Raw data collected in the second experiment.	231
D.3	Raw data collected from the SUS questionnaires for the 12 participants.	232

List of Tables

2.1	Multimodal interaction description languages. This table was built by merging our own observations –discussed in this chapter– with those made in [Dumas 09] and [Navarre 09].	24
2.2	Touch gesture description languages. This table was built by merging our own observations –discussed in this chapter– with those made in [Hoste 14]. In this paper, some criteria (e.g. modularization and partial feedback) had a continuous scale. In this table, the symbol ✓ was assigned when the property is present at some degree (> 0) in the language.	30
4.1	Sets of atomic events that can be detected by the input recognizers incorporated in Hasselt UIMS.	58
4.2	Atomic events and corresponding parameters.	60
4.3	Fields of the data structure carried by <i>kinect.skelpos</i> (Table 4.2).	61
4.4	Fields of the data structure carried by atomic events <i>tscreen.move</i> , <i>tscreen.down</i> , and <i>tscreen.up</i> (Table 4.2).	62
4.5	Differences between atomic events and composite events.	63
4.6	Operators supported by CEDL in increasing order of precedence	64
5.1	Available types of system responses in Hasselt UIMS	75
6.1	Differences between the FSMs generated by Hasselt UIMS from CEDL and SRDL code versus user-defined FSMs created with HMD2L.	107
7.1	Dimensions used for comparing Hasselt with equivalent event-callback code based on the two equivalent implementations of the multimodal interaction <i>put-that-there</i> , i.e. Algorithm 3 and code snippet 7.1.	126

- 7.2 Dimensions used for comparing Hasselt with equivalent event-callback code based on the implementation of the pinch gesture for resizing screen content. 132
- 7.3 Dimensions used for comparing Hasselt with equivalent event-callback code based on the implementation of the dialog-based Put-That-There system. 138

Chapter 1

Introduction

Traditional WIMP (Windows, Icons, Menus, Pointers) systems present graphical user interfaces (GUIs) containing sets of predefined components that users manipulate in order to activate the system’s functionality. Pressing a button, dragging the scrollbar’s thumb, and selecting a menu have been common ways of interacting with WIMP systems for decades.

Research on WIMP systems started at the Stanford Research Institute, Xerox Palo Alto Research Center (PARC), and MIT in the 1970s [Myers 00]. Some years later, in 1984, the Apple Macintosh came out with three revolutionary applications –namely Finder, MacPaint, and MacWrite– that popularized WIMP systems [Beaudouin-Lafon 94]. Since then, the mouse-and-keyboard interactions supported by WIMP systems have been widely adopted with only small variations, and a relatively slow growth of new techniques [Myers 00].

Multimodal systems aim to extend the interaction styles supported by WIMP systems. One reason for this enhancement is the appearance of technology –e.g. palmtops, digital audio players, e-book readers, and smartphones– over which the use of GUIs is cumbersome or tiresome [Sturm 02]. Besides, the emergence of novel display technology such as virtual reality and wearable computers can help break the information bottleneck caused by the mouse and keyboard [Sharma 98]. Last but not least, the quest of creating systems that facilitate communicating in a natural way is a constant motivation towards the development of multimodal systems [Sharma 98].

Multimodal systems can be commanded through the coordinated use of multiple input modalities, e.g. touch, speech, gaze, facial expressions, and body gestures. Because of their capability to integrate several modalities, these

systems can support a human-machine communication that is robust (e.g. spoken words can affirm gestural commands), accurate (e.g. lip movements can disambiguate noisy speech), flexible (e.g. users can choose between touch gestures or voice commands), and natural (e.g. users can now communicate through multiple senses). Therefore, multimodal systems have the potential to be used by a broader spectrum of everyday people and to accommodate more adverse usage conditions than in the past [Oviatt 03].

The well-known Put-That-There system described in the seminal paper of Richard Bolt in 1980 [Bolt 80] is generally considered as the first multimodal interactive system. This system allowed users to manipulate a set of virtual objects around a large-screen display surface through the concerted use of speech inputs and pointing gestures. For instance, a user can point to some spot on the large screen and say ‘*create a circle there*’ in order to make a new circle appears on the indicated spot. Because voice can be augmented with simultaneous pointing, the free usage of pronouns becomes possible, with a corresponding gain in naturalness. Conversely, gesture aided by voice gains precision in its power to reference. Since that date, research on multimodal systems was sparsely carried out until the late 1990s [Lalanne 09] when automatic gesture recognition, analysis of facial expressions, eye tracking, force sensing, or electroencephalography started to be considered as potential modalities for HCI [Sharma 98].

1.1 Motivation

Whereas the functionality of a WIMP system is invoked in reaction to a limited repertoire of actions on a standard set of widgets (e.g. clicking on buttons and menus), the functionality of a multimodal system can be invoked through arbitrarily complex series of coordinated actions (e.g. speaking, pointing, and gazing). The more complex interactions supported by multimodal systems come with a cost that programmers have to pay.

When using an event language to implement a WIMP interaction, by which a windows form is closed in response to a mouse click on the button ‘*Close*’, for instance, a function for closing the form has to be implemented and declared as the event handler of the event *click-on-Close*, which can be detected by the event-driven framework. In this way, every time the event *click-on-Close* occurs, the function for closing the form will be called automatically, thus making the system behaves properly. On the other hand, implementing the aforementioned multimodal interaction, by which the system creates a circle in response to a speech-and-pointing command, is not as straightforward. The

function for creating circles cannot be declared as the event handler of the event *create-circle*, simply because, unlike clicks on standard widgets, the event *create-circle* does not exist in the repertoire of events recognizable by event-driven frameworks. Every time the user speaks and points in order to create a circle, the event driven-framework will detect a sequence of speech inputs and pointing gestures, but it will never “know” that there is a relation among these events, nor when the event sequence is going to finish. Thus, programmers have no other choice but implementing two event handlers (to handle speech inputs and pointing gestures, separately) from which the user’s intention of creating a circle has to be deduced, and the function for creating circles has to be manually called at the right time.

In general, when implementing multimodal interactions with event languages, programmers have to write code for tracking sequences of related events, and for launching application-specific functions (e.g. for creating a circle) in a timely manner. This complex code involves several state variables that must be updated in different event handlers. By interrogating those state variables, the system can always decode the current state of the event sequence, which is crucial to respond to the user in the right time. Since similar code has to be added up for every multimodal interaction that the system has to support, the source code of a multimodal system gets rapidly infested with a multitude of state variables that are updated here and there, which end up being a serious challenge for programmers to understand and to maintain.

As will be proven in this thesis, the maintenance of state variables, or more generally, the task of tracking event sequences, can be automated by changing the way in which events are bound to event handlers. Mainstream event languages, such as Java or C#, restrict programmers to bind one event to one event handler only, that is, to instruct the system in the following manner: *when this event occurs* \rightarrow *call this function*. Unlike mainstream event languages, the language proposed in this thesis enables programmers to “say”: *when this sequence of events occurs* \rightarrow *call this function*. In this way, the responsibility of tracking event sequences and therewith, the task of maintaining state variables, is put on the underlying framework, in benefit of programmers. The user-defined event sequences that can be declared with our language will be referred to as **composite events**, which is the core concept of the present work.

We are well aware that multimodal interactions can also involve partial feedback, passive modalities, user errors, ambiguous inputs, and, in general, be much more complex than the interaction described herein, but our language is much more powerful as well. For the moment, we will just keep with this

introductory definition of composite events and one of the challenges to be met, i.e. automating the tracking of user-defined event sequences. In the remainder of the thesis, more complex interactions will be addressed as more new features of the proposed language will be exhibited.

1.2 Research goals

The overall research goal of this research is to identify the benefits and limitations brought about by a composite event-based language when it comes to describe multimodal interactions. In turn, this goal was divided into the following sub goals.

- G1** To create a language with notations for defining composite events and for binding these events to event-handling callbacks. This goal requires investigating what nature (e.g. textual or visual), what paradigm (e.g. event-driven or logic-based), and what notations are more convenient for a language that intends to be adopted by users and with enough expressiveness to cover a variety of interactions. The interactions described in the language must be tested in a runtime environment.
- G2** To evaluate the advantages and limitations brought about by a composite event-driven language when it comes to describe multimodal interactions. The evaluation will include static methods (e.g. code inspection) and dynamic methods (e.g. user studies).

The language and tooling referred to in the goals *G1* are described in Part I of this thesis. The evaluation of the proposed language, referred to in *G2*, is exposed in Part II.

1.3 Research approach

We believe that the hard-to-read “callback soup” resulting from implementing multimodal interactions with existing event languages can be simplified if these languages would empower programmers to bind composite events to event-handling callbacks.

In our vision, a composite event-based language must include general-purpose constructs (sequence, selection, iteration) as well as notations for composite event binding. Programmers must be able to declare both interaction code and application code with this language, as with any mainstream event

language. The difference is that, whereas mainstream event languages restrict programmers to bind one event to one event handler, our envisioned language must allow binding user-defined event sequences, i.e. composite events, to one or more event handlers that, at runtime, are to be called automatically in the right moment of the interaction. The envisioned language must be supported by an Integrated Development Environment (IDE) including all the comprehensive facilities that programmers need to develop software, e.g. code editors, interface builders, compilers, runtime environment, and debugging tools. In particular, the compiler must be able to transform any program declared with the composite event-based language into an autonomous EXE file. The scope of this envisioned solution had to be narrowed for the present PhD project.

This PhD project implemented the notations for describing multimodal interactions; the application code has to be written externally with a general-purpose programming language. Concretely, with Hasselt, the proposed language, programmers can describe interactions by declaring composite events and by binding these events to event-handling callbacks, which are externally-defined C# methods. At runtime, the event-handling callbacks are launched automatically in response to the (partial) detection of their associated composite event. Hasselt is supported by a User Interface Management System (UIMS) (see Section 2.1.4) that includes the code editors, runtime environment, and debugging tools required to write, run, and evaluate Hasselt specifications, respectively.

Hasselt simplifies the implementation of multimodal interactions with respect to event languages. Among other reasons this is because, with Hasselt, programmers only have to ‘declare’ composite events, not to ‘implement’ a mechanism for detecting them, as it would be the case when using event languages. Contrary to what is expected from a full-fledge IDE, Hasselt UIMS cannot deliver the final version of the intended system; it cannot merge the interaction code, written with Hasselt, with the application code, externally-defined with C#, into an autonomous executable file. Despite this, Hasselt UIMS is still well-suited for rapid prototyping. Programmers can quickly explore a wide variety of multimodal interactions by defining and redefining composite events while the application-specific code remains safe and unaltered in a canned, externally developed application. This UIMS-as-a-rapid-prototyping-tool approach has been used for almost all multimodal interaction description languages and gesture description languages to be studied in the Related Work chapter.

1.4 Contributions

The main contributions of this thesis are:

Hasselt, **a family of declarative languages** for describing multimodal interactions in terms of composite events. The relations between the series of coordinated actions performed by the end user and the responses conveyed by the multimodal system throughout the human-machine interaction are implemented by declaring composite events, each of which can be bound to one or more event-handling callbacks. The programming environment that supports our language has also introduced novelties in tooling and algorithms, as will be discussed in Section 9.4.

The identification of **two new benchmark problems for touch interactions**: the recursive version of the selection ambiguity problem and the dual-faced gesture problem. Both problems as well as their potential solutions will be discussed in Section 9.3. These problems can serve as a challenge for future specification languages and as a guidance for language developers.

An **empirical study** conducted to compare the programming efficiency of Hasselt versus a mainstream event language when it comes to modify multimodal interactions. By concerted use of observations, standardized questionnaires, and interviews, we managed to measure the completion rates, completion time, code testing effort, and perceived difficulty of the programming tasks, as will be shown in Chapter 8.

1.5 Supporting publications

The present work has been disseminated in several papers that have been presented in both conferences and journals in the domains of Human-Computer Interaction (HCI) and Software Engineering.

The CoGenIVE Concept Revisited: A Toolkit for Prototyping Multimodal Systems. *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS'13)*. Cuenca, Fredy [Cuenca 13a]. This doctoral consortium paper proposes that a language for describing multimodal systems must include notations for declaring compositions of events, notations for declaring with system responses, high-level notations for specifying human-machine dialogs, and notations for combining different types of

system outputs. It was written in the first year of the PhD, before Hasselt started to be implemented, and it was used as a guidance for its development.

Assessing the support provided by a toolkit for rapid prototyping of multimodal systems. *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems (EICS'13)*. Cuenca, Fredy and Van den Bergh, Jan and Luyten, Kris and Coninx, Karin [Cuenca 13b]. This paper compares the notations of three different multimodal interaction description languages (MIDLs) for the case of the interaction *put-that-there*. It tries to make readers able to distinguish what parts of the intended multimodal system need to be described in the MIDLs and what parts need to be implemented with general-purpose code.

A Domain-Specific Textual Language for Rapid Prototyping of Multimodal Interactive Systems. *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems (EICS'14)*. Cuenca, Fredy and Van den Bergh, Jan and Luyten, Kris and Coninx, Karin [Cuenca 14b]. This paper presents both the Composite Event Definition Language, the first member of the Hasselt family, as well as the initial version of Hasselt UIMS, which was therein referred to as ‘the supporting toolkit of Hasselt’. The paper illustrates multiple features of CEDL (e.g. event operators, arrays, and time-out events) and the supporting tool (e.g. code editors and debugging tools). It also shows an algorithm that transforms composite event descriptions into semantically-equivalent finite state automata.

Graphical Toolkits for Rapid Prototyping of Multimodal Systems: A Survey. *Interacting with Computers (IwC)*. Cuenca, Fredy and Coninx, Karin and Luyten, Kris and Vanacken, Davy [Cuenca 14a]. This article studies several tools for prototyping multimodal systems. It proposes a classification of tools based on the savings that programmers can achieved when creating a multimodal system with these tools. Programmers can be saved from configuring input recognizers, from configuring output synthesizers, from fusing inputs, from managing the human-machine dialog, and/or for conveying outputs in a coordinated manner. We noticed that all the studied tools fall into three categories –namely, flow-based, state-based, and token-based– that give programmers a quick idea of how much work can be saved when using each tool. As commented in the article, the term ‘composite event’ was found implicit in many of the languages underlying the studied tools.

Hasselt UIMS: a tool for describing multimodal interactions with composite events. *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS'15)*. Cuenca, Fredy and Van den Bergh, Jan and Luyten, Kris and Coninx, Karin [Cuenca 15b]. A summary of the three members of the Hasselt family is presented in this demo paper. The paper uses a running example to explain how to combine the Composite Event Definition Language (CEDL), the System Response Definition Language (SRDL), and the Human-Machine Dialog Definition Language (HMD2L). The example makes it clear that, with Hasselt, the interaction models, created with CEDL and SRDL, are separated from the dialog model, depicted with HMD2L.

Empirical Study: Comparing Hasselt with C to Describe Multimodal Dialogs. *Proceedings of the First International Workshop on Human Factors in Modeling (HuFaMo 2015)*. Cuenca, Fredy and Van den Bergh, Jan and Luyten, Kris and Coninx, Karin [Cuenca 15a]. A within-subjects experiment, in which twelve participants are asked to perform equivalent modifications to a dialog model described with both Hasselt HMD2L and C#, is presented this paper. No statistically significant results were obtained when comparing the completion times or code testing effort involved with each language. But, it was proven that the perceived difficulty of the modelling task was easier when using HMD2L than when using C#.

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 sixth workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2015)*. Cuenca, Fredy and Van den Bergh, Jan and Luyten, Kris and Coninx, Karin [Cuenca 15c]. A within-subjects experiment, in which twelve participants are asked to perform equivalent modifications to an interaction model described with both Hasselt CEDL/SRDL and C#, is presented this paper. The results showed that Hasselt led to higher completion rates, lower completion times, and less code testing. Additionally, Hasselt was perceived as easier to use according to the results obtained from subjective questionnaires. The threats that jeopardize the validity of the experiment are openly discussed and some lessons for the future are commented.

1.6 Thesis outline

Chapter 2. Definitions and Related Work This chapter presents the core concepts that are going to be mentioned intensively throughout the thesis. It also describes other UIMSs and domain-specific languages that inspired our work. Some languages can describe multimodal interactions whereas others are specialized in multitouch interaction.

PART I.

The first part describes Hasselt and its supporting tool, Hasselt UIMS. Hasselt is a family of languages intended to describe a wide variety of multimodal interactions in a simpler, faster way than when using event languages. It is composed of three domain-specific languages, each of which is described in a separate chapter.

Chapter 3. Hasselt UIMS, a composite event-based tool. Here we describe the steps required to create a project with Hasselt UIMS. A project typically includes an interaction model and externally defined back-end applications. The assortment of internal components that Hasselt UIMS invokes in order to support the whole lifecycle of a Hasselt program, from editing to execution, are discussed in this chapter.

Chapter 4. CEDL: Composing user events. This section describes the Composite Event Definition Language (CEDL), which provides the notations for combining a series of built-in user events (e.g. `mouse.down`, `kinect.rhand`) in a declarative and compositional manner.

Chapter 5. SRDL: Responding to composite events. This section presents the System Response Definition Language (SRDL), which enables programmers to declare spatial, temporal, and semantic constraints between the constituents of a composite event previously declared with CEDL. SRDL also allows the possibility to bind one composite event with multiple event handlers that may be triggered at different stages of the interaction.

Chapter 6. HMD2L: Separating events from dialog model. The Human-Machine Dialog Definition Language (HMD2L) provides visual notations to specify dialog models. This language is not mandatory when creating a Hasselt project, but it may be convenient to describe systems whose responses vary depending on the context-of-use.

PART II

The second part of this thesis describes the evaluation of Hasselt and Hasselt UIMS. The evaluation was performed by means of code inspection and user studies.

Chapter 7. Code comparison of two different paradigms. This chapter presents three comparisons of equivalent source codes written with both Hasselt and an event language. The source codes to be compared refer to one multimodal interaction, one multitouch interaction, and one high-level human-machine dialog.

Chapter 8. User study. This chapter reports the results of a user study involving a group of C# programmers that was asked to perform equivalent modifications with both Hasselt and C#. The study includes two within-subjects experiments that compare the efficiency of the two languages when modifying interaction models and dialog models.

PART III

This final part describes the contributions, limitations, conclusions, and implications of the presented research.

Chapter 9. Discussion. This chapter discusses the design decisions behind Hasselt, identifies the contributions of the research, and lists the limitations of our particular implementation.

Chapter 10. Conclusions and Future Work. This thesis finishes describing the conclusions and identifying potential implications of the concept of composite events in the design of future event languages.

1.7 Summary

The present chapter has highlighted that unlike WIMP systems, multimodal systems need to identify the interaction state in order to be able to respond to the end user in the right moment of the interaction. When using event languages, this functionality has to be manually implemented by maintaining a multitude of state variables across several event handlers, which complicates the work of programmers. As mentioned above, the need for manually tracking event sequences is one consequence of a fundamental limitation of event

languages: they restrict programmers to bind one event to one event handler. Hasselt, our proposed language generalizes this one-to-one mapping of events to event handlers by allowing programmers to bind many event handlers per composite event, which is, in turn, a user-defined combination of many related events. The core of this thesis is the presentation and evaluation of the Hasselt, but in the meantime, the next chapter will introduce other tools and approaches aimed at simplifying the creation of multimodal interactions.

Chapter 2

Background and Related Work

The present chapter starts by describing the terminology that is going to be used extensively throughout the remainder of this thesis. It then describes several languages intended for prototyping multimodal interactions and multitouch gestures. Some of these languages were specifically designed for describing multimodal and/or multitouch interactions whereas others were built on top of existing general-purpose models or logic-based languages.

2.1 Background

2.1.1 Interaction styles: multimodal, multitouch, cross-device

An interaction style is defined as the user's perception of a dialogue with a computer [Hartson 88]. Millions of people around the globe experience WIMP interactions in a regular basis when using the mouse and the keyboard to command desktop applications. But aside from this popular way of interaction, newer interaction styles are appearing as technology evolves. Below we describe the interactions styles that can be (partially) supported by the prototypes created with Hasselt UIMS.

Multimodal interaction

Multimodal interaction is part of everyday human discourse: We speak, move, gesture, and shift our gaze in an effective flow of communication. In stark contrast to human experience, human-computer interaction (HCI) has historically

been focused on unimodal communication (i.e. information communicated through a single mode or channel, such as a mouse or a keyboard) [Turk 14]. Multimodal systems aimed at putting the natural human behaviors in the center of HCI [Obrenovic 04]; these systems represent a research-level paradigm shift away from WIMP interfaces toward providing users with greater expressive power, naturalness, flexibility, and portability [Oviatt 99]. Among other conferences, the International Conference on Multimodal Interfaces (ICMI)¹ has become the premier venue for research in multimodal interaction. In recent years ICMI also merged with a European-focused workshop on machine learning and multimodal interaction (MLMI), expanding its focus and enlarging its community [Turk 14].

As mentioned in the previous chapter, the history of multimodal systems traces back to the Put-That-There system [Bolt 80]. Other multimodal systems such as CUBRICON [Neal 89], JEANIE [Vo 96], QuickSet [Cohen 97], and SmartKom [Wahlster 01], which are now classical study cases within the multimodal community, enriched and expanded the speech-and-pointing interaction supported by the Put-That-There system.

Multimodal interaction offers a set of modalities (e.g. touch, speech, gaze, pen, head and body movements) that end users can combine in order to interact with the machine [Dumas 09]. There are important advantages in combining multiple inputs. First, such systems potentially can function more robustly than unimodal systems [Oviatt 99]. The use of several input modalities can be exploited to perform disambiguation, e.g. speech recognition can be improved when supported by lip movements recognition [Gibbon 12]. Furthermore, the flexibility of a multimodal interface can accommodate a wide range of users, tasks, and environments [Oviatt 99]. In noisy environments, for instance, one may prefer to command a smartphone with touch commands rather than with voice commands. Finally, multimodal systems are more suitable for controlling sophisticated multimedia output capabilities, such as virtual environments, animated characters, and the like, for which the keyboard and mouse input are relatively limited and impoverished [Oviatt 99]. On the down side, a major hindrance is perhaps still in the inadequacies of the individual modalities that are used in the multimodal interface. The performance of speech recognition is still highly context dependent, often below the desired robustness. Force sensing lacks suitable devices with desired accuracy without constraining the user. Sensing of neural information requires extensive training. These problems have restricted today's multimodal interfaces to a very narrow class of

¹<http://icmi.acm.org/2015/index.php?id=cfp>

domains where the problems can be minimized [Sharma 98].

Multitouch interaction

Although touch can be viewed as one input modality in a multimodal system, the many complex issues to be tackled when implementing multitouch interaction (e.g. ever-changing number of touches) have raised this domain to a study field in its own right. A large community of researchers has periodical meetings in conferences, such as the International Conference on Interactive Tabletops and Surfaces (ITS)², to advance the state of the art in the design, development, and use of interactive surface technologies.

The technology required for multitouch interaction has been available since the late 1970s [Schöning 10]. Touchscreens have been used for large and collaborative projective walls (e.g. Diamond Touch System [Dietz 01]), for automated teller machines (ATMs), and for kiosks in museums, airports, or hotels, where mouse-and-keyboard systems do not allow a suitably intuitive, rapid interaction. But touchscreens became popular in the field of cellular telephony. After 2007, when Apple launched the iPhone³, the number of smartphone users has grown considerably during the years. It is estimated that, in 2015, there are around 1.9 billion smartphone users, a figure that could reach 2.5 billion in 2018⁴.

Multitouch interaction involves series of movements that end users perform with their hands on a touch-sensitive surface in order to activate an application function [Cirelli 14]. This interaction style offers advantages and disadvantages. As regards the advantages, it is proven that, the fastest multitouch interaction is about twice as fast as mouse-based selection, independent of the number of targets [Kin 09]. This efficiency has been exploited in ATM kiosks and other similar scenarios, where reducing the waiting time of customers is convenient for the companies. Moreover, the fact that multiple variables can be obtained from each single touch (e.g. pressure sensitivity and angle on several axes) means that it is possible for touch systems to distinguish a wide variety of multitouch gestures [Wu 03], which may potentially enrich the interaction vocabulary of the user. On the negative side, multitouch interaction complicates the occlusion problem, as several fingers are clouding even more parts of the touchscreen than on single-touch devices. This can, however, be eased by clever interface design approaches, as Wu and Balakrishnan

²<http://www.its2015.org/>

³<http://www.apple.com/iphone/>

⁴<http://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>

confirmed [Wu 03]. Another issue is the fat finger problem which enforces designers to use interaction objects of a certain minimum size, in order to be precisely touchable by human fingers [Siek 05].

So far, we have defined and discussed some intrinsic problems of multimodal and multitouch interaction. But, of course, there are also problems related with the construction of multimodal and multitouch systems. Spano et al. identified three important problems in the engineering and development of touch interfaces [Spano 13b]. It turns out that these problems are also present in the implementation of multimodal systems, as we noticed during our research. The *event granularity problem* appears when a framework can only launch callback functions after the full detection of an event pattern, which prevents systems from providing partial feedback. The *spaghetti code problem* refers to the fact that the interaction code has to be split across multiple event handlers, which increases the mental load of programmers. The *selection ambiguity problem* appears when multiple gestures have the same starting sequence of events (e.g. when one has to write unambiguous specifications for the *plus* gesture and for the *equal* gesture, both gestures starting with a horizontal stroke). The solutions to these and other engineering problems are proposed and discussed yearly in the Symposium on Engineering Interactive Computing Systems (EICS⁵). The three fundamental problems found by Spano et al. can be tackled by Hasselt as will be discussed in Chapter 9.

Cross-device interaction

Many of the tools of HCI originate from a time when interactions involve a single user using a single device, which used to be a desktop computer. The realities of our digital lives moved on from this long ago [Rowland 15]. Today, many people uses multiple computers in a regular basis. Besides traditional desktop computers, they use computers for home entertainment, smartphones, tablets, and other special-purpose devices such as music-players or e-book readers. Computer devices even enter their clothes and bodies as biomedical or special purpose communication devices. As the number of devices grows, so does the need to exchange information and mediate interaction between them [Scharf 13]. People increasingly express their frustration about the lack of integration between devices and see the need for network standardization and connectivity [Sørensen 04]. Cross-device interaction (XDI) may help to overcome these obstacles through interconnection of different devices.

⁵<http://eics2015.org/>

Cross-device interaction requires multiple, separate but interconnected input and output devices so that users can utilize the input devices in order to manipulate content on output devices within a perceived interaction space with immediate and explicit feedback [Friedl 15].

This chapter will present several languages, most of them are specially tailored for modeling either multimodal or multitouch interaction. The next chapters will show that Hasselt can describe both multimodal and multitouch interactions as well as one specific case of cross-device interaction, namely, a mobile phone controlling the content displayed on a computer screen.

2.1.2 Important programming paradigms

In his 1978 ACM Turing Award lecture, Robert W. Floyd defined a programming paradigm as the way of conceptualizing how the tasks that are to be carried out on a computer should be structured and organized [Floyd 79].

The tools and languages to be studied herein require one to use different programming paradigms in order to create a software system. It is therefore convenient for the reader to be able to distinguish between the most important paradigms to be referred to in this thesis.

Imperative programming

With the imperative paradigm, a programmer writes code that describes in exacting detail the steps that the computer must take to accomplish the goal. The flow of control of an imperative program is determined by the program text. Given the initial input, it is possible to foresee all the instructions that will be executed until the termination of the program. The imperative paradigm is well-suited to implement transformational systems [Harel 85], i.e. those systems that can perform their intended computation without the need of being repeatedly prompted by the outside world (e.g. compilers). Imperative languages trace back to FORTRAN, developed by John Backus at IBM starting in 1954. Other examples of imperative languages include BASIC, Pascal, and C.

Event-driven programming

Event-driven programs consist of a series of fine-granularity functions intended for handling user events such as mouse clicks, keystrokes, speech inputs, and other events such as timer expirations and network connections. Event-driven

programs only gain control sporadically when events occur, thus causing the execution of their corresponding event-handling functions. Since the events occur in an unpredictable order and timing, the path through the code is likely to differ every time an event-driven program is run [Samek 03], i.e. the flow of control is driven by events. WIMP systems make exemplary event-driven systems [Samek 09]. Event languages as widely used as Visual Basic and Java appeared in 1991 and 1995 respectively.

Declarative programming

Declarative programming is a non-imperative style of programming in which programmers describe the desired results without explicitly listing commands or steps that must be performed. The declarative paradigm is used in many commercially available database query languages such as SQL and XQuery.

Modern-day languages do not subscribe to one programming paradigm; multiple programming paradigms may be embedded in one single language. For example, C# supports the imperative programming paradigm, the object-oriented programming paradigm, and the event-based programming paradigm. Java has many characteristics of the imperative programming paradigm, the object-oriented programming paradigm, the web-based programming paradigm, the concurrent programming paradigm, and the event-based programming paradigm [Bansal 13].

This chapter will present several declarative languages aimed at describing multimodal and/or multitouch interactions. The future work chapter will discuss the appearance of event languages in an epoch that was dominated by imperative languages. In that discussion, we will try to foresee, as clearly as possible, a future scenario in which event languages are replaced (extended) by a new (augmented) programming paradigm.

2.1.3 Prototyping and Rapid Prototyping

The hardest part of building a software system is deciding precisely what to build. It is difficult for clients to specify completely, precisely, and correctly the exact requirements of a modern software product before having built and tried some version of the product they are specifying. Therefore one direction to attack the complexity of a software project is the development of approaches and tools for prototyping of software systems [Brooks 87].

Software prototyping consists of creating a trial version of a system, a prototype. The purpose of prototyping is to clarify users requirements and to identify critical design considerations long before the final version is built [Gordon 95]. Prototypes typically perform the mainline tasks of the intended system, but make no attempt to handle the exceptions, respond correctly to invalid inputs, abort cleanly, etc [Brooks 87].

There are different prototyping approaches including rapid prototyping, iterative prototyping, and evolutionary prototyping [Beaudouin-Lafon 03]. These approaches can be applied for any type of system including WIMP systems, multimodal, and multitouch systems.

The proposed Hasselt UIMS as well as other similar tools to be discussed in this chapter are intended for rapid prototyping multimodal systems and multitouch systems. Rapid prototypes must be inexpensive and easy to produce, since the goal is to quickly explore a wide variety of interactions and then throw them away [Beaudouin-Lafon 03]. Even if they must be re-implemented in the final version of the system, rapid prototypes are important for detecting and fixing interaction problems in a timely manner.

A common strategy used by rapid prototyping tools is to allow programmers and designers to represent the behavior of the intended system in a much simpler way than with low-level code, with a declarative language instead of with an imperative one. The architecture supporting such a technique is called User Interface Management System (UIMS).

2.1.4 User Interface Management System (UIMS)

The term User Interface Management System (UIMS) first publicly appeared in an article entitled “A User Interface Management System” [Kasik 82].

A User Interface Management System (UIMS) is a software tool whose role is to mediate the interaction between the end user and an application. A UIMS provides declarative language(s) in which human-machine interaction can be modeled at a high level of abstraction and separately from the application semantics, which must be encoded with a general-purpose language [Beaudouin-Lafon 94] (Figure 2.1). At runtime, the UIMS generates a user interface that enables the interaction described in the high-level models [Shaer 08].

The bilingual approach followed by UIMSS –one language for specifying interaction and another for specifying application– can be traced back to the Reaction Handler [Newman 68], which is considered the first UIMS [Myers 98] –even older than the definition of UIMS itself.

Based on the classification proposed in [Beaudouin-Lafon 94], the history of UIMSs can be divided into three epochs.

The UIMSs proposed in the first epoch, which reached its apex during the middle 1980s, focused on graphical systems. Their declarative languages aimed at describing the mouse interactions required to draw geometrical figures on graphical applications. ALGAE [Flecchia 87], Sassafras UIMS [Hill 86], Squeak [Cardelli 85], Syngraph [Olsen Jr 83], MIKE [Olsen Jr 86], and University of Alberta UIMS [Green 85] can serve as examples along with the aforementioned Reaction Handler [Newman 68].

The second epoch, during the late 1980s, witnessed a set of UIMSs intended to support the development of WIMP systems. But since the description of WIMP interactions is strongly coupled to the widgets of the GUIs, these UIMSs needed to include more tool sets, e.g. interface builders, and not just language interpreters. Carnegie Mellon University proposed two prominent UIMSs: Serpent [Bass 88] and Garnet [Myers 90].

The research of the aforementioned UIMSs and their underlying languages led to very popular uses of event languages in many commercial tools, such as Microsoft's Visual Basic [Myers 00]. Using the interface builder of Visual Basic for the layout and the Visual Basic language for scripting the "glue" that holds everything together enables people who are not professional programmers to create sophisticated and useful interactive applications [Myers 00].

The first two epochs dealt with mouse-and-keyboard interactions. But the third epoch, which started with the new millennium, focuses on multimodal interactions. Our proposal, Hasselt UIMS, belongs to this epoch along with MEngine [Bourguet 03], ICon [Dragicevic 04a], OpenInterface [Lawson 09], CoGenIVE [De Boeck 07], ICO [Navarre 09], HephaisTK [Dumas 10], Squidy [König 10], and Mudra [Hoste 11], among others. New situations such as parallel inputs (i.e. several inputs issued at the same time) and passive inputs (i.e. inputs such as facial expressions, which are issued without the explicit intention of commanding the system) have to be addressed by the UIMSs of this epoch.

2.1.5 Accidental complexity and essential complexity

A software engineering problem can be divided into its accidental and essential complexity [Brooks 87]. Accidental complexity refers to the difficulties programmers face due to the choice of software engineering tools. It can be reduced by selecting or developing better tools. Essential complexity, on the other hand, relates to the intrinsic characteristics of the target problem and

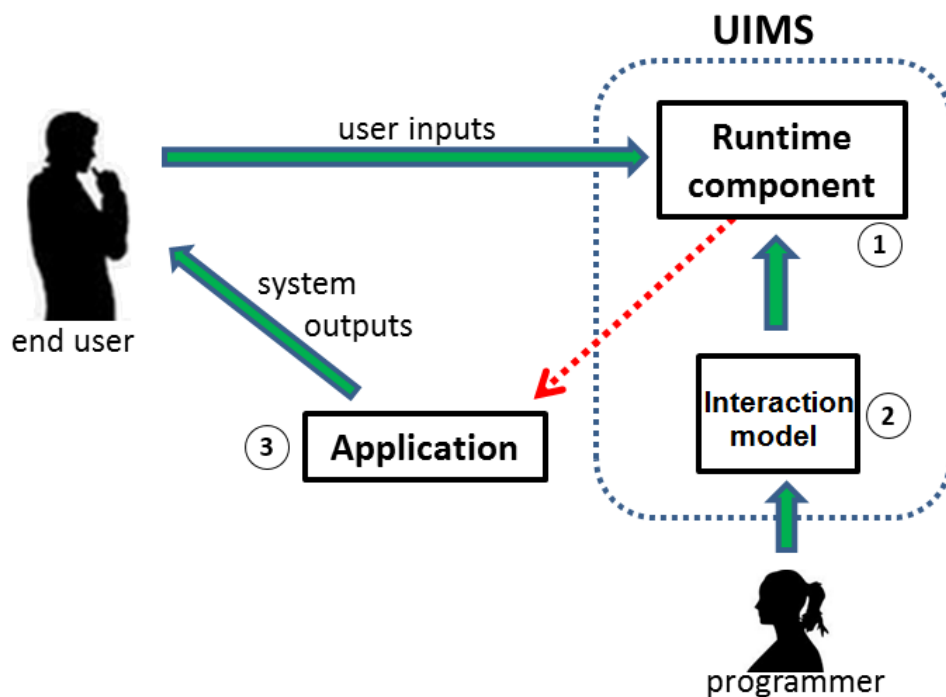


Figure 2.1: At runtime, different types of user inputs, such as mouse clicks, speech inputs, or touch inputs, are received by the runtime component ①, which is in charge of launching the methods of an externally developed application ③, according to the specifications of an interaction model ②. The externally developed application can combine different types of outputs, such as text, audio, images, animation, or video, to respond the user. The interaction model is elaborated with a declarative language that can be textual or visual; it specifies which methods of the application have to be called in response to which user inputs.

is irreducible.

When implementing multimodal systems, a problem like the presence of noisy inputs, for instance, can be considered as essential complexity. No matter how advanced the languages and frameworks one decides to use are, programmers will always have to deal with this problem since it is engendered by the sensors, not by the tools. In contrast, the need of implementing event pattern detection when creating multimodal or multitouch systems qualifies as accidental complexity. This is caused by a limitation of existing frameworks and, as commented in the previous chapter, may be reduced by creating a new type of frameworks: frameworks that permit binding user-defined event patterns with event-handling callbacks.

The purpose of Hasselt UIMS is to reduce the accidental complexity of implementing multimodal and multitouch interactions with existing event-driven frameworks.

2.1.6 Finite State Automaton and Finite State Machine

With Hasselt, several event-handling callbacks can be bound to a user-defined composite event. This implies that programmers have to be able to specify the moment in which each callback is to be launched. Hasselt uses finite state automata as *non-linear timelines* whose points may be annotated with event-handling callbacks.

A Finite State Automaton (FSA) is a logical machine composed of states and transitions. States are usually represented graphically by circles; and transitions, by arrows between states. The machine works as follows: if it is in state s_1 and input evt arrives, it goes into state s_2 as long as there is a transition labeled evt from s_1 to s_2 [Beaudouin-Lafon 94].

For the sake of precision, it is important to clarify the differences between FSA and a similar model. A Finite State Machine (FSM) is a FSA whose transitions can produce system outputs [Chen 08]. In its graphical representation, the outputs of a FSM, the same as its events names, are annotated in the arrows. In the domain of software engineering, FSMs may also have guard conditions associated to their transitions [Wagner 06].

At design time, Hasselt makes use of both FSAs and FSMs as shown in Figure 3.6. There the FSA auto-generated by Hasselt is upgraded to a FSM after annotating their nodes and links with system outputs.

2.2 Related Work

The textual notations provided by Hasselt UIMS allow describing multimodal and multitouch interactions; its high-level visual language permits describing human-machine dialogs. The present section will compare each of the three types of models supported by Hasselt with other similar models.

2.2.1 Multimodal interaction description languages

Almost all multimodal interaction description languages are visual languages whose models are variations from block diagrams, state machines, and Petri nets [Cuenca 14a].

Multimodal interaction as block diagrams

The visual languages provided by ICon [Dragicevic 04b], Squidy [König 10], and OpenInterface [Lawson 09] allow representing 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 (Figure 2.2a). 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). These tools allow programmers to connect a wide variety of input devices to an application, in a declarative, visual fashion. Applications no longer have to include code for the initialization and configuration of input hardware, since these tasks are performed internally by the block diagram-based tools.

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. These three tools do not combine data from different input sources. Therefore, the multimodal application has to store the input data coming from different modalities and identify when a meaningful event has occurred so that an adequate system response can be conveyed. In contrast, Hasselt and the tools to be discussed below are able to identify these meaningful events from user-defined declarative specifications.

	Icon	OpenInterface	Squidy	VRED	NiMMiT	ICO	HephaistTK	Mudra	Hasselt
Paradigm									
Dataflow	✓	✓	✓	✓					
Event-driven				✓	✓	✓	✓		✓
Logic-based								✓	
Type of models									
Visual	✓	✓	✓	✓	✓	✓	✓		
Textual								✓	✓
Interaction styles									
Multimodal	✓	✓	✓	✓	✓	✓	✓	✓	✓
Multitouch			✓			✓		✓	✓
Cross-device								✓	✓
About the language									
Separation dialogs/events				✓	✓		✓		✓
Time variables + delay events						✓			✓
Negation of events					✓	✓		✓	
Domain-specific notation	✓	✓	✓	✓	✓		✓		✓
About the supporting UIMS									
Generation of EXEs					✓				
Available in WWW	✓	✓	✓			✓			
Used in real-world project						✓			

Table 2.1: Multimodal interaction description languages. This table was built by merging our own observations –discussed in this chapter– with those made in [Dumas 09] and [Navarre 09].

Multimodal interaction as finite state machines

Finite state machines (FSM) have been widely used to model multimodal interaction. The nodes of the FSM represent the possible states of the multimodal system, and its arcs represent the events that cause the transitions in the system's state.

VRED [Jacob 99] was one of the first approaches targeted towards the specification of Post-WIMP interaction, which include multimodal interaction. It splits the graphical interaction specification into a data-flow component and an event component; this latter is modeled as a FSM. Although the authors acknowledged that “all continuous inputs must ultimately be quantized in order to pass them to a digital computer”, they decided to consider certain events, like the dragging of a mouse, as continuous input just because “the user does not think of generating individual ‘motion’ events, but rather of making a continuous gesture”. Our approach is different, we assume that Hasselt users are programmers, therefore they know that the system will process user actions as individual events arriving in discrete moments of time. On the one hand, VRED may be interesting for a wider audience –its potential users do not necessarily have to know what happens *‘inside the computer’*. On the other hand, Hasselt models give more control to programmers; these can see and use information that is invisible from the user's perspective. For instance, with Hasselt models, one can distinguish which of two simultaneous user inputs will be perceived first by the system and specify a distinct system response for each case, whereas, from the user's perspective, these two inputs will just arrive at ‘exactly’ the same time, which is not often the case [Oviatt 99].

In MEngine [Bourguet 02], multimodal interactions are described by depicting FSMs. Each arch of these machines can be annotated with only one event name. Each node may be annotated with a handling subroutine, which is to be launched when the node is reached. One problem of MEngine is that its models grow too quickly when involving simultaneous inputs. For instance, it is known that deictic terms can precede pointing or vice versa during speak-and-point selection; these inputs often do not co-occur [Oviatt 99]. When using MEngine, these two different orders of arrival have to be explicitly depicted by the user. Obviously, this gets more tedious if one has to describe interactions involving not only two, but three or more simultaneous inputs –in general, N inputs can arrived in $N!$ different ways. Hasselt UIMS protects its users from this state explosion through the automatic generation of state diagrams. Hasselt programmers only have to specify which inputs are to be simultaneous (by using the AND operator) and, under the hood, Hasselt UIMS

will generate a FSM that contemplates all the possible ways in which these inputs can arrive.

NiMMiT is another visual modeling language based on FSMs [De Boeck 07, De Boeck 08]. In contrast with MEngine, one can annotate several event names to one single arc of a NiMMiT model. Such arcs will be traversed only if all its associated events occur simultaneously. This implies that, unlike with MEngine, one does not have to explicitly depict all the possible orders of arrival in which the inputs can be sensed; this will be internally handled by NiMMiT in benefit of its users. To the best of our knowledge, the supporting tool of NiMMiT, called CoGenIVE, is the only tool capable of transforming interaction models into C# code. This confers CoGenIVE with a rank higher than that of a rapid prototyping tool. NiMMiT models do not have to be thrown away and re-implemented as it is the case with the other tools to be studied in this chapter, including Hasselt UIMS. With CoGenIVE, it is always possible to continue progressing towards the final version of the intended system by building up on top of the C# code obtained from NiMMiT models. On the other hand, NiMMiT has limitations too. One limitation is that its events do not include parameters, which undesirably increases the number of function calls. For instance, every time one needs to refer to the cursor position during a mouse click, a function that returns this information has to be invoked. A better approach, as implemented by Hasselt, is to include event parameters in the definitions of user events so that the values of these parameters can be automatically set, in a timely manner, and without the need of function calls. Another issue of NiMMiT is that it easily degenerates into bulky diagrams. To name just one example, a sequence of several events has to be modeled as a chain of multiple arcs –as many as events in the sequence. If we consider that each arc of a NiMMiT model must always carry a visual symbol meant to specify event-handling callbacks (even if one does not want to associate callbacks to the arc), it is easy to foresee the unnecessarily large size (in space and in number of symbols) of the resulting model. Situations like this encouraged us to rule out the option of upgrading NiMMiT, as it was the initial purpose of this PhD project. Rather, after surveying several multimodal interaction description languages [Cuenca 14a], we decided to synthesize the important features of these languages into a concise, easy-to-modify textual notation, which is what a majority of programmers are used to work with^{6,7}.

HephaisTK is a rapid prototyping tool that offers visual notations for describing multimodal interactions. In HephaisTK models, there is a clear sep-

⁶<http://spectrum.ieee.org/static/interactive-the-top-programming-languages>

⁷<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

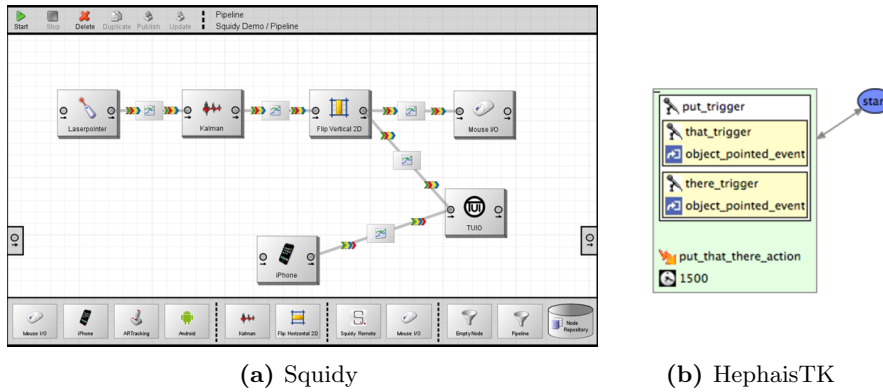


Figure 2.2: Multimodal interaction represented with a (a) Flow-based model, (b) State-based model.

aration between the specifications of events and the dialog model, which, in our opinion, enhances their readability. In HephaisTK, each arc of a FSM is annotated with a user-defined event pattern and an event-handling callback. The callbacks are to be launched when the event patterns occur, thus causing the system to switch to a new state. The straightforward way of binding event patterns to event-handling callbacks is the key idea that we used to create Hasselt. Such type of binding is something that programmers have been used to for decades: To define the human-machine interaction with a WIMP system, one has to bind events such as mouse clicks and keystrokes to event handlers. To define an event pattern, HephaisTK users have to specify the relation among its constituent events. In HephaisTK, the relations among events can be Complementary, Assigned, Redundant, or Equivalent, which are all the ways in which modalities can interact among each other, according to a well-known theoretical framework, called CARE [Coutaz 95]. One limitation of this tool is its inability to provide partial feedback. Event-handling callbacks are launched after some pattern of events has been fully detected, but cannot be launched in the middle of this process. Although, in theory, this can be fixed by splitting the event pattern and rewiring the visual model, this unnecessarily increases the size of the model and the task time. Unlike HephaisTK, Hasselt allows binding multiple event-handling callbacks to one single event pattern; this is possible thanks to its notations for specifying the moment when the callbacks have to be launched.

Multimodal interaction as Petri nets

ICO is a language intended for formal descriptions of multimodal interactive systems [Navarre 09] and it has been successfully applied in the field of safety-critical systems. By using PetShop, the tool supporting ICO, one can describe a wide variety of interactions by depicting Petri nets-based models (Figure 2.3a). Unlike other languages studied in this chapter, ICO models can be analyzed in static time, by exploiting the well-studied mathematical apparatus behind Petri nets [Murata 89]. This static analysis gives the advantage to prove properties about the intended system, before implementation commences, thus saving on testing and maintenance time. 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, specifically created to describe multimodal systems, enjoyed an enhanced domain-specificity and map closer to the multimodal domain than ICO. In Hasselt, for instance, the modalities involved in the interaction are explicitly specified. Besides, 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 12]. This efficiency is desirable in the prototyping phase, where the interaction descriptions have to go through multiple design-implement-test loops.

Multimodal interaction as a set of logic rules

Instead of providing a domain-specific visual language like the other multimodal interaction description languages studied herein, Mudra [Hoste 11] offers textual notations. When comparing different models of the interaction *put-that-there*, we noticed that the specification obtained with Mudra was more concise (in space) than other equivalent specifications obtained with different visual languages [Cuenca 14b]. The conciseness of Mudra notation is definitively beneficial for its users: the less material to be scanned, the higher the proportion of it that can be held in working memory, and the lower the disruption caused by frequent searches through the model [Green 96]. Mudra strongly influenced our decision of creating Hasselt as a textual language. Mu-

dra specifications have to be written in CLIPS⁸. Since CLIPS was specifically designed for expert systems, the language does not map as close to the multimodal domain as other domain-specific languages such as Hasselt or SMUIML (the language underlying HephaisTK). Whereas WIMP interactions have been implemented with event languages for decades, Mudra requires viewing multimodal interactions from a different perspective, by using the logic-based paradigm. In Mudra, the events are not notifications that have to be handled as they occur, as it is the case with Hasselt and mainstream event-driven languages. Rather, the events have to be seen as information that is to be 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 09]. On the other hand, using the existing CLIPS engine brings about the advantage of exploiting a language with a wide variety of notations that have been developed and evaluated through a long period of time (the first versions of CLIPS were developed in 1985) in many real-world projects⁹.

2.2.2 Gesture Description Languages

The complexity of describing multitouch interactions causes the appearance of languages and tools specially designed to prototype touch gesture definitions. As will be pointed out below, the notations of some of the abovementioned languages are rich enough to model touch interaction as well as multimodal interactions.

For Proton++ [Kin 12a], a gesture is a regular expression over the touch events *touch-down*, *touch-up*, and *touch-move*. Such regular expressions are auto-generated from a tablature, a visual notation in which users have to specify, for each finger, the sequences of events that will be triggered during the gesture. Each touch event may have attributes, which carry important information about the event, e.g. the (x-y)-coordinate of the touch point. Additionally, developers can define custom attributes. For instance, one can write the code for detecting the number of fingers on the touchscreen and use it to define a *total-fingers* attribute to be carried by *touch-move* events. The concept of attributes is similar to the concept of event parameters used in Hasselt. But in Hasselt, all event parameters are predefined, which restrains programmers. As to the handling of time, in order to specify the duration of a gesture in Proton++, one has to calculate how many *mouse-move* events

⁸<http://clipsrules.sourceforge.net/Version63.html>

⁹<http://clipsrules.sourceforge.net/OtherWeb.html>

	Proton++	Midas	ICO	GestIT	GDL (Khandkar)	Hasselt
Paradigm						
Event-driven	✓		✓	✓	✓	✓
Logic-based		✓				
Type of models						
Visual	✓		✓			
Textual		✓		✓	✓	✓
Features of the language						
Composition of gestures		✓	✓	✓	✓	✓
Modules of gestures (libraries)	✓	✓	✓	✓	✓	✓
Partial feedback	✓	✓	✓	✓		✓
Negation of events		✓	✓	✓		

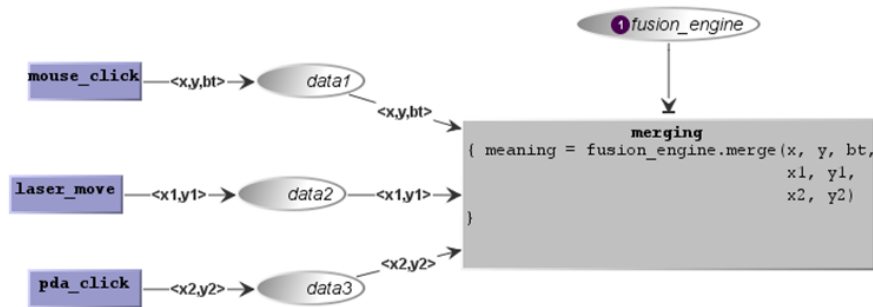
Table 2.2: Touch gesture description languages. This table was built by merging our own observations –discussed in this chapter– with those made in [Hoste 14]. In this paper, some criteria (e.g. modularization and partial feedback) had a continuous scale. In this table, the symbol ✓ was assigned when the property is present at some degree (> 0) in the language.

would fit into that gesture and use that number in the specification. Hasselt does not require manual calculations; it provides time variables and *delay* events with which a variety of time restrictions can be specified. Finally, whereas Proton++ assumes that only one gesture is performed at a time [Kin 12a], in Hasselt UIMS, one touch event can trigger multiple gestures. This feature enables users to rotate and scale simultaneously, for instance.

Midas [Scholliers 11] is a framework aimed at supporting gesture recognition. Like Hasselt UIMS, Midas supports compositionality and modularization of code. To obtain information about the GUI's objects, Hasselt programmers have to invoke a series of callbacks specifically implemented to return those values. In contrast, Midas specifications can refer directly to the GUI's objects, which are reified as *shadow facts*. Another difference is in the way to specify constraints. In Midas, the temporal and spatial relations must be specified through a set of predefined functions (e.g. *sNear*, *movingUp*, *tBefore*, etc.), whereas in Hasselt, programmers can define their own constraints by using their own user-defined variables connected to arithmetic/logical operators. None of these approaches is perfect. With the predefined functions of Midas, programmers may feel restricted at some point, whereas the definition of customized constraints allowed by Hasselt may be difficult for complex gestures. There seems to be an *optimal point* between these two approaches, but we do not pretend to identify it. Finally, unlike Hasselt and Proton, Midas does not use the event-driven paradigm (Figure 2.3b), which is commonly used to implement interactive systems. For Midas, an event is not 'something to be handled' as soon as it occurs, but a record to be stored into a database, which will be queried from time to time.

Like Hasselt, ICO [Hamon 13] is more than a multitouch interaction description language; it also supports descriptions of multimodal interactions. In the approach of ICO, each finger is seen as an input device that needs to be dynamically instantiated. The formality of ICO responds to its goal of providing complete, unambiguous descriptions of safety-critical systems. This formality brings about advantages (e.g. predictions based on static analysis) and disadvantages (low domain-specificity) as commented in Section 2.2.1. Hasselt, in contrast, is not based on a formal mathematical model and its models cannot be analyzed in design time. One design goal of Hasselt is to map close to the multimodal, multitouch domain so that programmers can simply and quickly describe and modify interactions, not being discouraged of the multiple iterations typical of the prototyping phase.

GestIT [Spano 13b] is a proof-of-concept library, which has been exploited for managing multitouch and full-body gestures. It is based on a formalism



(a) ICO

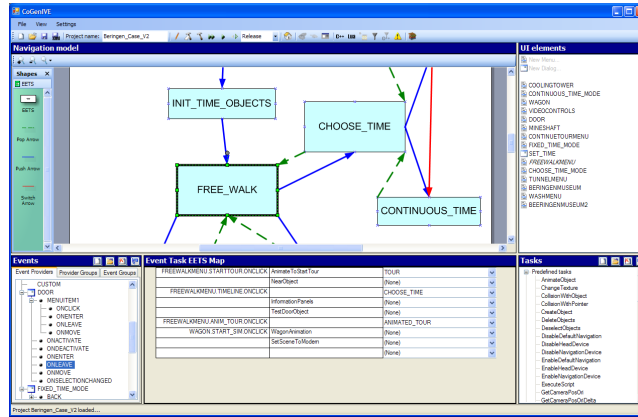
```

(defrule FlickLeft
  ?eventList[ ] <-
    (ListOf(Cursor(same:id)(within:500)(min:5)))
  (movingLeft ?eventList)
  ==>
  (assert (FlickLeft (events ?eventList))))

```

(b) Midas

Figure 2.3: (a) Petri nets-based model for multimodal interaction, (b) Logic-based code for describing a gesture.



(a) CoGenIVE

Figure 2.4: Languages for describing context-of-use-dependent interactions. CoGenIVE shows how the human-machine dialog changes; each context-of-use represents a set of interaction techniques available to the end user.

called Non Autonomous Petri Nets. With GestIT, the provision of partial feedback requires splitting a gesture into its subparts. With Hasselt, there is no need to split gesture definitions because due to the visual form of a composite event programmers can directly refer to the moment when the feedback must be issued. However, GestIT handles the selection ambiguity problem. When two gestures start with the same event, GestIT waits for the next event (i.e. lookahead) to determine the actions to be performed. Hasselt does not perform lookahead; it is the responsibility of Hasselt programmers to overcome such ambiguities by using guard conditions in the Hasselt specifications or if-else statements in the back-end application.

The Gesture Definition Language (GDL) [Khandkar 10] is intended for simple description of touch gestures that can be used across multiple hardware platforms. A touch gesture is defined as a set of rules that must be met by the raw touch data along with the value(s) to be returned when the gesture is detected. GDL allows defining multi-stroke gestures (e.g. a cross) as long as the strokes can be issued sequentially. Hasselt, in contrast, allows defining gestures involving sequential and parallel strokes. Furthermore, unlike Hasselt, GDL does not allow specifying temporal conditions.

2.2.3 Human-machine dialog modeling languages

In the visual models elaborated with CoGenIVE [De Boeck 08] and HephaisTK [Dumas 10], there is a clear distinction between the interaction descriptions and the dialog model. Whereas the interactions are described as the system responses to be given in reaction to certain user actions, the high-level dialog model specifies the relations between interactions, i.e. which interactions are available in a given system state.

CoGenIVE is not only in charge of interpreting NiMMiT, but also a high-level language that refers to NiMMiT models. In these high-level models (Figure 2.4a), each rectangle represents a Enabled Task Set (ETS) [Paterno 12], which, as the name suggests, represents all the tasks that the user can perform in a given moment. These tasks are described with the abovementioned NiMMiT notation. In order to prototype a multimodal dialog system with CoGenIVE, their users must utilize four languages: the ETS-based language, for defining the dialog; NiMMiT for describing the interactions; VRXML [Cuppens 04], for describing the presentation; and Lua¹⁰, for describing the application.

In the case of HephaisTK [Dumas 10], there is no high-level language on top of the language studied above. That does not mean that the interaction model and dialog model are not clearly distinguishable. The language provided by HephaisTK contains independent notations for declaring the human-machine dialog and for combining user events (Figure 2.2b). The interactions that the intended system can support in a given moment are represented by using different types of rectangles that can be nested in each other. The influence of these interactions on the system state can be seen from the circles and arrows of the FSM-based models.

Hasselt followed the approach of CoGenIVE. We decided to create a visual language (Chapter 6) that will be completely separated from the interaction descriptions, which will be done with textual notations (Chapter 4 and Chapter 5). This gave us the opportunity to evaluate both the visual language and the textual languages separately (Chapter 8).

2.3 Summary

This chapter provided the definition of the main concepts that will be used throughout the thesis. It then described a wide variety of UIMs intended for prototyping multimodal systems and multitouch gestures. All these UIMs

¹⁰<http://www.lua.org/>

provide declarative languages in which programmers can describe multimodal and/or multitouch interactions in a declarative manner. Some of these languages, e.g. HephaisTK and NiMMiT, were specifically designed for describing multimodal interactions and enjoy enhanced domain-specificity. Others like ICO, Mudra, and Midas are closely tied to more general languages (e.g. mathematical modeling languages like Petri nets or logic-based languages like CLIPS), which lowers their domain-specificity. Hasselt was built from scratch, and specially tailored for multimodal and multitouch systems. The following Part I of this thesis will expose the three languages comprising the Hasselt family and its supporting tool, Hasselt UIMS, each one in a dedicated chapter.

Part I

Hasselt, a family of languages

Chapter 3

Hasselt UIMS, a composite event-based tool

The purpose of this Part I is to present Hasselt UIMS along with its three underlying languages, namely Composite Event Definition Language (CEDL), System Response Definition Language (SRDL), and Human-Machine Dialog Definition Language (HMD2L). These artifacts allows automating the detection and handling of user-defined event patterns, which otherwise would have to be implemented with event languages, by maintaining a multitude of state variables across different event handlers.

Without providing specific details about the languages, the present chapter will start giving an overview of the steps required to create a software project with Hasselt UIMS. It will then describe the lifecycle of Hasselt programs, i.e. the process through which the interaction descriptions are converted into finite state machines that, at runtime, are exploited by Hasselt UIMS to track the interaction state automatically, in benefit of Hasselt programmers. The main components supporting the creation, transformation, execution, and debugging of Hasselt programs will be discussed throughout this chapter.

The details about the languages comprising the Hasselt family will be presented, each in a dedicated chapter, in the remainder of Part I.

3.1 Hasselt UIMS overview

Hasselt UIMS provides a set of languages and tools with which programmers can create a multimodal prototype, a trial version performing the mainline tasks of the envisioned system. The advantages of creating prototypes as part

of the system lifecycle were explained in Section 2.1.3. The steps required to create a prototype with Hasselt UIMS are outlined below.

3.1.1 Workflow

In order to load a running interactive system, Hasselt UIMS often requires two types of inputs: an interaction model and a set of back-end applications. These are essential components of a Hasselt project (Figure 3.1). Optionally, one can also define a dialog model on top of the interaction model.

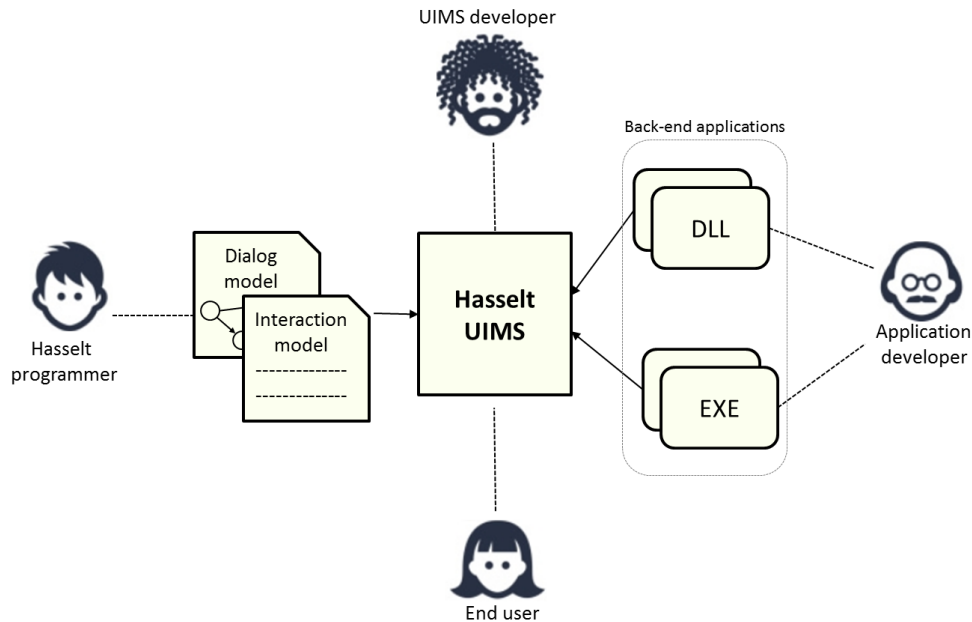


Figure 3.1: Artifacts and roles involved in a Hasselt project. At runtime, Hasselt UIMS senses and responds to the end user actions by launching the methods of the back-end applications according to the specifications of the interaction model and dialog model. Whereas the interaction model and dialog model are specified with the languages of the Hasselt family, the back-end applications (EXE applications and/or DLL libraries) are externally developed with .NET languages.

The interaction model describes the interplay between the end user and the intended prototype, the relations among the user actions and systems responses. The back-end applications implement the presentation model and the callback functions that will be launched in response to end user actions.

Whereas the interaction model can be specified with the languages of the Hasselt family, the back-end applications (EXE applications or DLL libraries) can be developed with any programming language supported by the .NET framework, e.g. C# or Visual.NET, and have to be imported into Hasselt UIMS through a designated window (Figure 3.2). The limitation to .NET languages comes from a similar limitation of the class `Assembly`¹ included in the Reflection libraries exploited by Hasselt UIMS.

With Hasselt, the interactions are described by mapping composite events to event-handling methods. For instance, two basic interactions with a touch-screen photo viewer can be described by binding the user-defined composite event *touch-flicking-left* to the method `SHOWNEXTPHOTO()`, and the user-defined speak-and-touch *move-this-here* event to the methods `HIGHLIGHTPHOTO()` and `MOVEHIGHLIGHTEDPHOTO()`. In the latter case, Hasselt programmers can even specify the moment when the methods have to be launched.

All the composite events of an interaction model do not have to be written in one single file; different sets of logically related composite events can be saved in different files to be subsequently imported into one single Hasselt project. Thus, the composite events defined for one project do not have to be redefined for another one; rather, they can be reused.

It must be highlighted that the use of back-end applications is not mandatory: some prototypes may constrain their responses to audio playback and/or synthesized voice. In these cases, Hasselt programmers can directly exploit the synthesizers incorporated into Hasselt UIMS without need of invoking externally defined .NET code.

Optionally, one can define a dialog model, a high-level model created on top of the interaction model. Without a dialog model, the user actions will be responded in the same way through the whole runtime. By declaring a dialog model, instead, one can specify the mutual influence between the contexts-of-use of the intended system and the user actions.

The languages required to specify interactions and dialogs will be studied in the next chapters, but in the meantime, readers can refer to a publicly available video² that shows how to use Hasselt UIMS to create and run a prototype that involves an interaction model, a dialog model, and one externally developed EXE application.

¹<https://msdn.microsoft.com/en-us/library/system.reflection.assembly%28v=vs.110%29.aspx>

²Hasselt UIMS workflow: <https://www.youtube.com/watch?v=jC5EuBYWwRc>

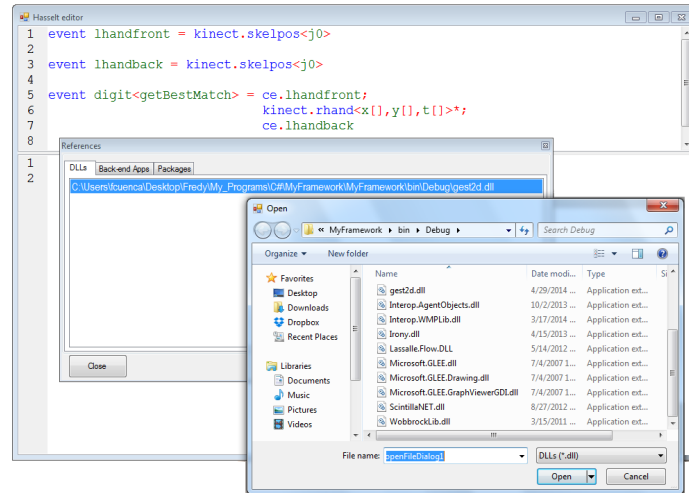


Figure 3.2: The three tabs of the References window allows declaring the (1) EXE applications, (2) DLL libraries, and (3) previously defined Hasselt programs that are to be imported into the current project. Hasselt UIMS allows extensibility and modularization of code.

3.1.2 People and roles involved

Four different classes of people are involved in a Hasselt project (Figure 3.1). The first is the person using the resulting prototype, who is called the end user. The actions that the end user performs (usually in the GUI of an EXE file) are responded according to the specifications of the interaction model and the dialog model, which are created by the user of Hasselt, the Hasselt programmer. Working with the Hasselt programmer will be the application developer. He implements the .NET applications whose functionality will be invoked at runtime. Given that the operation of Hasselt UIMS is relatively stable at the present time, the fourth participant in this role-playing, the UIMS developer, has now little or no participation in Hasselt projects. Although he has the power to change every aspect of Hasselt UIMS, including the compilers, the runtime environment, and the Hasselt languages themselves, he will only make his appearance when a structural change, like for example augmenting Hasselt UIMS with new input modalities, needs to be performed.

Although this classification discusses each role as a different person, in fact, there may be many people in each role or one person may perform multiple roles. The author of this thesis, for instance, used to play the four roles.

For the purposes of this thesis, it will be assumed that the development of

Hasselt UIMS is frozen, i.e. the UIMS developer cannot intervene anymore.

3.1.3 Startup configuration

A XML-based configuration file that is read at startup contains parameters that define the runtime behavior of Hasselt UIMS.

To Hasselt UIMS, multiple events are considered simultaneous if they are detected within a time interval. The length of this interval (in milliseconds) is one parameter of the configuration file. The handling of simultaneity is exposed in Section 5.4.1.

Another parameter configures the interruptibility of Hasselt UIMS. The end user can change his mind and interrupt a partially entered command to start issuing a new one. For these cases, one can declare a special speech input, a *reset command*, so when this input is perceived, Hasselt UIMS clears up all its variables and restarts the tracking of all the composite events. Prototypes can call roll back functions upon the detection of the reset command, as discussed in Section 5.4.2.

Finally, other parameters refer to the location of speech recognition grammar files. To use speech as an input modality, one must create two speech recognition grammar files following the W3C Speech Recognition Grammar Specification (SRGS)³. The grammar files contain the set of rules that specify the spoken words and phrases that Hasselt UIMS will recognize at runtime. The reason why Hasselt UIMS requires two grammar files is discussed in Section 5.4.4.

3.2 Lifecycle of Hasselt programs

A program has a lifecycle that includes distinct phases, starting with the editing of the code that specifies the system behavior, and extending through execution, which exhibits the specified behavior. This section will describe how Hasselt UIMS supports these different phases.

3.2.1 Design time

The design time is the phase during which the interaction model and the dialog model are specified.

For the interaction model, Hasselt UIMS provides two code editors. Both the composite events to be detected and the system responses to be conveyed

³<http://www.w3.org/TR/speech-grammar/>

throughout this process are to be specified in different code editors. In order to minimize the effort required for switching attention between two editors, we integrated both editors into one single windows form (Figure 3.3a). Both offer syntax highlighting, auto-completion popups, tooltip messages, and line numbering. The code editors are integrated with a non-editable, auto-refresh canvas that displays a finite state automaton (FSA), which is the visual representation of the composite event under definition. As will be explained below, the FSAs are automatically generated from the interaction model. The code editors were implemented with support of the text editing control ScintillaNET⁴. The FSAs embedded in the editors was depicted with support of Microsoft Automatic Graph Layout (MSAGL)⁵.

Optionally, if the system to be modeled has clearly distinguishable contexts-of-use that have a strong influence on its responses to the end user, it may be convenient to create a dialog model. Hasselt's dialog models, like the one shown in Figure 3.3b, for instance, allow specifying how the interactions alter the system context-of-use, which in turn, influences on the subsequent interactions. If the visual language is not used, Hasselt UIMS will assume that the system state is not affected by previous interactions, i.e. the same interaction has to be handled in the same way over the whole runtime. The graphical editor was implemented with support of AddFlow⁶, a powerful flowcharting/-diagramming component.

3.2.2 Compile time

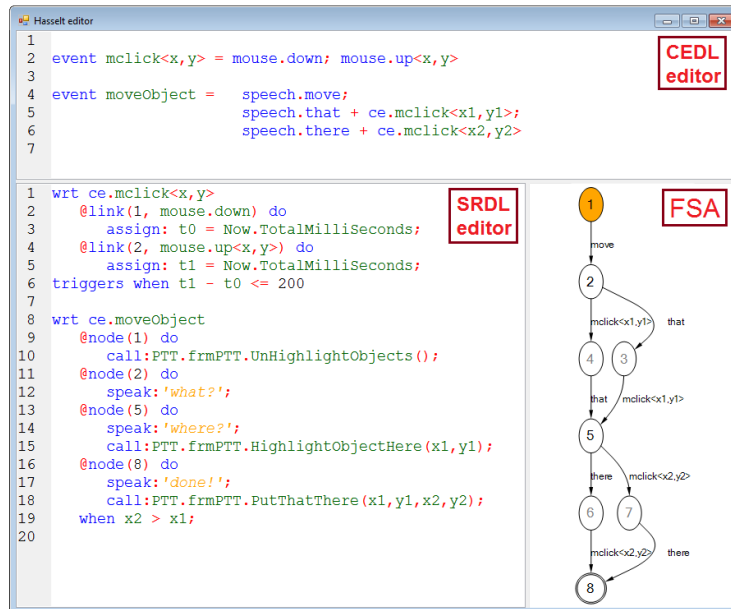
The compile time is the process that transforms a program written in one language, called the *source language*, into an equivalent program in another language, called the *object language*, which is usually the machine language of a computer, or something close to it [Bornat 07]. Although Hasselt UIMS does not generate machine code, it performs the main tasks of a compiler, namely lexing, parsing, code analysis, and generation (Figure 3.5).

In general, the lexing consists of transforming a program into a set of tokens (i.e. data structures representing the terminal symbols of a grammar). The parsing consists of verifying whether the sequence of tokens produced by the lexer satisfy the rules of a grammar, in which case, an intermediate representation, known as parse tree, is generated as a final output of the parsing process. Later, the parse tree can be used to perform code analysis,

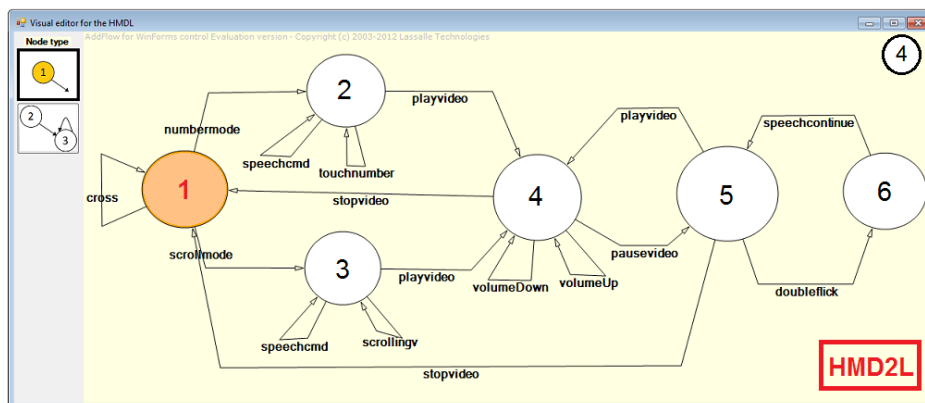
⁴<https://scintillanet.codeplex.com/>

⁵<http://research.microsoft.com/en-us/projects/msagl/>

⁶<http://www.lassalle.com/>



(a) Text editors for the CEDL and SRDL, the languages that are used to create the interaction model. Both offer syntax highlighting, auto-completion popups, and tooltip messages. The non-editable right bottom frame changes as the cursor moves through the editors; it always displays the state machine corresponding to the composite event under definition.



(b) Graphical editor for the HMD2L, the language with which the dialog model is specified. Arrows can be annotated with composite event names and guard conditions.

Figure 3.3: Editors for the three languages comprising the Hasselt family, namely Composite Event definition Language (CEDL), System response Definition Language (SRDL), and Human-Machine Dialog Definition Language (HMD2L).

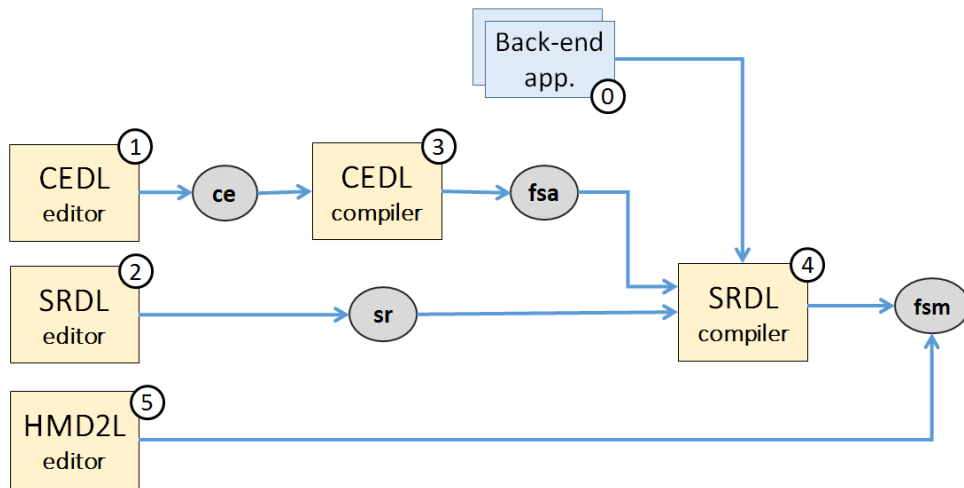


Figure 3.4: Design time and compile time architectures. A composite event ce is transformed into a finite state automaton fsa , which is to be annotated with system responses, sr , thus resulting a finite state machine, fsm . The system responses may refer to the methods of externally defined back-end applications. Both CEDL editor and SRDL editor are integrated into the same form, as shown in Figure 3.3a. The FSM created in the HMD2L editor, if exists, will be treated in the same way as the FSMs auto-generated from CEDL and SRDL at runtime.

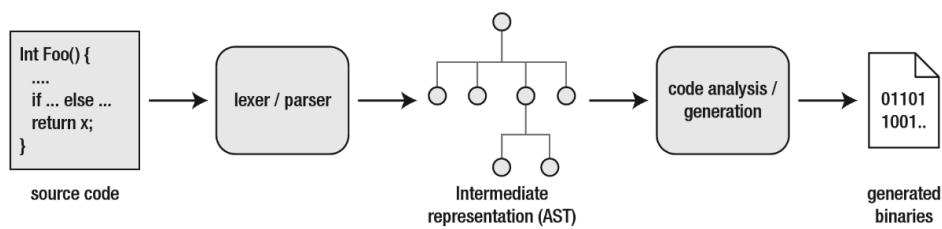


Figure 3.5: Typical flow of compile time activities [Wu 10]. The compilers incorporated in Hasselt UIMS followed the same flow except that they do not generate binary files, but finite state machines (FSMs).

such as type checking or some code optimization. In the end, the compiler generates the output binaries [Wu 10].

Each of the languages required to create the interaction model, namely CEDL and SRDL, has a designated compiler. Unlike these textual languages, the graphical language HMD2L is not compiled (Figure 3.4).

Lexical analysis and syntactic analysis

Hasselt UIMS exploits a third party component that is in charge of lexing and parsing both CEDL and SRDL code. Thanks to the Irony library⁷, we did not have to implement the lexer or the parser required by a compiler; rather, we only had to specify the CEDL and SRDL grammars against which Hasselt specifications are matched.

Irony is intended for textual languages and therefore we could not use it for the visual HMD2L, which is executed directly without checking the correctness of its syntax or semantics; errors in a HMD2L model will only be detected at runtime. The user-defined visual model created with HMD2L, if exists, is treated in the same way as the FSMs auto-generated from CEDL and SRDL code (Figure 3.4, ⑤).

The description and failing line of each lexical or syntactic error detected by Irony will be displayed in an window error message.

Code analysis

The CEDL compiler (Figure 3.4, ③) performs basic code optimization and semantic analysis.

With regard to code optimization, the CEDL compiler can remove parentheses that are unnecessarily used. For instance, the parse tree obtained from the composite event $((((A; B))))|C$ can be transformed into a simpler tree, as the one that would be obtained from the equivalent composite event $(A; B)|C$. This optimization is performed by manually pruning some branches of the parse trees returned by Irony. More precisely, by pruning those long, thin branches with no ramifications so that they can be reduced to their child nodes. This optimization prevents the function `CREATEFSA` (see Algorithm 1 in Section 3.3) from going into unnecessarily deep recursion, which may damage the efficiency of the process that transforms the parse trees into FSA, a task that is critical for the operation of Hasselt UIMS.

⁷<http://irony.codeplex.com/>

As to the semantic analysis, the CEDL compiler can identify and reject syntactically correct expressions such as $A + A$, which are meaningless since an event cannot occur in simultaneous with itself.

Such errors, which are manually detected by navigating the parse trees returned by Irony, are also displayed in the error message window.

Since the function of the SRDL compiler is much simpler: to annotate an existing FSA from a specification that explicitly says what to annotate and where to do it, we did not find opportunities for code optimization or semantic analysis.

Generation

Neither the CEDL compiler nor the SRDL compiler generate output binaries, which is the common output of a compiler. Rather, the CEDL compiler generates a set of finite state automata (FSA) that are then annotated by the SRDL compiler (Figure 3.4). Both the algorithm that generates the FSA and the algorithm used for annotating the FSA with system outputs are shown in Section 3.3, at the end of this chapter.

For illustrative purposes, the main transformations undergone by one particular composite event (drag-and-drop) at compile time are shown in Figure 3.6. The transformations labelled as (1) and (2) are performed by the CEDL compiler; the transformations (3) and (4), by the SRDL compiler.

3.2.3 Runtime

Runtime means the time when the program is executed. This is the phase when the Hasselt's framework accomplishes its goal of detecting composite events and launching event-handling callbacks in response.

Upon entering runtime mode, Hasselt UIMS activates multiple software recognizers and synthesizers, and loads the back-end applications incorporated into the project.

Hasselt UIMS has input recognizers for sensing mouse gestures, keystrokes, speech inputs, touch gestures, and body movements. The mouse events and keystrokes are intercepted by Hasselt UIMS through hook procedures; speech recognition is achieved thanks to the speech recognition engine provided by the Microsoft .NET Framework; touch events are detected via TUIO⁸; and skeleton tracking is performed via Microsoft's Xbox Kinect in combination with the MS Kinect SDK. These user events, as well as other internally-generated

⁸<http://www.tuio.org/>

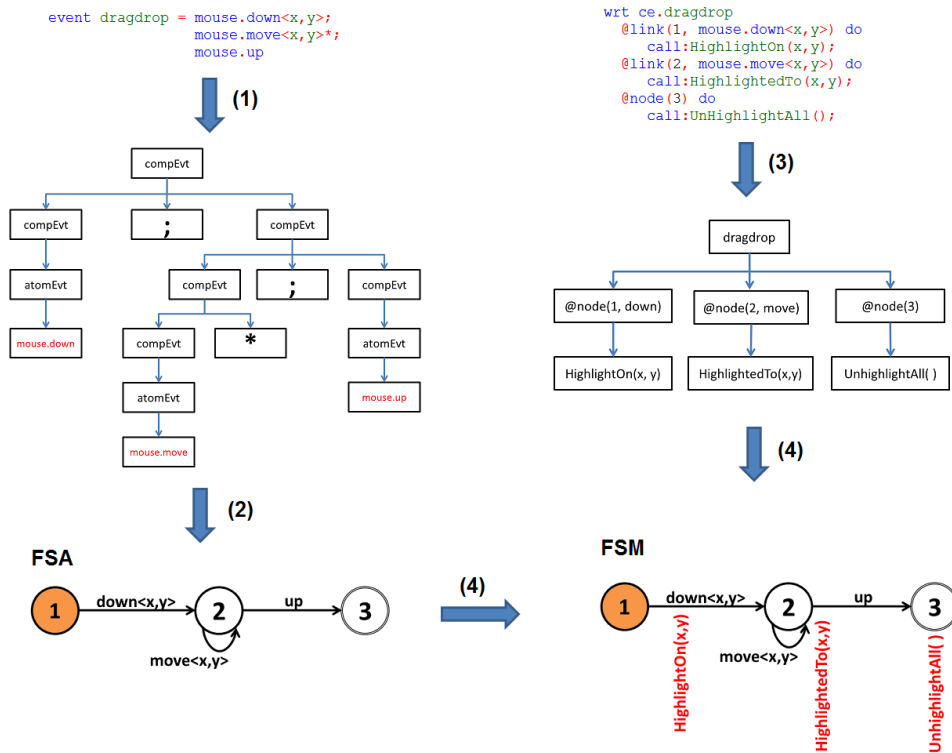


Figure 3.6: Chain of transformations undergone by the composite event drag-and-drop. (1) The CEDL code is transformed into a parse tree by a third-party component. (2) This parse tree is converted into a finite state automaton (FSA) by Algorithm 1. (3) The SRDL code is parsed, once again, by the Irony library. (4) The nodes and links of the FSA are augmented with system outputs, according to the SRDL specifications (Algorithm 2), thus resulting in a finite state machine (FSM). Algorithm 1 and Algorithm 2 are shown at the end of this chapter.

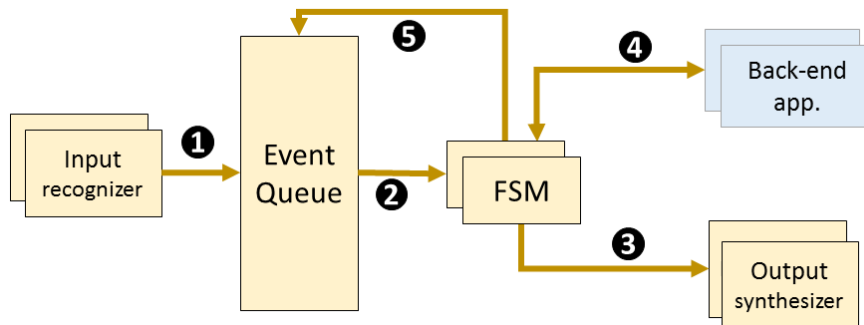


Figure 3.7: Runtime architecture. The finite state machines (FSMs) are fed with user events detected by the input recognizers and internally-generated events. Its transitions may activate the back-end applications and/or the synthesizers.

events, such as timeout events, composite event notifications, or user-defined events are sent to an event queue (Figure 3.7, ❶ and ❺).

The output synthesizers included into Hasselt UIMS enable it to synthesize human voice from text and audio from a wav file. Both synthesizers are invoked via the .NET API.

At runtime Hasselt UIMS dispatches the events of the Event Queue (Figure 3.7, ❷) to the finite state machines that were auto-generated at compile time. The state machines are served one by one, in the order in which their reciprocal composite events were defined. An event may produce state transitions in some FSMs and be ignored by others. The state transitions of a FSM may cause the activation of the UIMS synthesizers (Figure 3.7, ❸), the invocation of the callback functions included in the imported back-end applications (Figure 3.7, ❹), or the firing of user-defined (composite) events (Figure 3.7, ❺). The events are discarded immediately after being dispatched to all the FSMs.

If the Hasselt project included a dialog model, i.e. a user-defined FSM depicted with HMD2L, this FSM would be the first to receive event notifications. This preferential treatment permits to update the context-of-use in a timely manner, right upon composite events occur. Besides this, at runtime, Hasselt UIMS makes no distinction between the FSMs auto-generated from CEDL and SRDL code and the user-defined FSM elaborated with HMD2L.

Debugging tools

These tools facilitate programmers to get aware of his errors at runtime.

The variable browser (Figure 3.8, VB) shows how the values of the variables of the model change over time. By using this window, programmers can notice whether some variable is unintentionally overwritten, for instance.

The event viewer (Figure 3.8, EV) displays all the input events that are being detected by Hasselt UIMS. With this viewer, programmers can notice communication problems with the input devices, e.g. the non-presence of *kinect* events may indicate that the Kinect sensor is disconnected.

The automata view (Figure 3.8, AV) displays the composite events in their visual forms. If the project included a dialog model, this visual model would be also displayed in the automata view. In this way, one can evaluate the execution of both the interaction model and the dialog model with one single tool. As the interaction evolves, the highlighted nodes move to reflect the changes in the system state. This tool is useful to provide a quick identification of input recognition problems, e.g. when the UIMS becomes stagnant in a particular node, it is reasonable to check whether the input hardware that generates those events associated to its outgoing arcs is working correctly.

The immediate window (Figure 3.8, IW) is a scratchpad window in which C# statements involving the event parameters can be evaluated during debugging. With this window, one does not have to break execution to get feedback on how the application is performing. Moreover, unlike the other debuggers, this provides user-defined feedback, which is displayed in a separate area, so it does not interfere with outputs that the end user sees.

The initial version of Hasselt UIMS included a back-end inspector, which allowed invoking the methods of the back-end applications, at design time, so that one can have an idea of what the imported back-end applications can do even before creating the interaction model and without having to enter into runtime mode. Programmers just had to choose the back-end method they wanted to invoke and set their input parameters. This tool was useful in the early stages of Hasselt UIMS, for speech-and-mouse interactions, but it was no longer useful when we started working with touch and body movement interactions. It was difficult to manually recreate those long arrays of points that are generated when a gesture is performed. It was also difficult to guess what points would be returned by the MS Kinect sensor if one wanted to test a back-end application for one particular body pose. Thus, the development of this tool was discontinued.

Based on the experience gained after testing several interactions, we noticed that it would be convenient for future UIMS developers to consider a tool for recording and replaying interactions automatically. One potential benefit would be the reduction of the UIMS developer's mental workload: he

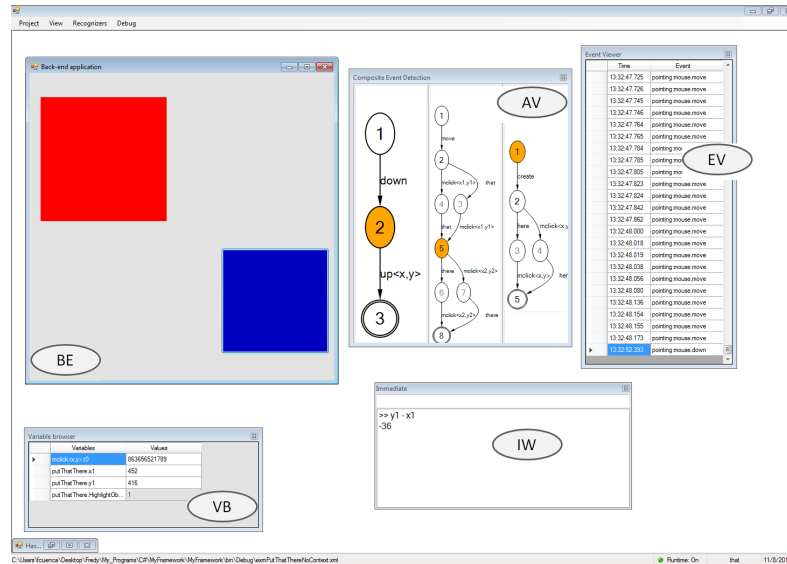


Figure 3.8: Hasselt runtime environment. The back-end application (BE) the end user is interacting with was built in C# and imported through a designated window.

could unhurriedly observe the evolution of the program state without having to interact himself.

Information such as the one displayed by the variable browser or the event viewer used to be collected in log files by other UIMSs, such as CoGenIVE [De Boeck 09], ICO [Navarre 09], and HephaisTK [Dumas 10]. However, the automata view (AV) represent a new means of debugging since the stage of multiple interactions can be followed simultaneously by observing one single viewer. This parallelism cannot be exploited by the linear, textual log files. Furthermore, to the best of our knowledge, Hasselt UIMS is the only tool that permits on-the-fly debugging through the immediate window (IW).

3.3 Algorithms used at compile time

This section provides technical descriptions of the processes used to transform a textual specification into a set of finite state machines (FSMs).

3.3.1 CEDL compiler. From composite events to FSA

The CEDL compiler generates a parse tree from each valid composite event. This tree is then transformed into a FSA by the recursive function CREATEFSA

(Algorithm 1). The base case of this function occurs when its argument is a single child node. Such nodes have trivial transformations, e.g. the smallest graph of Figure 3.9a represents the atomic event $e3$. Recursive cases involve intermixing different FSAs. Each event operator defines a different way to intermix existing FSAs.

When two composite events are linked by a ‘FOLLOWED BY’ (;) operator, the CEDL compiler concatenates its reciprocal FSAs (Figure 3.9b). When two composite events are connected by the disjunctive operator ‘OR’ (|), the compiler creates a new automaton by overlaying the initial and final states of its graphical counterparts (Figure 3.9c). Two composite events connected by the conjunctive operator ‘AND’ (+) causes the creation of a FSA whose paths between its initial and final nodes are the permutations of all the events contained in their reciprocal automata (Figure 3.9d). In this only case, the intermediate nodes are classified as special types of nodes called unstable nodes, as opposed to all other nodes, which are considered stable nodes. Unstable nodes are useful to handle parallel inputs as will be explained and illustrated in the Chapter 5. Finally, a single composite event followed by the ‘ITERATION’ (*) operator causes the alteration of its parallel FSA: the ingoing arcs of its final state will be redirected to its initial state (Figure 3.9e).

3.3.2 SRDL compiler. From FSA to FSM

Each FSA generated by the abovementioned algorithm represents a set of user inputs that the system must track. In order to have a complete interaction model, the system responses must be specified and link to the user inputs. This is done by annotating the nodes and links of the FSA according to SRDL code. Such annotations raise the transition network called FSA to the level of FSM.

The function `GENERATEFSM`, shown in Algorithm 2, has two parameters: the first represents the finite state automaton, fsa , generated from a composite event evt ; the second is the parse tree obtained from the SRDL code associated with evt , i.e. the fragment of code that starts with $wrt ce.evt$ –see SRDL syntax in Chapter 5. The annotation $output$ may contain a function name, a text message, an audio file name, or a guard condition, which are runtime are treated differently (e.g. functions are launched, text messages are synthesized, audio files are played, and guard conditions are evaluated).

Algorithm 1 Transforms a parse tree into a FSA

```

procedure CREATEFSA(node)           ▷ node is the root of a parse tree

  if isAtomic(node.children[1]) then
    return TRIVIALSD(node.children[1])

  else if isComposite(node.children[1]) & node.children[2] = '*' then
    sd1 ← createFSA(node.children[1])
    return LOOP(sd1)

  else if isComposite(node.children[1]) & node.children[2] = ';' &
isComposite(node.children[3]) then
    sd1 ← createFSA(node.children[1])
    sd2 ← createFSA(node.children[3])
    return CONCATENATE(sd1, sd2)

  else if isComposite(node.children[1]) & node.children[2] = '|' &
isComposite(node.children[3]) then
    sd1 ← createFSA(node.children[1])
    sd2 ← createFSA(node.children[3])
    return OVERLAY(sd1, sd2)

  else if isComposite(node.children[1]) & node.children[2] = '+' &
isComposite(node.children[3]) then
    sd1 ← createFSA(node.children[1])
    sd2 ← createFSA(node.children[3])
    return PERMUTE(sd1, sd2)

```

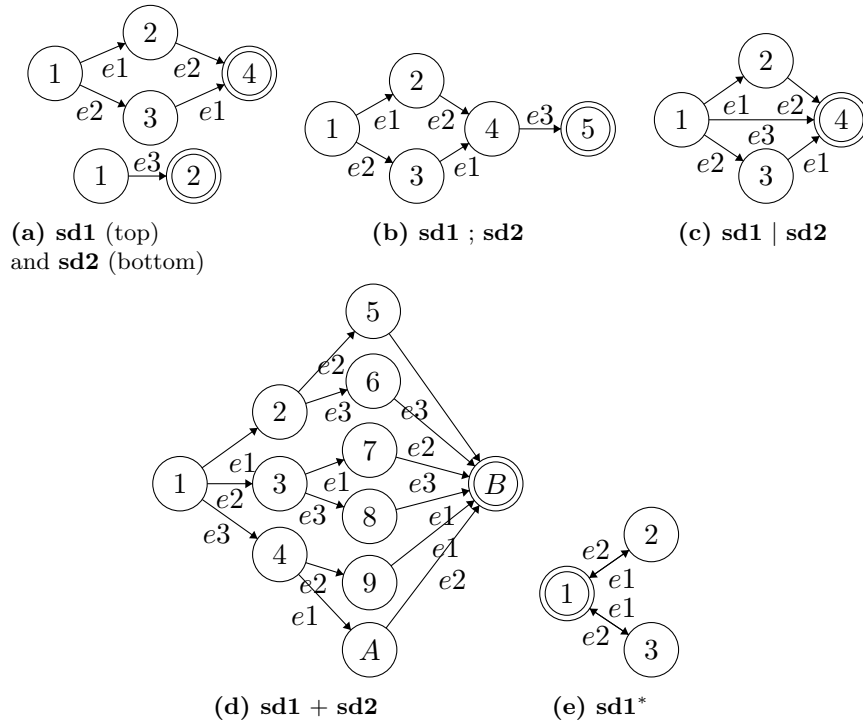


Figure 3.9: The CEDL provides four types of operators that one can use to compose events (Section 4.1.2). Each of these operators implies different operations in the internal process of transforming the CEDL code into FSA. This figure illustrates the effect of each event operator when applied to state machines $sd1$ and $sd2$, shown in (a). (b) **CONCATENATE**($sd1$, $sd2$). (c) **OVERLAY**($sd1$, $sd2$). (d) **PERMUTE**($sd1$, $sd2$) and (e) **LOOP**($sd1$).

Algorithm 2 Annotates the nodes/links of a *fsa* with the outputs specified in the parse tree *ptree* obtained from SRDL code

```

procedure GENERATEFSM(fsa, ptree)
  for each childNode c in ptree do
    if c is of type @node(nd) then
      output = c.getChildNode()
      annotate node nd of fsa with output
    else if c is of type @link(nd, evt) then
      output = c.getChildNode()
      annotate link (nd, evt) of fsa with output
  end for

```

3.4 Summary

This chapter has presented Hasselt UIMS, a set of tools aimed at creating multimodal prototypes.

The chapter started by outlining the steps required to create Hasselt projects, namely, the import of externally developed back-end applications and the elaboration of a Hasselt model (which includes an interaction model and a dialog model). Whereas the human-machine interaction can be described with Hasselt, the back-end applications (EXE applications or DLL libraries) have to be implemented with a .NET language (e.g. C# or Visual.NET).

Later, the components that support the editing, compilation, execution, and debugging of Hasselt programs were identified and discussed. Hasselt UIMS provides code editors and graphical editors with which programmers can specify multimodal (dialog) systems at design time. At compile time, the interaction specifications are transformed into a set of finite state machines (FSMs) that Hasselt UIMS uses at runtime for tracking composite events. At runtime, Hasselt UIMS activates multiple input recognizers to sense and respond to the end user actions according to the specifications made in the interaction model and the dialog model. The system responses may include launching the methods of the back-end applications or activating the output synthesizers incorporated in Hasselt UIMS. Finally, Hasselt UIMS also provides a variety of tools that facilitate the debugging of a running program. The speech vocabulary, the handling of parallel inputs, and the interruptibility of Hasselt UIMS can be gauged through a configuration file.

The next three chapters will be dedicated to explain each of the three languages comprising the Hasselt family.

Chapter 4

CEDL: Composing user events

Whereas with mainstream event languages (e.g. Java, C#), programmers are restricted to bind a set of predefined events (e.g. mouse clicks, speech inputs) to event handlers, our proposal allows binding user-defined combinations of events to event handlers. These user-defined combinations of events, hereafter called composite events, are specified with the Composite Event Definition Language (CEDL), which will be main topic of the current chapter.

Multimodal interactions are described by (1) defining composite events, and (2) binding these composite events to callback functions. The specification as well as the execution of multimodal interactions are supported by Hasselt UIMS, the proposed rapid prototyping tool.

This chapter presents the set of primitive events recognizable by the CEDL, the parameters carried by these events, and the operators used to combine them into composite events. These and other special features of the CEDL, e.g. arrays or compositionality, are illustrated by means of running examples. On top of the technical aspects, the chapter also discusses conceptual aspects such as the benefits and limitations of the CEDL.

4.1 Composite Event Definition Language (CEDL)

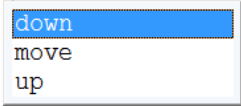
4.1.1 Atomic events

An *atomic event* is an instance of a data structure generated by an input recognizer in response to user actions. It is called atomic because, unlike composite events, it cannot be defined as a combination of other more fine-


```

moveObject = speech.move;
speech.that + mouse.down<x1,y1>;
speech.there + mouse.

```

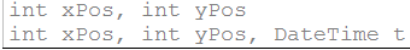


(a) Auto-completion popups

```

moveObject = speech.move;
speech.that + mouse.down<x1,y1>;
speech.there + mouse.down<

```



(b) Tooltip messages

Figure 4.1: Intellisense technology is used by the CEDL editor of Hasselt UIMS.

grained events. Like atoms, atomic events are indivisible in a certain sense. The atomic events that can be detected by the input recognizers incorporated in Hasselt UIMS are shown in Table 4.1.

Recognizer	Atomic events
.Net API	mouse.down, mouse.move, mouse.up
.Net API	keyboard.keydown, keyboard.keyup
Windows Speech API	speech.<words in grammar>, speech.any
MS Kinect SDK	kinect.skelpos, kinect.useron, kinect.useroff, kinect.rhand, kinect.lhand
TUIO	tscreen.firston, tscreen.down, tscreen.move, tscreen.up, tscreen.lastoff

Table 4.1: Sets of atomic events that can be detected by the input recognizers incorporated in Hasselt UIMS.

Atomic events must be specified in the following way: first the name of the input hardware that generates the event must be mentioned, followed by a dot, and the name of the input event. Intellisense technology helps with this task. Right after placing a dot, an auto-completion popup will display all the atomic events supported by the input hardware at issue (Figure 4.1a).

Event parameters

Many atomic events carry *event parameters*. The parameters that can accompany the atomic events provided by Hasselt UIMS are shown in Table 4.2; their syntax, scope, and datatypes are discussed below.

The parameters carried by atomic events are specified within angular brackets ($\langle \ , \ \rangle$) right after the name of an atomic event. All parameters shown in Table 4.2 do not have to be included in the specification of a given atomic event. For instance, mouse events exist without parameters (e.g. *mouse.down*), with two parameters (e.g. *mouse.down* $\langle x, y \rangle$), or with the three parameters (e.g. *mouse.down* $\langle x, y, t \rangle$). Right after opening square brackets in the CEDL editor, programmers are shown a tooltip message displaying the sets of parameters available for the atomic event at issue (Figure 4.1b). The event parameters of a given atomic event were defined based on already offered information (by the recognizers) and the needs to define several interactions.

The CEDL scopes event parameters by composite event meaning that the same parameter name can be used in different composite events without risk of conflicts.

With regard to Table 4.2 and the parameters' datatypes, the components of the (x, y) points, the ASCII code c , and the identifiers id are integers; the timestamps t are datetime variables; the text form str of the speech input is a string; the speech recognition confidence level $conf$ and the components of the (spx, spy) touch velocity are float; the variables sk and $tscr$ are data structures whose fields are shown to the programmer through auto-completion popups. As to the data structure carried by the atomic event *kinect.skelpos*, all its fields are of type double (Table 4.3). As to the data structure carried by atomic events *tscreen.move*, *tscreen.down*, and *tscreen.up*, the type of the field *numberOfFingers* is integer; all the other fields are of type double (Table 4.4). To Hasselt UIMS, the angle between two touches is the angle between the ray composed by these two touches and the X-axis of the Window coordinate system, which is used as a reference direction (Figure 4.2). The minimum, average, and maximum of these values, which are calculated only when there is more than one touch, are set to the fields *minAngleToFirst*, *avgAngleToFirst*, and *maxAngleToFirst*, respectively.

Finally, event parameters are initialized when their associated event is detected in the right moment of the interaction, e.g. the variables $x1$ and $y1$, shown in Figure 4.1b, will be set with the mouse cursor position when the mouse is pressed after the speech input move.

Atomic event	Description	Parameters
mouse.down, mouse.up mouse.move	Mouse is depressed Mouse is released Mouse is moved	(x,y) = cursor position t = timestamp
keyboard.keydown keyboard.keyup	Key is pressed Key is released	c = ASCII code of a key t = timestamp
speech.<wrđ>	Word <i>wrđ</i> was detected	conf = speech recognition confidence level
speech.any	Some word was detected	str = textual representa- tion of speech input conf = speech recognition confidence level
kinect.skelpos	A frame was fired by MS Kinect SDK	sk = 3D positions of ske- leton joints
kinect.useron	First person appears on Kinect field of view	
kinect.useroff	Last person dissapears from Kinect field of view	
kinect.rhand kinect.lhand	A frame was fired by MS Kinect SDK	(x,y) = hand's position in screen coordinates t = timestamp
tscreen.firston tscreen.lastoff	First finger touches the screen Last finger abandons the screen	(x,y) = touch position t = timestamp id = touch identifier
tscreen.down tscreen.up	Touch down Touch up	(x,y) = touch position t = timestamp id = touch identifier tscr = aggregated inform- ation of all touches
tscreen.move	x, y, spx, spy, t, id, tscr	(x,y) = touch position (spx,spy) = touch velocity t = timestamp id = touch identifier tscr = aggregated inform- ation of all touches
delay- <i>nnn</i>	A timer of <i>nnn</i> millise- conds has expired	id = timer identifier

Table 4.2: Atomic events and corresponding parameters.

Field	Content
Head.X Head.Y Head.Z	x-position of the head y-position of the head z-position of the head
ShoulderLeft.X ShoulderLeft.Y ShoulderLeft.Z	x-position of the left shoulder y-position of the left shoulder z-position of the left shoulder
ShoulderRight.X ShoulderRight.Y ShoulderRight.Z	x-position of the right shoulder y-position of the right shoulder z-position of the right shoulder
ElbowLeft.X ElbowLeft.Y ElbowLeft.Z	x-position of the left elbow y-position of the left elbow z-position of the left elbow
ElbowRight.X ElbowRight.Y ElbowRight.Z	x-position of the right elbow y-position of the right elbow z-position of the right elbow
HandLeft.X HandLeft.Y HandLeft.Z	x-position of the left hand y-position of the left hand z-position of the left hand
HandRight.X HandRight.Y HandRight.Z	x-position of the right hand y-position of the right hand z-position of the right hand

Table 4.3: Fields of the data structure carried by *kinect.skelpos* (Table 4.2).

Field	Content
numberOfFingers	number of touches
minDistanceToFirst	minimum distance of a touch with respect to the first touch
avgDistanceToFirst	average distance of a touch with respect to the first touch
maxDistanceToFirst	maximum distance of a touch with respect to the first touch
minAngleToFirst	minimum angle formed by a touch with respect to the first touch
avgAngleToFirst	average angle formed by a touch with respect to the first touch
maxAngleToFirst	maximum angle formed by a touch with respect to the first touch

Table 4.4: Fields of the data structure carried by atomic events *tscreen.move*, *tscreen.down*, and *tscreen.up* (Table 4.2).

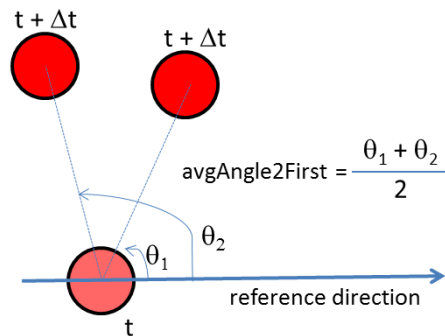


Figure 4.2: The pink-colored circle represents the first of three touches placed on a touchscreen device. The figure shows how the angles with respect to the first touch are calculated and, in particular, what the value of the field *avgAngle2First* will be for this case.

	Atomic events	Composite events
Structure	CEDL's building blocks Indivisible	Formed by many events
Event name	Predefined	User-defined
Syntax used to refer to event	Input hardware followed by event name e.g. <i>mouse.up</i>	Keyword <i>ce</i> followed by user-defined name e.g. <i>ce.myEvent</i>
Parameters	Predefined	User-defined
Programming effort for adding new events	Hard UIMS developer has to intervene	Easy Hasselt programmers can do the job

Table 4.5: Differences between atomic events and composite events.

4.1.2 Composite events

Through a set of predefined symbols, the atomic events can be combined to define high-level events, herein called composite events. In turn, composite events can be part of the definition of other more high-level composite events, i.e. composite events can be defined in a compositional fashion. A composite event serves to specify, in a declarative manner, those series of coordinated actions that the end user performs in order to accomplish a single task (e.g. to move a virtual object or to zoom in a map).

Composite event names

To declare a composite event, one has to use the keyword *event* followed by the name of the composite event, the assignment symbol (=), and a group of events connected through a set of event operators.

Optionally, one may define an arbitrary number of parameters for a composite event. For this purpose, the composite event name must be followed by the parameter names, which must be enclosed within angular brackets, e.g. $event\ myEvt(p_1, \dots, p_n) = \{combination\ of\ events\}$.

A composite event can have any arbitrary name (as long as it starts with a letter). If a composite event is to be referred to in the definition of another

more high-level composite event, the prefix *ce* will have to be added to its name (Table 4.5). This will be later illustrated with an example in Section 4.3.

Event operators

Programmers can combine (atomic or composite) events through a set of event operators that give rise to an arithmetic of events. The event operators represent temporal and semantic relations among events. Table 4.6 lists all operators supported by the CEDL.

Operator	Example	Semantics
FOLLOWED BY (;)	$A ; B$	B must occur after A
OR ()	$A B$	B or A must occur
AND (+)	$A + B$	B and A must occur within a specified timeframe
ITERATION (*)	A^*	A must occur zero or more times

Table 4.6: Operators supported by CEDL in increasing order of precedence

Explicit use of parentheses is allowed to indicate the evaluation order of the terms. For instance, the events $A;B|C$ and $(A;B)|C$ are different. The former will be triggered upon the detection of event A followed by either B or C . The latter will be triggered after the consecutive occurrence of A and B or, alternatively, upon the detection of C .

The event operators provided by the CEDL were chosen after studying the domain of active databases (Appendix B). An active database can notify an application whenever a given series of related events is recorded into a database. Since those series of related events are specified with a specialized notation, there are many languages similar to our CEDL in the field of active databases. As will be discussed, in Section 5.5.2, the CEDL supplemented by the SRDL allow defining prototypes that support Complementary, Assigned, Redundant, and Equivalent (CARE) inputs. Furthermore, the operator ITERATION (*), for which there is no equivalent in the CARE framework [Coutaz 95], proved to be convenient for specifying the arbitrarily long sequences of events, e.g. the stream of *touch-move* events generated by touch gestures. One issue with Hasselt UIMS is the difficulty for augmenting its predefined set of atomic events. This task requires changing the source code of the UIMS so that the new event as well as its new set of parameters can be recognized by the compilers and the intellisense tools incorporated in Hasselt UIMS. It would be desirable that new events can be added dynamically after importing some DLL, add-in,

or configuration file, but due to the complexity of the task, it could not be implemented in the context of this PhD.

4.2 Put-That-There in Hasselt UIMS

This section illustrates how to write and test CEDL code with the support of Hasselt UIMS. For this purpose, we will show how to implement a simplified version of the Put-That-There system mentioned in the first chapter. The system to be presented permits users to move virtual objects around a windows form by uttering the sentence ‘*put that there*’. The user must utter the pronouns ‘that’ and ‘there’ while simultaneously clicking on the target object and its intended position respectively. The steps required to obtain an executable description of this interaction are as follows.

4.2.1 Implementing back-end applications

Hasselt UIMS serves to describe multimodal interactions, whereas fine-grained .NET code is required to implement the presentation model displaying the interactive objects and the callback functions that will be launched during the described interactions.

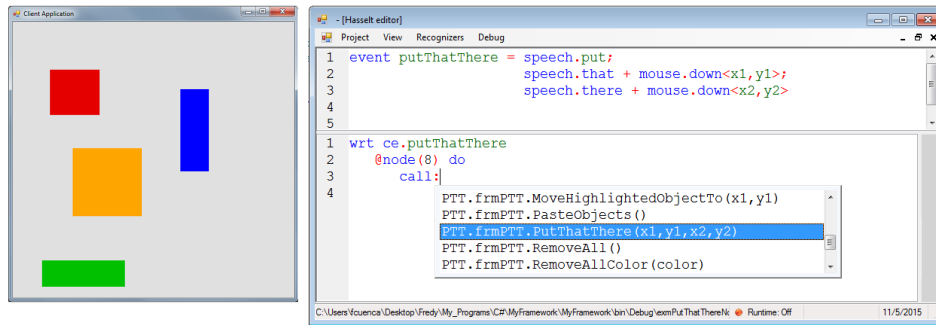
For the system under discussion, a Windows form containing several boxes was used as the presentation part of a back-end application that includes a method, called *PutThatThere*. This method is capable of moving the object placed in position (x_1, y_1) to the point (x_2, y_2) –where x_1 , y_1 , x_2 , and y_2 are the parameters of the said method. The GUI of the back-end application is shown in Figure 4.3a; its code is shown in Appendix 6.

The back-end application does not include code for handling user events, thus it is not interactive by itself. It has to be imported into Hasselt UIMS since it is Hasselt UIMS that will launch the method *PutThatThere* upon the detection of a user-defined composite event.

4.2.2 Declaring composite events

The simplest way to implement the *put-that-there* interaction requires defining the composite event *moveObject* with the following CEDL code.

$$\begin{aligned}
 \text{event } \textit{moveObject} &= \textit{speech.put} ; \\
 &\textit{speech.that} + \textit{mouse.down}\langle x_1, y_1 \rangle ; \\
 &\textit{speech.there} + \textit{mouse.down}\langle x_2, y_2 \rangle
 \end{aligned}
 \tag{4.1}$$



(a) Back-end application.

(b) Hasselt UIMS editor.

Figure 4.3: Artifacts involved in the implementation of the *put-that-there* interaction. (a) Windows-form application developed in C# without support of Hasselt UIMS. (b) Binding the composite event *moveObject* to the method *PutThatThere* implemented the back-end application.

4.2.3 Binding composite event with event handlers

The event *moveObject* must be bound with the event-callback function named *PutThatThere*, defined in the back-end application. This function must be selected from an editable drop-down list. As shown in Figure 4.3b, there is some additional code required to specify the binding. This code, which belongs to the SRDL language, will be explained in the next chapter. For this chapter, which concentrates on CEDL, it should be enough to mention that binding to externally developed functions is possible with Hasselt.

4.2.4 Testing the multimodal interactions

Through an option menu, end users can test their interaction descriptions by entering into runtime mode.

For the case under study, the event *moveObject* will be triggered upon the detection of the speech input *put* followed by the co-occurrence of the speech input *that* and a mouse click, and this, in turn, followed by the co-occurrence of the input *there* and another mouse click.

End users can notice that the speech inputs and the mouse clicks that disambiguate their meanings do not have to occur exactly at the same time; rather, Hasselt UIMS expects these inputs to arrive within a short time interval whose length is defined in a configuration file (Chapter 3). This design feature was implemented to be in line with existing empirical evidence that reveals that

multimodal signals do not co-occur temporally at all during human-computer or natural human communication [Oviatt 99].

If when trying to issue a speech input and a mouse click simultaneously, only one of these inputs is detected (e.g. due to speech recognition error), when the aforementioned time interval expires, Hasselt UIMS will forget the recognized input and give the end user a new chance to issue both inputs again. This contrast with other tools, like HephaisTK [Dumas 09] for example, that get stagnated once simultaneity fails –as we were told by B. Dumas when he visited our research lab. The technical details behind this behavior will be exposed in Section 5.4.1 with more appropriate examples.

4.3 CEDL advanced features

This section illustrates some special features of CEDL. Concretely, it will show how to capture any speech input into a string variable, how to accumulate event parameters into arrays, how to define timeout events, and how to reuse composite events to define other higher level composite events.

The back-end application shown in Figure 4.3a was extended with other functions, on top of the *PutThatThere*, and used to implement other multimodal interactions.

4.3.1 Arbitrary speech input

End users can remove all the objects of a specific color by uttering a sentence like ‘*take the green out*’. This interaction was described by binding the method *removeAllColor(string color)* –with the obvious functionality– to the composite event *removeColor*, which was defined as follows:

```
event removeColor =  speech.take;
                    speech.any{color};
                    speech.out
```

The speech input *speech.any* causes Hasselt UIMS to consume any speech input. The textual form of this input will be stored in the variable *color*. This contrast with the events *speech.take* and *speech.out*, each of them referring to one specific speech input.

The event parameter of *speech.any* can carry one single word or one full sentence. This depends on the user inputs and on the content of a speech recognition grammar file (Section 5.4.4).

The event *speech.any* can be connected with other speech inputs through the disjunctive operator (e.g. *speech.any{color}|speech.highlighted*). In these

cases, the event *speech.any* will have the lowest priority: Hasselt UIMS will first try to match an incoming speech input with every member of the conjunction, and only if all these matches fail, the speech input will be consumed as an event *speech.any*.

4.3.2 Arrays of variables

End users can remove an arbitrary number of objects from the canvas of the application. The objects to be removed must be pointed with the mouse. The input stream ‘*remove this (click) and this (click) now*’ is an example of how the said functionality can be activated.

This interaction was implemented by binding the externally-defined method *removeThisAndThis(int xs[], int ys[])* to the following composite event:

```
event removeMany =  speech.remove;
                   speech.this + mouse.click<x[ ], y[ ]>;
                   (speech.and;
                    speech.this + mouse.click<x[ ], y[ ]>)*;
                   speech.now
```

Hasselt UIMS will treat variables *x* and *y*, included in the definition of *removeMany*, as arrays because of the brackets that come upon. At runtime, every time a click is detected, the mouse coordinates are inserted at the end of the arrays *x* and *y* that will eventually be passed as parameters to the *removeThisAndThis* method.

4.3.3 Timeout events

One can define a composite event, *manyClicks* that matches single, double, and triple clicks, with each variation leading to a different computation in the back-end application. To this end, the following composite event must be defined:

```
event manyClicks =  mouse.down<xs[ ], ys[ ]>;
                   (mouse.down<xs[ ], ys[ ]>;
                    mouse.down<xs[ ], ys[ ]> | delay-250
                   ) | delay-250
```

The keyword *delay* serves to define a timeout event. The number that comes upon the hyphen (‘-’) indicates the number of milliseconds after which the timeout event is thrown. In the previous expression, no more than 250 milliseconds can elapse between two consecutive clicks when issuing double or triple clicks.

The composite event *manyClicks* was bound to the externally defined method *nClicks(int xs[], int ys[])*. Based on the size of the arrays *xs[]* and *ys[]* passed as parameters, *nClicks* implements different responses to the single, double, and triple click detection. In the future chapter, we will present simpler and more general description for this and other similar events.

4.3.4 Compositional definitions

The abovementioned *put-that-there* interaction is too rigid in the sense that it enforces end users to issue one fixed set of speech inputs. In this new example, a wider variety of speech inputs can be issued to displace one selected object, e.g. *move this here*, *displace this there*. To achieve this flexibility, the composite event *moveObject* shown below has to be bound to the method *PutThatThere(int x1, int y1, int x2, int y2)*. Whereas the *moveObject* presented here is slightly different than the one shown in Equation 4.1, the method *PutThatThere* is the same that was used before.

$$\begin{aligned}
 \text{event } verb &= \text{speech.move} \mid \text{speech.displace} \mid \text{speech.change} \\
 \text{event } demonst &= \text{speech.that} \mid \text{speech.this} \\
 \text{event } adverb &= \text{speech.here} \mid \text{speech.there} \\
 \text{event } moveObject &= ce.verb; \\
 &\quad \text{mouse.down}\langle x1, y1 \rangle + ce.demonst ; \\
 &\quad \text{mouse.down}\langle x2, y2 \rangle + ce.adverb
 \end{aligned}
 \tag{4.2}$$

The composite events *verb*, *demonst*, and *adverb* do not have to be bound to any method; their purpose is to be reused in the definition of the (upgraded) composite event *moveObject*. Notice that when *verb*, *demonst*, and *adverb* are referred to in the definition of *moveObject*, their names are extended with the prefix *ce*. Because of this keyword, atomic events and composite events are clearly distinguishable (Table 4.5).

The compositionality of CEDL allows a better organization of the code: the definition of a complex composite event can be divided into smaller, reusable composite events. In this case, the four events shown in Equation 4.2 were declared in one single file; but it could also be possible to save the events *verb*, *demonst*, and *adverb* into a separated file to be imported (Figure 3.2, Tab 3) into a project containing the definition of *moveObject*. In this way CEDL supports modularization of code.

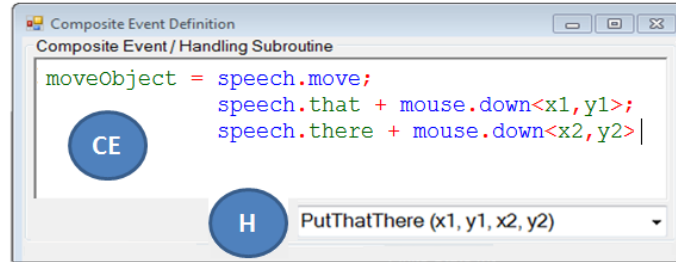


Figure 4.4: CEDL editor of the first, deprecated version of Hasselt UIMS, presented in [Cuenca 14b]. Each composite event could be bound with one event-handling callback only. Noticed that this editor was much simpler than the current one, shown in Figure 4.3.

4.4 Limitations of the CEDL

The first version of Hasselt UIMS, presented in [Cuenca 14b], only included CEDL. The FSAs generated from CEDL code were only used (1) to internally track the interaction state and (2) to be displayed as an animated diagram in a debugging tool (Section 3.2.3, Automata View). Although both functions were important, we later realized that the potential of the FSAs was not being well exploited.

Up until that moment, with the CEDL alone, programmers were restricted to bind one event-handling callback per composite event (Figure 4.4 shows the event *CE* being bound to the handler *H*). This implied that the callbacks could only be called once their associated composite events were fully detected, which prevented programmers from describing interactions with partial semantic feedback. For instance, with regard to the *put-that-there*, the highlighting of the selected object could not be implemented by using only CEDL since object selection occurs before the *put-that-there* finishes.

Another issue with the first version of Hasselt UIMS was the inability of CEDL to allow defining constraints among the atomic events comprising a composite event. This prevented one from describing touch interactions, for instance. When a user drags his fingers over a touch-sensitive screen, the time elapsed between the beginning and ending of a touch gesture may be decisive to determine whether the end user is flicking or panning.

Thanks to the questions raised during the EICS'14 conference and to the

discussions of the author with P. Palanque during his internship held in University Paul Sabatier at the end of 2013, we noticed that both limitations could be redressed if programmers were to be enabled to explicitly annotate every node and link of the FSAs with both event-callback functions and guard conditions. This is precisely the purpose of the System Response Definition Language (SRDL), to be presented in the next chapter.

4.5 Summary

This chapter presented the Composite Event Definition Language (CEDL) and its supporting tool, Hasselt UIMS. The CEDL is a declarative language that allows combining several user events into one single high-level event, herein called composite events. Those composite events are expected to be bound to externally defined methods. At runtime, Hasselt UIMS will call those externally defined functions whenever their corresponding composite events are detected.

With CEDL, unlike with mainstream event languages such as C# or Java, programmers do not have to implement a mechanism for detecting chains of events since this task is internally performed by Hasselt UIMS. Despite of this gain, the set of interactions that can be described with CEDL alone is restricted due to two limitations: (1) the inability of specifying constraints among the atomic events comprising a composite event, and (2) the inability to bind multiple event handlers to one single composite event. But these two issues were redressed by supplementing the CEDL with another language, as will be explained in the next chapter.

Chapter 5

SRDL: Responding to composite events

The preceding chapter presented the Composite Event Definition Language (CEDL), which was the only language in the first version of Hasselt UIMS, presented in [Cuenca 14b]. In that version, each composite event could be bound to only one event-handling callback, which was expected to be called upon the full detection of the composite event. However, we noticed that for many interactions, it is important to launch event handlers not only upon the full detection of a composite event but throughout this entire process so that partial feedback can be provided.

The possibility to attach multiple handlers to one composite event is brought about by a new language that supplemented the CEDL. This new language also allows defining spatial, temporal, and semantic constraints between the atomic events comprising a composite event. The System Response Definition Language (SRDL) is the topic of the present chapter.

With the System Response Definition Language (SRDL), the finite state automata (FSAs) auto-generated from CEDL code start playing a more important role than in the initial version of Hasselt UIMS, in which CEDL was alone. Beyond their initial function as animated diagrams and internal data structures of Hasselt UIMS, with the creation of SRDL, the FSAs can now also be used as *non-linear timelines* in which programmers can annotate multiple event-handling callbacks at different moments of the human-machine interaction.

5.1 System Response Definition Language (SRDL)

The Composite Event Definition Language (CEDL) serves to describe composite events, which, in simple terms, represent the sets of coordinated actions that the end user performs to activate the system's functionality. The System Response Definition Language (SRDL) serves to specify the multiple responses that the system will convey as the composite events are detected. Both languages together allow specifying human-machine multimodal interaction.

Hasselt UIMS can respond by launching the event handlers contained in the imported back-end application(s), by activating its internal software synthesizers (e.g. to synthesize voice from text), or by raising user-defined events. SRDL allows specifying both the moment and conditions under which the system responses are to be conveyed.

5.1.1 Multiple system responses at different times

To specify the moment when a system response has to be conveyed, SRDL requires programmers to refer to the FSA auto-generated from CEDL code (Section 3.3). The different nodes of an auto-generated FSA represent different interaction states. By annotating one response (e.g. an event-handling callback) in a node of a FSA, one is indicating that the response has to be conveyed when the interaction state represented by that node is reached. One can also annotate responses in the links of the FSA meaning that these responses must be conveyed during the interaction's state transitions.

Programatically, the multiple system responses to be conveyed in response to the (partial) detection of a composite event named *myEvent* are specified with the following pattern.

```

wrt ce.{myEvent}
  [ [@node({nodeID1}) | @link({nodeID1}, {evtName1})] do
    list1 of system responses
  [ when {guard condition1}} ]
    ⋮ ⋮ ⋮
  [ [@node({nodeIDn}) | @link({nodeIDn}, {evtNamen})] do
    listn of system responses
  [ when {guard conditionn}} ]
[triggers when {triggering condition}]

```

(5.1)

The keywords *wrt* and *ce* stand for *with respect to* and *composite event* respectively. Table 5.1 shows the syntax to be used for each type of system response supported by Hasselt UIMS.

System response type	Description
<i>call</i> : { <i>ns.cls.subName</i> }	call named subroutine in back-end application
<i>speak</i> : { <i>expression</i> }	speak sentence through text-to-speech
<i>play</i> : { <i>filePath</i> }	play an audio file
<i>raise</i> : { <i>evtName</i> }	raise an event named <i>ce.evtName</i>
<i>assign</i> : { <i>lstVarAssign</i> }	assign values to weakly typed variables

Table 5.1: Available types of system responses in Hasselt UIMS

As mentioned above, to specify the moment in which the system responses must be conveyed, one has to refer to the nodes/links of the FSA generated from *myEvent*. Multiple responses can be associated to one single node/link (e.g. an event-handling callback can be launched while a beep is played). The nodes and links may have a guard condition to indicate whether their associated responses must be executed or not. In the case of a node, the system responses bound to it will only be executed if its guard condition returns **true**. For a link, both their associated responses and the transition it represents will only be executed if its condition returns **true**.

Optionally, one can restrain the notification of a composite event by using a triggering condition, a special type of guard condition that is evaluated after the whole composite event has been detected. A triggering condition returning **false** will prevent the composite event *myEvent* to go to the Event Queue (Figure 3.7). In contrast, a triggering condition returning **true** will cause *myEvent* to go to the Event Queue, from where it will be eventually dequeued and used by other composite events –those including *ce.myEvent* in their definitions. By default, composite events go to the Event Queue.

5.1.2 Hasselt variables

The previous chapter showed that the information related to user events comes encapsulated in a set of variables called event parameters (Table 4.2). There are other types of variables that Hasselt programmers can define.

Types of variables

User-defined variables are those that programmers can explicitly declare with the keyword *assign* (Table 5.1), e.g. *assign: count = 0*. Programmers do not have to specify the datatype of the variables they define; Hasselt UIMS will process these variables as if they had the same datatype of their initial

value. User-defined variables can be integer, double, datetime, or strings. The keyword *assign* also allows maintaining variables, e.g. a counter can be updated with the statement *assign: count=count+1*. Multiple variables can be assigned and/or initialized in one single statement, e.g. *assign: x=1, sum=0, f2=f1+10*.

Callback-generated variables are those containing the returning values of the functions implemented in the back-end applications. Right after calling a callback function, Hasselt UIMS creates a variable with the same name of the function and sets it with its return value. For instance, after invoking a function with *call: numBoxesOn(x,y)*, the variable *numBoxesOn* will be automatically generated and initialized with the number of boxes placed over the point (x,y), which is a value calculated in a back-end application. The callback functions can only return values of type integer, double, datetime, or string; other data types will cause a runtime error. If a data structure or an object of a user-defined class has to be returned to Hasselt from a back-end application, it has to be encoded as a string. An example using callback-generated variables will be shown in Section 5.3.

Lifetime of variables

Event parameters, user-defined variables, and callback-generated variables are created at runtime, at exactly the first time in which they are referred to in the Hasselt code. These variables will be destroyed right after their associated composite events are fully detected or when the *reset command* (Section 3.1.3) is issued. These variables are called *local variables*.

User-defined variables may have longer lifespans if programmers assign a value to them at the zero-th node, e.g. *@node(0) do assign: x = 1*. Although, no FSA contains such a node, this assignment will create a variable that retains its value through multiple occurrences of a composite event, i.e. neither the completion of an interaction nor the reset command will reinitialize this variable. These variables are called *static variables*. An example of static variables is later shown in Section 6.2.2.

Parameter passing

In Hasselt, arguments can be passed to the parameters of an externally defined function only by value. Similarly, the output of a back-end function is returned by value to Hasselt UIMS. There is no direct way to modify the value of a Hasselt variable from the back-application and vice versa.

With respect to event bubbling, when a composite event is triggered, its event parameters are passed by value to other higher-level composite events.

5.1.3 Hasselt properties

The always difficult specification of time constraints can be eased with the properties provided by Hasselt, which can be invoked, at any moment, and from every composite event. The properties *Now.Date* or *Now.TotalSeconds*, for instance, return the current date and the number of seconds elapsed since a well-known point in time (00:00 UTC January 1, 0001 in the Gregorian calendar) respectively. The returning value of these properties can be stored in user-defined variables. For example, one way to measure the time elapsed between two distinct moments of an interaction is by annotating two different nodes of a FSA with the statements *assign: t1=Now.TotalSeconds* and *assign: t2=Now.TotalSeconds* and by interrogating the difference $t2 - t1$ at later point in time, i.e. at a more distant node.

5.1.4 Hasselt guard and triggering conditions

Both guard and triggering conditions may involve event parameters, user-defined variables, properties, or callback-generated variables that can be joined through a series of arithmetic and logical connectors. Furthermore, utility functions can be applied to the variables within the definition of a guard or triggering condition.

Arithmetic and logical connectors

Variables can be compared with the symbols $<$, $>$, $<=$, $>=$, $=$, $<>$ as well as with the operators *in* and *like*. Some examples of valid comparisons include $t1 < t2$, *color in ('yellow', 'blue', 'green')*, and *color like 'light*'*—where the types of both $t1$ and $t2$ are datetime variables whereas *color* is a string.

Guard or triggering conditions can contain several statements, like the ones mentioned above, joined by logical connectors, i.e. *and*, *or*, *not*. Valid guards include $x > 0$ *and* $y < 0$ and *count > 0 or x is not null*.

All basic arithmetic symbols ($+$, $-$, $*$, $/$) can operate on numeric variables in the usual manner. The symbol $+$, when applied to strings, serves to concatenate strings. The symbol $-$, when applied for two datetime variables (e.g. $t2 - t1$), returns the number of milliseconds that have elapsed from $t1$ to $t2$.

Utility functions

As mentioned above, Hasselt UIMS can only pass (receive) primitive values –namely integer, double, datetime, string– to (from) the back-end application. If an object of a user-defined type had to be passed (received), it has to be serialized, i.e. transformed into a string. There are utility functions to help with this process. These functions can be invoked at any time and from any composite event.

The size of a string can be obtained through the function *len(string)*. The function *trim(expression)* removes all leading and trailing blank characters. The function *substring(fullstring, start, length)* returns a fragment of *fullstring*, a portion of *length* characters that starts in the *start*-th position. The function *iif(expr; truepart; falsepart)* returns *truepart* if *expr* is true or *falsepart* in other cases. The function *isnull(expression; replacementvalue)* returns true if *expression* is true or *replacementvalue* otherwise.

For instance, the function *getScore()* of a back-end application can return an object of the class *Student* as the string ‘Fredy, Peru,20’. The student name, country of origin, and score encoded in these fixed-length fields can be decoded, with Hasselt, by applying the function *substring* three times on the callback-generated variable *getScore*. For this decoding to work correctly, the Hasselt programmer must know how the back-end application encodes the messages. That is why we mentioned, in Section 3.1.2, that the Hasselt programmer and the application developer must work in coordination with each other.

5.1.5 Hasselt user-defined events

A system may respond to a user action by calling an external function (with the keyword *call*), by activating built-in synthesizers (with *speak* or *play*), or by launching user-defined events (with a *raise* statement).

The statement *raise: evtFailure* in the SRDL code will make Hasselt UIMS to put a notification for the event *evtFailure* in the Event Queue (Figure 3.7). This would be irrelevant unless there is another composite event including *ce.evtFailure* in its definition, which is expected to be the case. User-defined events do not need to have a definition in CEDL; they are dynamically created when found in the SRDL code. User-defined events can carry an arbitrary number of event parameters. By convention, user-defined event names start with the prefix *evt*. A prototype that exploits these type of events is commented below, in Section 5.5.2.

5.1.6 Types of constraints describable by Hasselt

By using the aforementioned SRDL syntax, programmers can describe spatial, temporal, and semantic constraints among the constituent events that comprise a composite event.

A spatial constraint describes a relation between the relative positions where two or more events occur. This can be as simple as requiring a touch not to move while holding on an interface element, or more complex, such as those imposed to several sequential gesture strokes that together pretend to draw a geometrical pattern (e.g. crosses or asterisks). Spatial constraints can be described with SRDL as part of a guard or triggering condition.

A temporal constraint describes a relation among the timestamps of two or more events. Temporal constraints can be qualitative or quantitative; the former establishes the order of occurrence of two or more events; the latter imposes a more strict relation such as the time elapsed between two events [Lalanne 09]. Qualitative temporal constraints are described with the symbols FOLLOWED BY (;) and AND (+) of CEDL; quantitative constraints can be imposed by using user-defined datetime variables in the guard or triggering conditions of a SRDL block.

A semantic constraint is an application-specific relation that scopes the set of possible actions that the end user can take. For instance, a system that requires end users not to put a second finger on a touchscreen while a first finger is performing a unistroke gesture.

To illustrate the SRDL, we present another version of the put-that-there interaction, where partial semantic feedback as well as temporal constraints are specified.

5.2 Enhancing put-that-there with SRDL

Here we present a slight variation of the interaction *put-that-there* described in the preceding chapter. First, the selected object is highlighted, which requires invoking a handler before the completion of the composite event *moveObject*. Second, the mouse click is defined as the sequence of *mouse.down* and *mouse.up* events given that both events occur within a time interval of 200 milliseconds (time constraint).

We define this interaction with the composite events in Equation 5.2.

$$\begin{aligned}
 \text{event } mclick\langle x, y \rangle &= \text{mouse.down;} \\
 &\quad \text{mouse.up}\langle x, y \rangle \\
 \\
 \text{event } moveObject &= \text{speech.put;} \\
 &\quad \text{speech.that} + \text{ce.mclick}\langle x1, y1 \rangle; \\
 &\quad \text{speech.there} + \text{ce.mclick}\langle x2, y2 \rangle
 \end{aligned}
 \tag{5.2}$$

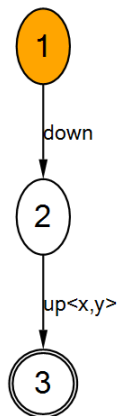
Before binding the aforementioned composite events to event handlers, one has to syntax-check the CEDL code. During this process, Hasselt UIMS generates the two automata shown in Figure 5.1. Programmers can use the nodes and/or links of these automata as reference points, to indicate the moments when event handlers must be launched and/or when the spatial, temporal, semantic constraints must be evaluated.

The automaton (b), for instance, represents the different stages on the path to the detection of *moveObject*. By referring to this automaton, programmers can instruct Hasselt UIMS to respond after the full detection of this event, i.e. state 8, after selecting the target object, i.e. state 5, during the recognition of the speech input ‘put’, i.e. link (1, *put*), etc.

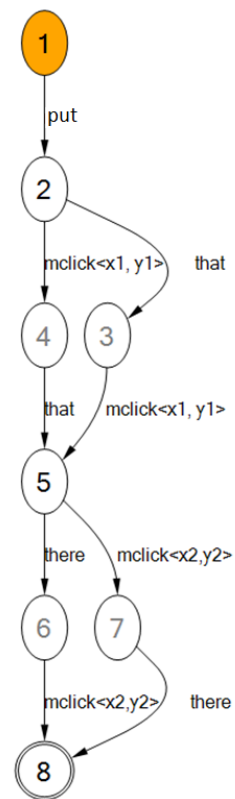
The model that describes the interaction *put-that-there* is completed with the following SRDL code.

$$\begin{aligned}
 \text{wrt } ce.mclick\langle x, y \rangle \\
 \quad @link(1, \text{mouse.down}) \text{ do} \\
 \quad \quad \text{assign : } t0 = \text{Now.TotalMilliseconds;} \\
 \quad @link(2, \text{mouse.up}\langle x, y \rangle) \text{ do} \\
 \quad \quad \text{assign : } t1 = \text{Now.TotalMilliseconds;} \\
 \quad \text{triggers when } t1 - t0 \leq 200 \\
 \\
 \text{wrt } ce.moveObject \\
 \quad @node(2) \text{ do} \\
 \quad \quad \text{speak : 'what?'}; \\
 \quad @node(5) \text{ do} \\
 \quad \quad \text{speak : 'where?'}; \\
 \quad \quad \text{call : } ptt.frmPTT.HighlightObjectOn(x1, y1); \\
 \quad @node(8) \text{ do} \\
 \quad \quad \text{speak : 'done!'}; \\
 \quad \quad \text{call : } ptt.frmPTT.PutThatThere(x1, y1, x2, y2);
 \end{aligned}
 \tag{5.3}$$

The event *mclick* will be triggered whenever the mouse is depressed and released in a quick succession –no longer than 200 milliseconds. The temporal constraint can be specified thanks to the variables *t0* and *t1*, which are time-stamped whenever the end user presses and releases the mouse respectively.



(a) Event mclick



(b) Event moveObject

Figure 5.1: Implementing the *put-that-there* command.

The event *mclick* will carry two parameters when triggered; these represent the (x-y)-cursor position when the mouse was clicked.

With respect to the event *moveObject*, its gradual detection will be acknowledged through voice feedback, e.g. after saying ‘*put*’, the UIMS will immediately reply with ‘*what?*’ so that users can be aware that their intention of moving an object has been recognized. The selected object will be highlighted through the method *HighlightObjectOn*, and moved through the *PutThatThere* method. These methods are implemented in the class *ptt.frmPTT* of the back-end application. Figure 4.3a on page 66 shows the GUI of the back-end application.

5.3 Describing touch and body gestures

As commented in the previous chapter, touch and body movement interactions could not be described with the CEDL alone. This is because CEDL does not allow establishing constraints among the events comprising composite events. This section will show that when CEDL is complemented with SRDL, several types of touch and body movement interactions involving different types of constraints can be described. The next chapter will show a comprehensive application that combines these two modalities.

5.3.1 Single-stroke touch gestures

A single-stroke touch gesture is produced when one finger navigates on a touchscreen with the intention of performing a task. Hasselt allows describing single-stroke touch gestures as will be shown below.

The *flick-down* gesture is that where the user puts its finger on a touchscreen and drags it down in a quick motion. This gesture is characterized by a *touch-down* event followed by an arbitrary number of *touch-move* events and these, in turn, followed by a *touch-up* event. The formal description of this gesture is as follows:

$$\begin{aligned} \text{event flickdown} = & \text{tscreen.down}\langle x1, y1, t1, id1 \rangle ; \\ & \text{tscreen.move}\langle x2, y2, spx, spy, t2, id2 \rangle^* ; \\ & \text{tscreen.up}\langle x3, y3, t3, id3 \rangle \end{aligned} \quad (5.4)$$

The FSA-based representation of the *flickdown* event is shown in Figure 5.2b. The system response to this composite event is specified in Equation 5.5. This equation states that only those touch events coming from the first finger will be processed ($id = 1$). The condition $y3 > y1$ guarantees that the touch moves down. The condition $t3 - t1 < 500$ restricts the flick to a maximum duration

of 500 milliseconds. The event *flickdown* is only triggered if these spatial and temporal constraints are met.

```

wrt ce.flickdown
  @link(1, tscreen.down(x1, y1, t1, id1)) do
    when id1 = 1;
  @link(2, tscreen.move(x2, y2, spx, spy, t2, id2)) do
    when id2 = 1;
  @link(3, tscreen.up(x3, y3, t3, id3)) do
    when id3 = 1;
triggers when y3 > y1 and t3 - t1 < 500

```

(5.5)

5.3.2 Multi-stroke touch gestures of arbitrary length

The event *manyClick*, defined in Section 4.3.3 on page 68, accumulated the clicks into an array so that the back-end application had to deduce the number of clicks from the length of the array. This example shows a more effective way to describe events that are repetitions of other more fine-grained events, e.g. double-tap, double-click, etc. We will define an event characterized by an arbitrary number of flicks quickly performed one after another (Figure 5.2a). This is described with the following pattern.

```

event reperflicks = ce.flickdown;
                  ce.flickdown*;
                  delay-500

```

(5.6)

```

wrt ce.reperflicks
  @node(1) do
    assign : cont = 0;
  @node(2) do
    assign : cont = cont + 1;
  @node(3) do
    call : mediaPlayer.Form1.decrVolume(cont);

```

(5.7)

The SRDL code shown in Equation 5.7 describes how to handle the composite event *reperflicks*, whose reciprocal FSA is shown in Figure 5.2c. The event *ce.reperflicks* is triggered when the end user stops flicking, i.e. after 500 milliseconds of inaction. The user-defined variable *cont* is used to count the number of successive flicks. This variable is passed to a back-end application when the user stops flicking. The back-end application is a video player whose volume will be decreased depending on the number of flicks performed by the end user. This interaction is part of the study case presented in Chapter 6.

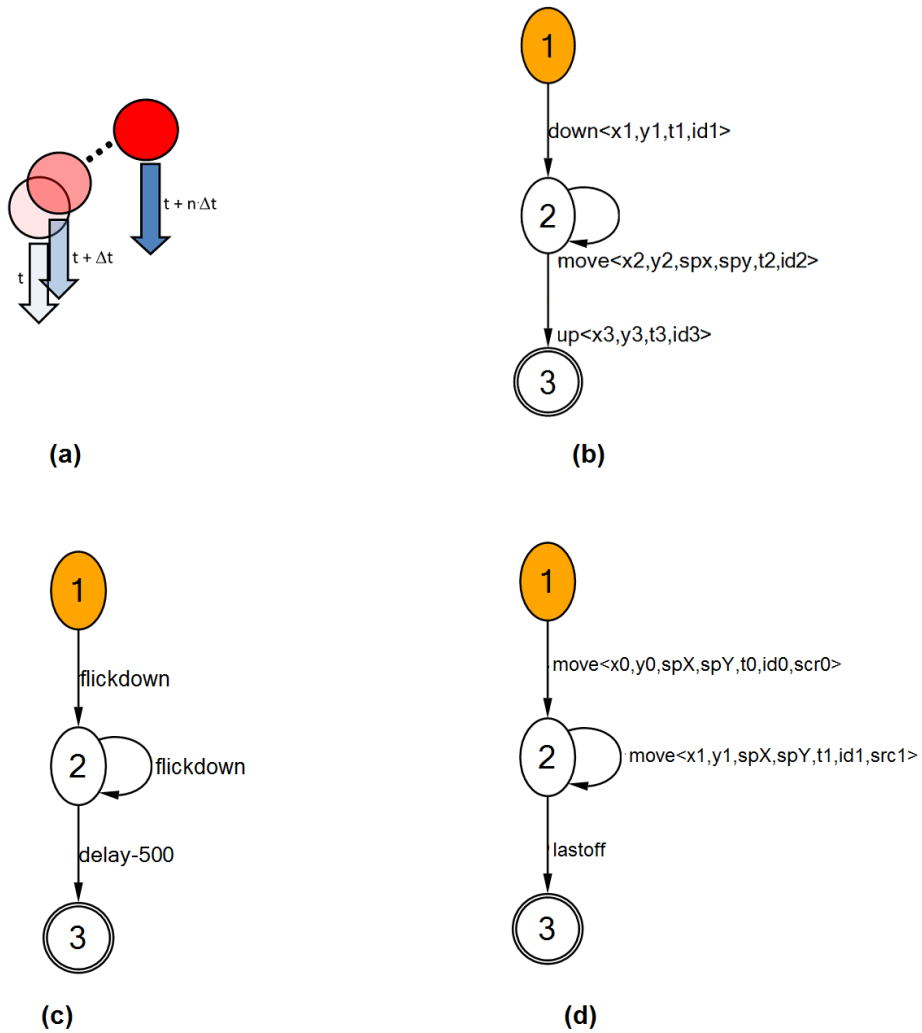


Figure 5.2: (a) Event composed of an arbitrary number of flick downs. (b) FSA-based representation of event *flickdown* (Equation 5.4), which describes one of the strokes shown in (a). (c) FSA-based representation of event *reperflicks* (Equation 5.6), which describes the whole multi-stroke gesture shown in (a). (d) FSA-based representation of the multi-touch gesture *zoom* (Equation 5.8)

5.3.3 Multitouch gestures

The touch events recognizable by Hasselt UIMS do not only carry individual information about each touch (e.g. position, speed, timestamp) but also aggregate information about the whole set of touches on the touchscreen (e.g. number of touches, maximum distance between the touches, minimum angle with respect to the first touch). This information does not have to be aggregated from the individual touch events, but it is automatically calculated and provided to users of Hasselt UIMS.

The availability of aggregate information simplifies the description of those gestures affected by the *finger permutation problem* [Cirelli 14]. The pinch gesture, for instance, commonly used to enlarge/collapse a region of the touchscreen, can be performed by modifying the number of fingers to the whims of the user, e.g. he can start using three fingers, switch to four in the middle of the gesture, and finish with two fingers only. It seems unfeasible to create models that consider all possible finger permutations explicitly.

The aggregate information calculated by Hasselt UIMS allows an alternative way to model multitouch interaction. Instead of specifying the behavior of each touch, multitouch interaction can now be described by referring to properties of the whole set of touches. With this latter approach, the pinch gesture is robustly described as “if there are multiple fingers on the touchscreen and the distance between them is changing, then, the screen content must be resized”. This specification does not refer to any individual touch but to their relative distance. Therefore, it is indifferent if the number of touches remains fixed or changes over the gesture, which permits to bypass the finger permutation problem. The specification of pinch gesture is implemented with the following CEDL code:

$$\begin{aligned} \text{event } \text{zoom} = & \text{tscreen.move}(x0, y0, spX, spY, t0, id0, scr0); \\ & \text{tscreen.move}(x1, y1, spX, spY, t1, id1, scr1)^*; \\ & \text{tscreen.lastoff} \end{aligned} \quad (5.8)$$

As shown in Table 4.6, the last parameter of the event *tscreen.move*, i.e. *scr*, is a data structure containing aggregate information of all the fingers placed over the touchscreen. This information includes the number of touches, the minimum/maximum/average distance between the first touch and the other ones, and the other values shown in Table 4.4.

The event *zoom*, whose reciprocal FSA is shown in Figure 5.2d, must be

handled according to the following SRDL code.

```

wrt ce.zoom
  @link(1, tscreen.move(x0, y0, spX, spY, t0, id0, scr0)) do
    assign : prevDist = scr0.avgDistanceToFirst;
    when scr0.numberOfFingers > 1;
  @link(2, tscreen.move(x1, y1, spX, spY, t1, id1, scr1)) do           (5.9)
    call : rotation.Form1.resize(scr1.avgDistanceToFirst);
    assign : prevDist = scr1.avgDistanceToFirst;
    when scr0.numberOfFingers > 1 and
      scr1.avgDistanceToFirst <> prevDist;

```

The field *avgDistanceToFirst* of the data structure *src* stores the average distance of all touches to the first touch. The function *resize*, implemented in a back-end application, will be launched whenever this distance changes over time (*scr1.avgDistanceToFirst* <> *prevDist*). By using the aggregate information of *src*, the gesture model does not have to take into account the (variable) number of touches placed on the touchscreen; their average distance is the only thing that matters. The more touches used in the pinch-out gesture, the higher the rate of touch-move events, and thus, the faster the resizing of the selected object.

Hasselt UIMS remembers the order in which the touches were placed. If the first touch is removed during the gesture, the second touch will be considered as first touch.

5.3.4 Free-form hand gestures

Hasselt UIMS cannot only invoke and pass values to the functions of imported back-end applications. As described above, in Section 5.1.2, one can also process the values returned from the back-end applications by using callback-generated variables.

The present example describes a system capable of recognizing the digits that a user delineates with his right hand on the air (Figure 5.3a). The drawing mode starts when the end user puts his left hand in front of him, and finishes when this hand is put back to its normal position.

This system was implemented by defining one event that triggers when the user's left hand is in front of him and another event for when the left hand is in its normal position. Additionally, a DLL had to be imported into Hasselt UIMS. It contains a function *getBestMatch(x[], y[], t[])*, which is able to recognize two-dimensional gestures from a set of timestamped points.

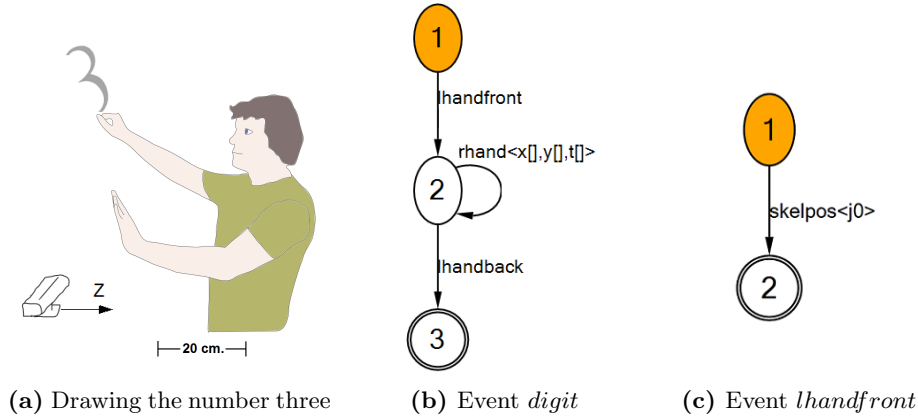


Figure 5.3: (a) The end user must put his left hand in front to start drawing a digit. (b) The points covered by his stroke, and their timestamps are accumulated in $x[]$ and $y[]$, and $t[]$ respectively. (c) Common structure for the events *lhandfront* and *lhandback* used in the definition of the event *digit*.

The code for describing the aforementioned interaction is as follows.

```

event lhandfront =      kinect.skelpos(j0) ;

event lhandback =      kinect.skelpos(j0) ;

event digit(getBestMatch) = ce.lhandfront;
                          kinect.rhand(x[ ], y[ ], t[ ])*;
                          ce.lhandback

```

(5.10)

```

wrt ce.lhandfront
  triggers when j0.Head.Z - j0.HandLeft.Z > 0.20

wrt ce.lhandback
  triggers when j0.Head.Z - j0.HandLeft.Z < 0.05

wrt ce.digit(getBestMatch)
  @node(3) do
    call : gest2d.utils.getBestMatch(x, y, t);
    triggers when getBestMatch <> 'none'

```

(5.11)

The events *lhandback* and *lhandfront* are triggered when the user moves his left hand back and forth respectively. The event *digit* reuses the events *lhandback* and *lhandfront* to describe the overall interaction. As specified

in the event *digit* (Equation 5.10), from the moment when the left hand is put forward, all the (x-y)-points of the right hand are collected into arrays until the left hand is put back to its normal position. In this final moment, Hasselt UIMS will call the function $getBestMatch(x[], y[], t[])$, implemented in a DLL.

The return value of $getBestMatch(x[], y[], t[])$ is a string with the name of the digit encoded in the arrays x and y , or ‘none’ if the free-form gesture did not match with any digit template. As discussed in Section 5.1.2, this return value can be read from the callback-generated function $getBestMatch$, which will be automatically created and set by Hasselt UIMS.

The condition $getBestMatch \neq \text{‘none’}$ guarantees that the event *digit* will only be triggered if the free-form gesture is recognized as a 1-9 digit. In this case, *digit* will carry a parameter indicating the name of the digit depicted by the end user. In Figure 5.3a, for instance, when the event *digit* will be finally triggered, it will carry the parameter $getBestMatch = \text{‘three’}$.

Finally, notice that the recognition of free-form gestures generated with a mouse or with a finger on a touchscreen device require similar code to the one used to define the event *digit*.

5.3.5 Body movements

Here we present a prototype able to track the coordinated movements of the two hands of a user doing stretching exercises. Once the presence of the end user is detected, the prototype synthesizes the voice message ‘start’ meaning that he can start to alternatively raise his left and right hand (Figure 5.4a). During the supervised training session, the system counts out loud the number of repetitions the user is doing. The system can also rush the user (by saying ‘go faster’) if he is being too slow. When the user decides to finish his routine, by saying ‘finish’, the system informs him about the number of repetitions, the time required to do so, and an overall evaluation about whether the efficiency achieved is good or bad. This prototype does not require back-end applications; it is completely described with Hasselt

The aforementioned prototype is implemented by declaring four composite events: (1) the event *lHandUp* occurs when the user raises his left hand, (2) *rHandUp* occurs when the user raises his right hand, (3) the whole training session is described as the event *training*, and (4) one artificial event, called *timing*, is used to measure the time between consecutive movements and to generate voice feedback based on these measurements. Three of these composite events are declared in Equation 5.12. The event *rHandUp* is not shown

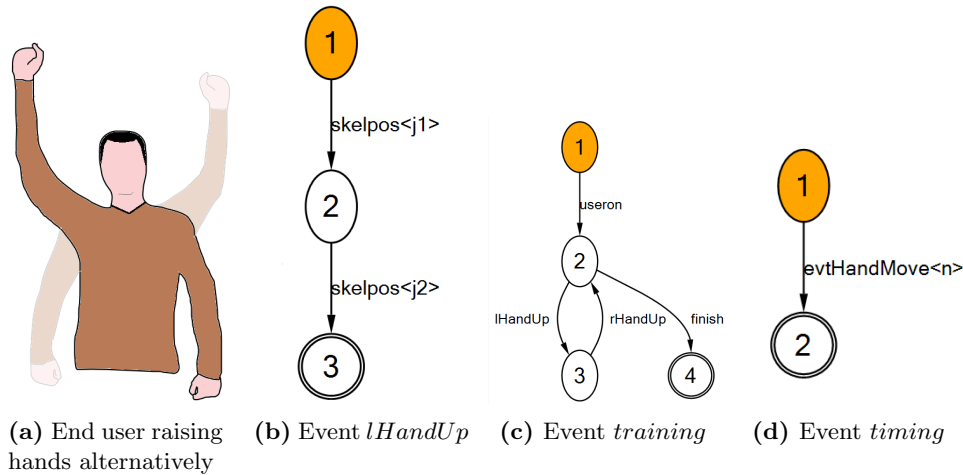


Figure 5.4: Description of a prototype that supervises the end user during his training session. A demo of this system is shown in <https://youtu.be/rKBNi4VEaKM>

but it is analogous to *lHandUp*. These three events are handled according to the specifications of Equation 5.13.

The event *lHandUp* occurs when the user is detected with his left hand on top his head ($j2.HandLeft.Y > j2.Head.Y$) provided that some time earlier (not necessarily in the previous frame, but no longer than 3 seconds either) he was caught with the left hand below the head ($j1.HandLeft.Y < j1.Head.Y$).

The event *training* starts when the Kinect detects the presence of the user (i.e. when *kinect.useron* is fired) and it can include an arbitrary number of *lHandUp* and *rHandUp* events (notice the loop in Figure 5.4c). The event *training* maintains a counter, *count*, that is increased every time the events *lHandUp* and *rHandUp* are detected in the expected order. Finally, every time that *lHandUp* and *rHandUp* occur, a user-defined event, *evtHandMove*, is raised carrying the counter *count* as a parameter. The event *evtHandMove* is captured by the composite event *timing*.

The event *timing* synthesizes the number of repetitions ('one', 'two', ...) so that the user can keep the pace. By using a static variable, *t1*, the composite event *timing* also estimates the time elapsed between consecutive hand raising movements. Every time this number falls below a certain threshold,

the prototype synthesizes ‘go faster’ so that the end user have to speed up.

```

event lHandUp = kinect.skelpos(j1) ;
                kinect.skelpos(j2)

event training = kinect.useron;
                 (ce.lHandUp; ce.rHandUp)*;
                 speech.finish
(5.12)

event timing = ce.evtHandMove(n)

```

```

wrt ce.lHandUp
@link(1, kinect.skelpos(j1))do
  assign : t1 = Now.TotalMilliseconds;
  when j1.HandLeft.Y < j1.Head.Y;
@link(2, kinect.skelpos(j2))do
  assign : t2 = Now.TotalMilliseconds;
  when j2.HandLeft.Y > j2.Head.Y;
triggers when t2 - t1 < 3000

wrt ce.training
@link(1, kinect.useron)do
  assign : count = 0, init = Now.TotalSeconds;
  speak : ' start';
@link(2, ce.lHandUp)do
  assign : count = count + 1;
  raise : evtHandMove(count);
@link(3, ce.rHandUp)do
  assign : count = count + 1;
  raise : evtHandMove(count);
@link(2, speech.finish)do
  assign : fin = Now.TotalSeconds, totaltime = (fin - init);
@node(4)do
  speak : ' you did' + count +' repetitions in' + totaltime +' seconds.
          That is ' + iff(totaltime/count < 2, ' great', ' very - bad');

wrt ce.timing
@node(0)do
  assign : t1 = Now.TotalMilliseconds;
@link(1, ce.evtHandMove(n))do
  assign : t2 = Now.TotalMilliseconds;
@node(2)do
  assign : t1 = t2;
  speak : iff(t2 - t1 > 1500 and n > 1, ' gofaster', ' ' + n);
(5.13)

```

5.4 Technical details

5.4.1 Management of parallel inputs

For Hasselt UIMS, two events are considered simultaneous as long as they are detected within a short time interval.

What are stable and unstable nodes?

Hasselt UIMS handles parallel inputs by exploiting the fact that the nodes of the auto-generated FSA are classified as one of two types: stable or unstable. This classification is performed automatically, when the CEDL code is compiled (Section 3.3.1), and cannot be altered later. At runtime, Hasselt UIMS will never remain more than a certain amount of time in an unstable node. If this (configurable) time threshold is reached, the UIMS will automatically go back to the last visited stable node.

To better illustrate this mechanism, let us refer back to the *put-that-there*, or, more precisely, to the FSA shown in Figure 5.1b, which was generated from the composite event *moveObject*. There the nodes 3, 4, 6, and 7, which are represented in lighter color, are unstable whereas the others are stable nodes. At runtime, if the speech input *that* does not co-occur with an already perceived mouse click (e.g. due to speech recognition error), the UIMS will go back from node 4 to node 2. And from this point, the end user will have a new chance to issue the speech input and the mouse click, but this time at (almost) the same time.

On the one hand, this mechanism guarantees simultaneity. The UIMS will move to node 5 only if the two involved inputs co-occur. On the other hand, it also produces error awareness. When entering back into node 2, the UIMS will synthesize *what?* (again!) and thus, the end user will realize he failed to select an object; otherwise, the UIMS would have asked him: *where (to move the selected object)?*.

As discussed below, this example can be modified so that the end user can receive more detailed feedback, not only to acknowledge him about the current stage of the interaction, but also about the input recognition errors occurred during the interaction, if any.

Feedback on simultaneity failures

In the Put-That-There system presented in Equation 5.2, the voice messages *what (to move)?* and *where (to move it)?* are not issued only when the

interaction is progressing correctly. As commented above, as a way to provide end users with error awareness, these messages are also synthesized when the speech inputs and clicks are not detected simultaneously. But based on this voice feedback only, the user cannot distinguish one situation from the other. Fortunately, Hasselt has the expressiveness required to specify specific feedback for each scenario.

A more sophisticated version of the Put-That-There prototype can be observed in action in a publicly available video¹. If everything goes right, the user will hear, in a timely manner, the typical *‘what do I have to move?’* and *‘where to move it?’*. But if the end user fails trying to speak and click in parallel, he will receive more specific messages. The sentences *‘you had to click on an object’* or *‘you had to click on the new position’* are synthesized to make the user aware that he forgot to click when trying to select or to move an object respectively. Besides, the message *‘I did not hear anything’* will be synthesized to make the end user notices that he was not heard.

The key to implement the aforementioned feature is to determine whether the interaction enters into *node(2)* or *node(5)* (see Figure 5.1b) by walking forward towards the final node or after falling back from an unstable node. This can be achieved by setting a variable *msg* with different values in each node adjacent to *node(2)* and *node(5)*. Later, instead of synthesizing a fixed string, as in the original case, the voice synthesizer of Hasselt UIMS must now be fed with the variable *msg*, e.g. *speak: msg*. The full code of the commented interaction is shown in Appendix C.1.

5.4.2 Interruptibility and rolling-back

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 programmers to declare a *reset command*, a specialized speech input whose detection causes the immediate reset of the UIMS: local variables are destroyed and FSMs return to their initial state. The reset command is not declared within the programming environment, but in a configuration file that Hasselt UIMS reads at startup (Section 3.1.3).

In many cases, such as those when the interaction does not involve partial semantic feedback, the reset of Hasselt UIMS is enough and programmers will not have to do anything else to maintain their prototypes in a consistent state. But if the reset command is detected after partial semantic feedback

¹Hasselt UIMS giving feedback about input recognition errors: <https://www.youtube.com/watch?v=01ETIsCoMq8>

was generated programmers may have to roll back the effects produced by the feedback. For example, if end users cancelled the *put-that-there* interaction after object selection, the highlighting produced because of the selection has to be undone. Hasselt UIMS cannot directly “unhighlight” the selected object because this is part of an external back-end application and its internal representation is not directly visible from Hasselt UIMS.

However, programmers can handle this situation by attaching a back-end function *UnHighlightObjects()* to the initial node of the FSM representing the *put-that-there* interaction. This initial node will be automatically reached upon the detection of the reset command. If the back-end application performed other internal computations during object selection, the function *UnHighlightObjects()* must undo their effects too. Just as Hasselt UIMS resets its local variables, the back-end applications are expected to do the same job with their internal variables.

Finally, there is still one issue with aforementioned solution. The initial state is reached in two different scenarios: (1) after the reset command is detected and (2) after the *put-that-there* interaction is completed. The back-end function *UnHighlightObjects()*, or in general, any roll back function, has to be called only if the initial state is reached because of the reset command (i.e. first scenario). This discrimination can be implemented in Hasselt by maintaining one static variable, which can be interrogated in the initial state to know whether the last state of the *put-that-there* was the final state or not. As mentioned in Section 5.1.2, static variables are never reset.

In Appendix C.2, one can see the code of a prototype that allows users to move and create an arbitrary number of objects through speak-and-mouse commands. Both the creation and displacement of objects can be cancelled in the middle of the interaction and the prototype can undo the effects of the partially entered commands in a proper manner. The prototype can be seen in action in a publicly available video².

5.4.3 Evaluation of expressions

Hasselt expressions are operations between constants, variables, properties, and event parameters. In order to determine the value of an expression (i.e. to evaluate an expression), Hasselt uses the class *DataTable*³. Right before evaluating an expression, Hasselt UIMS creates a table with as many columns as variables are included in the expression. The table is then filled with the

²Rolling-back with Hasselt: <https://youtu.be/zcKFgZTaFhw>

³<https://msdn.microsoft.com/en-us/library/6zd7cwzh%28v=vs.110%29.aspx>

values of the involved variables and augmented with one auto-computed column whose formula is the expression that has to be evaluated. In this way, the value of the expression will appear automatically in the auto-computed column right after this column is created. For instance, assume that at the moment of evaluating the expression $x + y$, the Hasselt variables x and y have the values 10 and 20, respectively. This will cause Hasselt UIMS: (1) to create a table with two columns, labelled x and y , (2) to fill the first and only row of the table with the tuple (10, 20), (3) to add one auto-computed column –with the formula $x + y$ – to the table, and (4) to obtain the value of $x + y$ (= 30) by querying the auto-computed column. If the expression to be evaluated contains variables that were never initialized, these will be considered as null variables.

The use of tables with auto-computed columns gives the UIMS developer the advantage of not having to parse expressions and obtained their values from their parse trees; instead, he delegates the evaluation of expressions to the `DataTable` class. But this shortcut comes with disadvantages too. The formulas of the auto-computed columns cannot contain references to array elements meaning that expressions such as $xs[1] + xs[2]$ will not be recognized neither in guard conditions nor in assignment statements. To reduce the impact of this limitation, at least to some extent, we arranged Hasselt UIMS so that expressions can refer to aggregate information of arrays. After such arrangements, expressions like $xs.avg > 0$ and $xs.count > 1$ become valid and allow, at least, some manipulation of arrays; other aggregate functions that can be used over arrays are *min*, *max*, and *sum*, whose names are self-explanatory. Besides, the arrays can also be passed as a whole to the back-end applications, as shown in Section 5.3.4.

5.4.4 Speech recognition grammars

Hasselt requires two speech recognition grammar files that will be automatically merged at startup, meaning that all terminal symbols included in both grammars will be recognizable by Hasselt UIMS, at runtime.

The reason for separating the set of recognizable speech inputs into two grammar files is merely cosmetic. Only the terminal symbols of one grammar file, herein called *main grammar*, will be displayed in the auto-completion popups. When modeling voice commands such as ‘*create a blue circle*’ or ‘*remove red squares*’, for instance, we used to put only the inputs ‘*create*’ and ‘*remove*’ into the main grammar file whereas the potentially long lists of color names and geometrical shape names will go to the secondary grammar. This

way we avoid the situation in which the auto-completion popups displayed by the CEDL editor (after the keyword *speech*) are excessively large to the extent that they become useless. The secondary grammar can also be used to store all those terminal symbols that include whitespaces: sentences like ‘*Of course*’, ‘*Cancel the command*’, and ‘*Yes, I do accept*’ can be recognized by Hasselt UIMS as if they were one single word (Section 4.3.1) even though the CEDL compiler cannot parse atomic events such as *speech.Yes, I do accept*. This was another case that encouraged us to provide programmers with a simple way to make Hasselt UIMS aware that some speech inputs, despite being recognizable, must be hidden from the auto-completion popups.

5.5 Expressiveness of CEDL/SRDL

5.5.1 Negation of events

Mudra includes notations for negating events. Their authors claimed, without proofs or examples, that this is a powerful feature in a multimodal interaction description language [Hoste 11]. While searching for additional opinions about an operator for negating events, we noticed that, in the domain of Complex Event Processing (see Appendix B), many composite event definition languages include a negation operator, but the presence of this operator has also been put into question. For instance, MIT researchers, Mei and Madden, claimed that “there is no way to represent the absence of an occurrence” [Mei 09]. “Though we could add a special ‘did not occur’ event –the authors continued–, it is unclear what timestamp to assign such events or how frequently to output them. Semantically, it also makes little sense to combine negation with disjunction (i.e., $A|!B$) or Kleene closure (i.e., $!A^*$)”.

Based on these divided positions, we decided not to implement a negation of events, first, because we did also foresee the implementation problems pointed out by Mei and Madden, and second, because Hasselt already allows negating events without having a specialized event operator to represent negation. For example, the sequential occurrence of the events *move.down* and *mouse.up* without any interleaving instance of *mouse.move* can be declared as the event *perfectClick* = *mouse.down;mouse.up* | *mouse.move*(*x,y*), annotated with the triggering condition *triggers when x is null*. As can be interpreted from Figure 5.5a, the (undesired) occurrence of *mouse.move* will break the non-nullity of their parameters (i.e. *x* and *y* will be assigned with values), which, in turn, will prevent *perfectClick* from being triggered. As mentioned in Section 5.4.3, when evaluating triggering conditions, those pa-

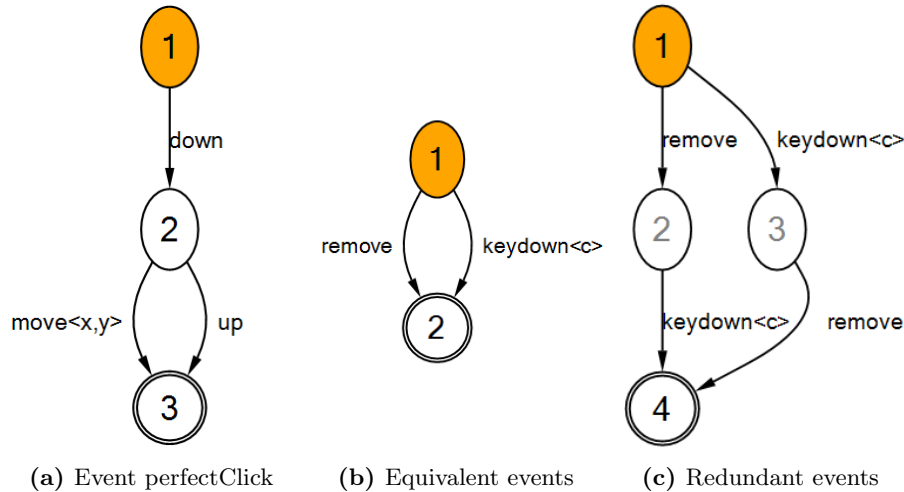


Figure 5.5: (a) To avoid interleavings of *mouse.move* events during a mouse click, *perfectClick* must only be triggered when x is null. (b) Any system response annotated in *node(2)* will be conveyed if the command for deletion is detected either via speech or via keyboard. (c) Any system response annotated in both unstable nodes, *node(2)* and *node(3)*, will be conveyed only once even if the command for deletion arrives redundantly via voice and via keyboard.

rameters that were never set –which is the case of x and y when a perfect click is performed– are considered null.

The same pattern can be repeated to specify interactions like *perfectTap* or other more complex scenarios like, for instance, a body gesture that finishes once it is interrupted by speech input.

5.5.2 About the CARE properties

The CARE properties are a well-known formal framework for reasoning about the design of multimodal systems [Coutaz 95]. Some researchers have explicitly introduced symbols to represent the Complementarity, Assignment, Redundancy, and Equivalence (CARE) of inputs that characterize multimodal interaction in their modelling languages [Serrano 08, Dumas 10]. Without direct references to CARE, Hasselt can describe these four types of inputs combinations and more than that.

Complementary inputs were already described for the interaction *put-that-there*. The complementarity between the speech inputs and the mouse clicks required to disambiguate their meanings was represented by the event operator

AND (+). As to the Assignment of a modality, the second CARE property, we can refer the reader back to Section 5.3, which was full of examples in which the touch modality was assigned, i.e. it was the only modality involved during the interaction.

With regard to the equivalence and redundancy of inputs, which are two related concepts, let us refer to a system that can recognize both the speech input *remove* and a keystroke on `DEL`. These two inputs are said to be equivalent if one of them is necessary and sufficient to call the function *rmvSelObjs*, in charge of removing selected objects. This equivalence can be described in Hasselt by connecting the speech input and the keystroke with the operator OR (`|`), and by annotating a function call to *rmvSelObjs* in the final node of this composite event (Figure 5.5b). For the same system, the two inputs are said redundant if each of them can be issued alone or both can be issued at the same time, but in either case, the system will only respond once, i.e. the function *rmvSelObjs* will be executed only once. This redundancy can be described by connecting the speech event and the keystroke event with the operator AND (+) and by annotating a function call to *rmvSelObjs* in each of its unstable nodes (Figure 5.5c).

Finally, the operator ITERATION (*) does not have an analogous one in the CARE properties. This operator is very important to describe the streams of inputs conveyed by the Kinect sensor, by a moving touch, or by a moving mouse, as shown in multiple examples throughout the thesis. One potential explanation for this omission may be that the CARE were designed to represent the “relations between interaction techniques”, not to represent the relations between user inputs. In our opinion, the CARE properties may be at a too high level of abstraction to the taste of a programmer.

A more complex example of redundancy

Hasselt can handle redundancy in more complex interactions, as proven in a publicly available video⁴. The video shows that when only one input, either the speech input ‘*remove*’ or a keystroke on `DEL`, is perceived, the system will require the end user to confirm the deletion of the interface objects. But such a confirmation is not required when both inputs are issued at the same time.

One of the composite events required to specify this interaction is the event *deleteNow*, whose FSA-based representation is shown in Figure 5.6a. As in the previous, simpler example, the function *rmvSelObjs* is also annotated in the the final node of the FSA. But, for this case, the FSA has additional

⁴Hasselt UIMS handling redundant inputs: <https://youtu.be/B4Zwmw6M1eI>

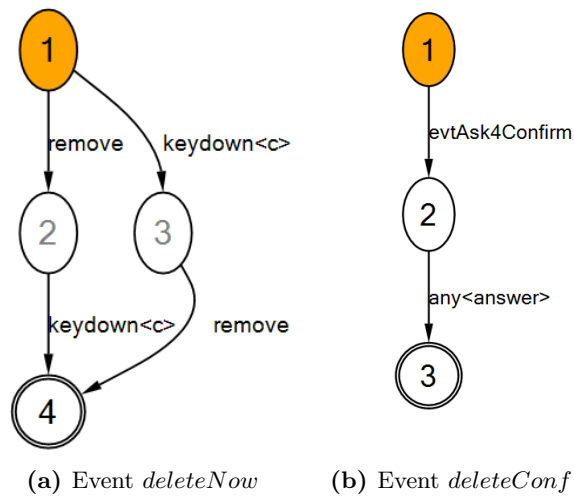


Figure 5.6: (a) Event *deleteNow* will be triggered when two redundant inputs are used simultaneously with the intention of removing all objects without need of confirmation. (b) Event *deleteConf* will be triggered after receiving a confirmation message for removing the interface objects. This latter composite event starts when a notification that only one of the two equivalent inputs (the speech input ‘*remove*’ or a keystroke on DEL) has been detected is received from event *deleteNow*.

SRDL annotations, which are required to distinguish if the interaction returns to the initial state after reaching the final state (i.e. after both inputs were issued redundantly) or after falling back from an unstable node (i.e. after only one of the two inputs was issued). If the system returns to the initial state from an unstable state, the user-defined event *evtAsk4Confirm* will be raised (with the SRDL command *raise* shown in Table 5.1) as an indication that the deletion needs to be confirmed. The occurrence of *evtAsk4Confirm* will be noticed by another composite event, the event *deleteConf* (Figure 5.6b), which will be completed once the end user confirms the deletion. More concretely, the *node(2)* of the composite event *deleteConf* is annotated with a voice message for requiring the end user to confirm the deletion; the *node(3)*, which will be reached after the deletion is confirmed, is annotated with the function *rmvSelObjs*. The full code of this system is shown in Appendix C.3.

5.5.3 Types of feedback

As it was already proven by previous examples, Hasselt UIMS supports lexical and semantic feedback.

Lexical feedback is considered as echo produced in response to user actions [de Ruiter 88]. The keywords *play* and *speak* (Table 5.1), which command the built-in synthesizers of Hasselt UIMS, were introduced to enable Hasselt prototypes to provide lexical feedback. This type of feedback was used by our Put-That-There system, when the end user was responded with voice synthesis (*what?* and *where?*) as a way to making him aware that his inputs were being correctly detected. To provide this type of feedback, there is no need to take a look at the application variables, which contrasts with semantic feedback.

Semantic feedback is the type of feedback where determining the appropriate response to user actions requires specialized information about the variables and objects of the application [Myers 94]. Providing semantic feedback requires calling the back-end application, which in Hasselt, is achieved with the command *call*. In the *put-that-there* interaction, semantic feedback was used to inform the end user, through highlighting, that his target object was correctly selected. Unlike in the previous example of lexical feedback, here the UIMS cannot help the end user unless it receives support from the back-end application. This is because the UIMS ignores which object is under the mouse pointer (actually it ignores that the back-end application has a GUI hosting many objects) and therefore, it has to delegate this task to the back-end application that knows every detail about its hosted objects.

5.6 Limitations of SRDL

- CEDL allows accumulating event parameters in arrays, but the individual elements of these arrays cannot be referred to individually with SRDL code (e.g. CEDL allows *move.move*(*x*[], *y*[]), but SRDL does not recognize *assign: ini = x[1]*). In the current version, one can pass the arrays directly to the back-end applications or to use their aggregate data (e.g. their average, minimum, or maximum value, or the size of the array). The technical reason behind this limitation is the technique used to evaluate expressions, which was exposed in Section 5.4.3. The negative consequence of this issue is that one has to look for alternative ways to represent ideas that may be more naturally represented with arrays. For instance, sometimes one wants to check the final element of an array in order to determine the final position of a touch trajectory, which may be achieved by referring to *x[x.count]* and *y[y.count]*. Rather, on top of storing the trajectory points, Hasselt programmers have to declare two variables *lastX*, *lastY* that are to be overwritten for each *touch.move* event in an *assign* statement.
- The data types of the user-defined variables are limited to strings, integers, double, and datetime types and SRDL does not provide syntax to define user-defined datatypes, e.g. data structures or customized classes. This restriction was imposed to favor the simplicity of the SRDL syntax, which is now free of notations for declaring datatypes. But, this brings about the negative consequence that programmers may have to use several loose variables, which otherwise could have been packaged into one single data structure.
- It may happen that multiple nodes and links of a FSA have to be annotated with the same set of system responses or the same guard condition. For these cases, one has to repeat several blocks of SRDL statements for different elements of a FSA. In the SRDL code for the event *flickdown* (Equation 5.5), for instance, the same guard condition was repeated three times. This need of repeating code, which may discourage some programmers, can be eliminated by sweetening the SRDL syntax so that one can assign a block of system responses for multiple elements of a FSA at once. By putting more syntactic sugar in SRDL, one should be able to write statements such as *@node(1), link(3,mouse.up) do* to specify, in one single line, that the set of responses that follow must be associated to many elements, in this case, to both *node(1)* and *link(2,mouse.up)*.

- The SRDL is too closely coupled to the elements of the FSAs. This implies that if the programmer changes the CEDL definitions, the SRDL semantics could be broken, e.g. a certain node of a FSA may have a different meaning, if it still exists, after its reciprocal composite event is redefined. SRDL requires a more error-proof referencing solution.
- The guard conditions are associated with the nodes or links of the FSA. This implies that all the responses will be triggered or not depending on the value of their guard conditions. It would be more powerful if one single node or link could have several blocks of instructions, each with a distinct guard condition. So, instead of specifying something like ‘convey all these responses when this condition is true’, programmers could command Hasselt UIMS in the following way: ‘convey these responses if *condition*₁ is true, those other responses if *condition*₂ is true, and so forth’. Depending on the complexity of the case, multiple conditional system responses can be achieved by using utility functions, like *iff*, or, in the worst case, by replicating one composite event so that it has different guard conditions in each definition.

5.7 Summary

An interaction model describes the interplay between the end user and the system, the relation between the user actions and the system responses. Whereas the preceding chapter presented the Composite Event Definition Language (CEDL) as a notation for specifying sets of user actions as composite events, this chapter has studied the System Response Definition Language (SRDL), which allows specifying the system responses that must be conveyed in reaction to the (partial) detection of composite events. Both the CEDL and SRDL are required to create an interaction model.

SRDL allows binding multiple system responses to one single composite event. SRDL also permits to specify the moment when the system responses have to be conveyed. The type of responses one can specify include launching callback functions of back-end applications, activating the output synthesizers incorporated in Hasselt UIMS, or firing user-defined events. Furthermore, SRDL provides notations for declaring and maintaining user-defined (local or static) variables, which may eventually be interrogated in guard conditions, used as arguments when calling a back-end function, or used as parameters of user-defined events.

The chapter presented an extended discussion about the expressiveness of

CEDL and SRDL. It was shown that these languages working together allows specifying Complementary, Assigned, Redundant, and Equivalent events (CARE), repetitive events, and “did not occur” events. This expressiveness permits to create prototypes capable of providing different types of feedback (e.g. lexical and semantic), and supporting different types of interaction (e.g. single-stroke, multi-stroke, multitouch, and hand gesture) subjected to different types of constraints (e.g. spatial, temporal, and semantic).

The next chapter will describe the third and last member of the Hasselt family, the Human-Machine Dialog Definition Language (HMD2L), which is expected to create high-level dialog models on top of the interaction models created with CEDL and SRDL.

Chapter 6

HMD2L: Separating events from dialog model

With the CEDL and SRDL studied in the previous chapters, one can declare how the system must respond to a set of user-defined composite events. But this relation between composite events and system responses remains fixed, as if the system always stayed in the same context-of-use.

Dialog systems operate on different contexts-of-use and are capable of updating the context-of-use on the basis of the human-machine conversation [Traum 03]. The MMI Navigation system¹ incorporated in Audi vehicles is a good example of dialog system. This system has four distinguishable contexts-of-use, namely navigation, telephone, radio, and media mode, and allows users to seamlessly switch from one context-of-use to another through voice commands. In each context-of-use, the system awaits for a specific set of voice commands, which may not be recognizable in other contexts.

Although it is possible to create dialog systems with CEDL and SRDL alone, the maintenance of multiple contexts-of-use has to be implemented with low level code in the back-end application. This chapter investigates whether a declarative specification of human-machine dialogs may bring about advantages in the creation of multimodal dialog systems.

A final extension of Hasselt consisted of a dialog modeling language, called Human-Machine Dialog Definition Language (HMD2L). Its visual notation allows describing both the potential context-of-uses of a system and the interactions allowed in a given context-of-use. HMD2L was introduced into Hasselt by following a guideline given by Dumas et al. [Dumas 10]. They

¹<https://www.youtube.com/watch?v=Z2SJ3oCYy8Q>

suggest that the declarative language provided by a UIMS must be such that the declaration of multimodal events can be separated from the description of the human-machine dialog (Figure 6.1). We implemented this idea by putting HMD2L on top of CEDL and SRDL.

6.1 Hasselt's visual language: The Human-Machine Dialog Definition Language (HMD2L)

This section starts describing the place of HMD2L within a Hasselt project, the relation of HMD2L with its siblings CEDL and SRDL. It then provides details about how to create HMD2L models with the Hasselt's graphical editor. Finally, it discusses the differences between the finite state machines elaborated with HMD2L versus those auto-generated by Hasselt compilers.

6.1.1 HMD2L within Hasselt

HMD2L is a visual language that allows reusing the interactions' descriptions elaborated with CEDL and SRDL to create a high-level model, herein called dialog model. The dialog model represents all the different contexts-of-use of the intended dialog system as well as the events and conditions that cause the system's state transitions.

Whereas CEDL and SRDL code was written in a textual editor, HMD2L models are created in a separate, graphical editor from which composite events definitions can be referred to but not changed. Therefore, there is a separation between the definitions of the composite events and the definition of the multimodal dialog. This separation is more strict than the separation obtained with SMUIML [Dumas 10], where the specialized notations for composing events and for describing human-machine dialogs are used in the same visual editor. Hasselt, in contrast, allows both a logical and a physical separation.

The interactions presented in the previous chapters may be considered as trivial cases of human-machine dialogs: dialogs with one single context-of-use, where the system response to a given composite event always remains the same through the whole runtime.

As mentioned before, HMD2L is not mandatory to create dialog systems with Hasselt. One can alternatively create dialogs systems with CEDL and SRDL as long as some imperative code for dialog management is written in the back-end application. A comparison against this alternative solution will be shown at the end of this chapter.

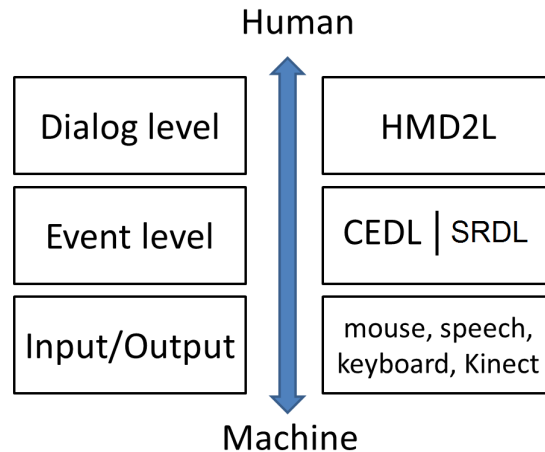


Figure 6.1: On the left side of the diagram, one can see the different levels of abstraction used by Dumas et al. [Dumas 10]. On the right side, one can see how our proposed tool follows the same framework: our HMD2L is at the dialog level whereas the CEDL and SRDL are at the events level.

6.1.2 HMD2L models

Assuming there is an interaction model defined with CEDL and SRDL code, one can use the graphical editor of Hasselt, shown in Figure 7.1, to create a dialog model with HMD2L.

HMD2L allows modeling human-machine dialogs as finite state machines (FSMs): the nodes of these FSMs represent the potential contexts-of-use where the dialog system may be; its arcs represent the transitions between different contexts-of-use. Arcs have to be referred to the composite events previously defined with the textual notations of Hasselt. At runtime, when a composite event is triggered, the intended dialog system may switch to a new context-of-use depending on the specifications of the HMD2L model.

HMD2L models are depicted in the graphical editor provided by Hasselt UIMS (Figure 7.1). These models can contain two types of nodes: one initial node and an arbitrary number of intermediate nodes. Programmers indicate which type of node has to be depicted through the toolbox buttons placed at the left side of the graphical editor. Arrows can be drawn so that they join any pair of nodes or a node with itself (i.e. a loop). Each arc can be annotated with a composite event name, a guard condition, and an assignment statement. These annotations can be made through a property window that pops up when an arc is double clicked. In this property window, the names

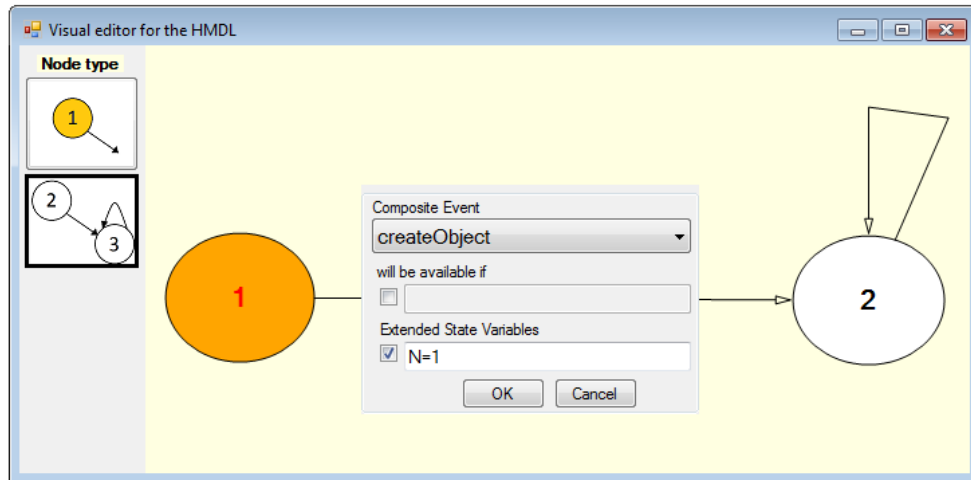


Figure 6.2: HMD2L editor. A property window is displayed when programmers click on an arrow of the user-defined FSM. The drop-down list contains all composite events previously defined with CEDL and SRDL. The textboxes can be optionally written with a guard condition (top) or with a list of assignments statements (bottom).

of the composite events previously declared with CEDL and SRDL are shown through a drop-down list, so one does not have to remember the composite event names or to switch from one editor to the other. Optionally, one can annotate guard conditions and assignment statements to an arc. The guard conditions must follow the same syntax rules described in Section 5.4.3, the same syntax used for the SRDL guard conditions. The assignment statement must be declared as if they were part of an *assign* statement, that is, by following the syntax commented in Section 5.1.2. The use of SRDL syntax in the HMD2L model is to reduce the cognitive effort required to learn this new language. Once the property window is closed, all the information is passed to the graphical model. The composite event name will appear in plain text; the guard condition, within square brackets; and the assignment statement, within curly braces.

6.1.3 Differences between auto-generated FSMs and HMD2L models

As shown in Table 6.1, there are some differences between the FSMs auto-generated from CEDL and SRDL and the user-defined FSMs elaborated with HMD2L models.

	Auto-generated FSMs	HMD2L models
Topology	Constrained Generated by fixed rules	Arbitrary User-defined
Final state	Mandatory	Not necessary
Types of events	Composite or atomic events	Only composite events
Dispatching order	Order of definition	Always first

Table 6.1: Differences between the FSMs generated by Hasselt UIMS from CEDL and SRDL code versus user-defined FSMs created with HMD2L.

Whereas the links of HMD2L can be placed in any direction and between any arbitrary pair of nodes, the links of the auto-generated FSMs always point downstream to the final state reflecting the fact that interactions have a well-defined, short lifecycle. In contrast, a dialog evolves more arbitrarily. In the aforementioned MMI vehicle system, for instance, the four contexts-of-use – namely navigation, telephone, radio, and media mode– can be visited in any order, depending on the whims of the user. The arbitrariness of a dialog is the reason why we allow Hasselt programmers to create FSMs of arbitrary structures with HMD2L.

Furthermore, the lifespan of a human-machine dialog is much longer than the duration of an interaction. Back to the MMI system, this may be running a specific radio station in response to a user’s command. In our interpretation, one interaction is already finished, but the dialog is still going on so that, at a later point in time, the user can interact with the system again to ask for another function. Due to the long lifespan of a dialog, which may even be “on” all the time, HMD2L models do not have to have final states.

The links of the FSMs auto-generated from CEDL and SRDL code may refer to atomic or composite events. In contrast, the links of a HMD2L model always refer to composite events, which reflects its higher-level of abstraction.

The HMD2L model will receive a special treatment by Hasselt UIMS. At runtime, once an event is withdrawn from the Event Queue (Section 3.2.3), it will be first dispatched to the HMD2L model. In this way, the dialog model will be the first to be notified about the occurrence of a composite event. Thus, the maintenance of the context-of-use will not be delayed.

6.2 Proof-of-concept application

To finish with the study of Hasselt UIMS and Hasselt, this section presents a comprehensive running example that shows that Hasselt can create prototypes that support multimodal, multitouch, and cross-device interaction.

6.2.1 Couch Potato. A Multimodal Video Player

Couch Potato is a multimodal system 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. The system can also react to the user's passive behavior.

Initially, Couch Potato shows a window with the name and version of the system (Figure 6.3a). When the user waves 'hello', Couch Potato displays an enumerated list of movie names (Figure 6.3b). The list can be scrolled through voice commands or touch gestures. By saying '*next*', '*previous*', or more flexible commands like '*four steps forward*' or '*ten steps backward*', the user can quickly select the desired video. Alternatively, the user can depict a number on the touch-sensitive screen of his mobile phone. In this case, Couch Potato will interpret this number as the index of the video to be selected. Both selection methods can be combined, if desired.

Once the intended video is selected, one can play it by flicking to the right on the screen of his mobile phone while pointing forward. Notice that to decode this command, Couch Potato has to combine two input modalities in order to realize that the user is (a) pointing towards the Kinect sensor and (b) flicking to the right, simultaneously. Similarly, as the video plays (Figure 6.3c), one can point out to the Kinect sensor and flick to the left or tap on the mobile phone in order to stop or pause the playback respectively. Right after stopping the video, the full list of videos will be shown again. The volume of the video can also be increased/decreased by flicking up/down. Unlike in regular remote controls, the volume can be changed linearly and exponentially. The more flicks in a short time interval, the higher the acceleration with which the volume will change. This feature may be appreciated by impatient users.

Couch Potato also reacts to the user's passive behavior. If the user leaves the room –read the Kinect's field of view– when no video is being played, the discreet Couch Potato will hide the list of videos the user was watching and show the initial window again. If the user leaves when a video is being played, nothing will happen –the user may be watching a rock concert or a football match and he still can listen to it from another room of the house.

6.2.2 Implementation

Back-end applications

The interactions supported by Couch Potato were declared with Hasselt, but its application-specific functionality was implemented using a Windows application, a DLL, and a mobile application so that Couch Potato can operate as described above.

The window application consists of a form hosting a video player and a listbox containing the names of the video files located in a specific directory. This window application implements both the presentation part and the application-specific functionality for controlling the video player.

The DLL contains a single-stroke recognizer. It receives a series of (x-y)-points and returns a string containing the name of a 0-9 digit or the string 'none' when no match is possible. Both the DLL and the window application were imported into Hasselt UIMS.

The mobile application that translates touch events into TUIO messages is the open-source TUIOdroid². TUIOdroid had to be configured so that its messages are sent to the same port that Hasselt UIMS listens on.

Human-machine dialog

The runtime behavior of Couch Potato is depicted in Figure 6.3d. Here the node 1 represents the state where the initial window is shown to the user, the node 2 is the state where the list of videos is displayed, and the node 3 represents the state where a video is being played. The appearance of Couch Potato in these three stages was shown in Figure 6.3a-c.

The code implementing some of the composite events used in the HMD2L model is discussed below. The full code can be seen in Appendix C.6.

Playing the selected video

Flicking to the right and extending the right hand forward are two events that when occurring simultaneously cause the chosen video to be played. The

²<https://code.google.com/p/tuiodroid>

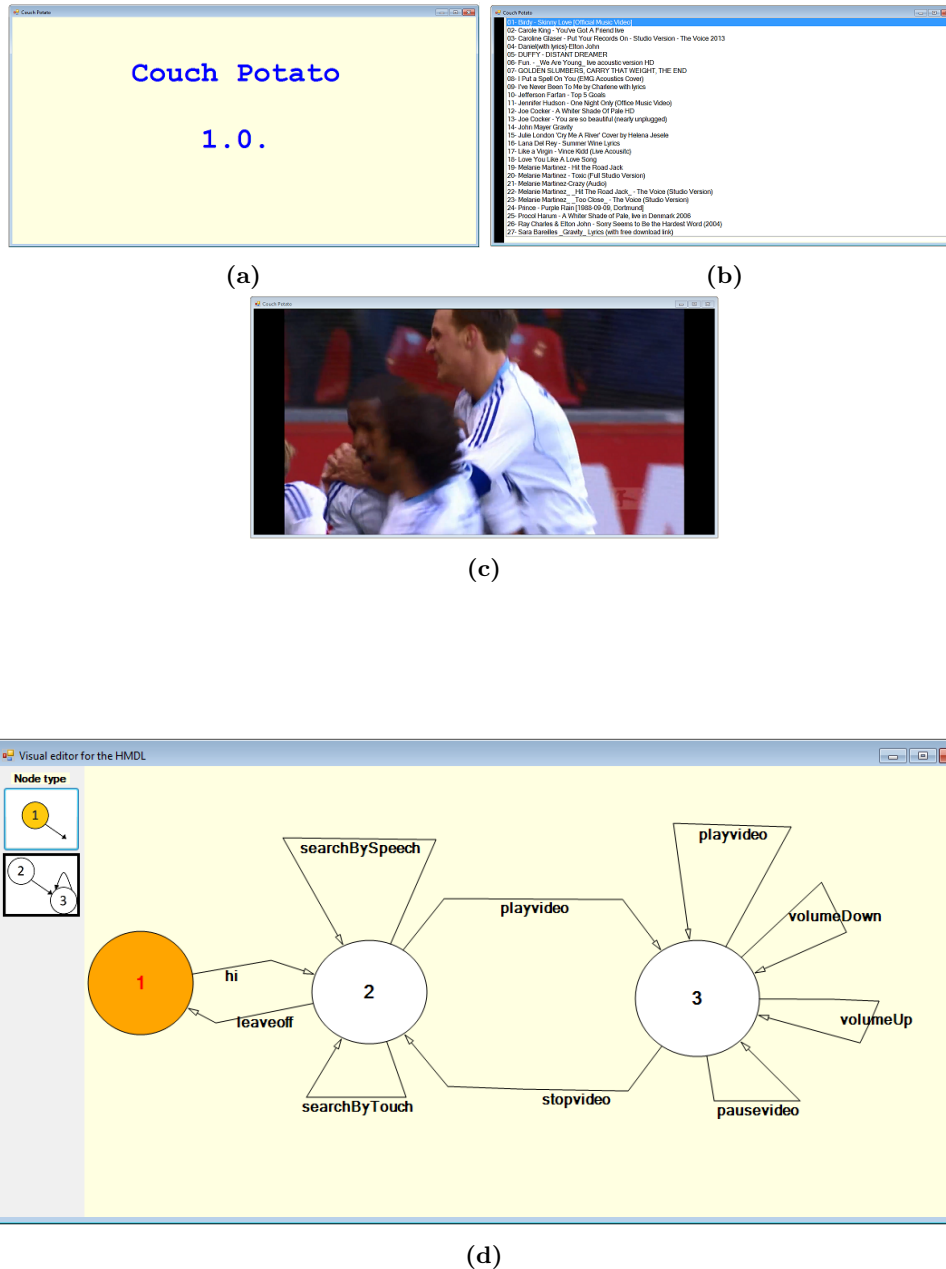


Figure 6.3: At the top, one can see Couch Potato operating in its three different contexts-of-use: (a) Initial mode, (b) Selection mode, and (c) Playback mode. These contexts-of-use as well as their interrelations are modeled in (d).

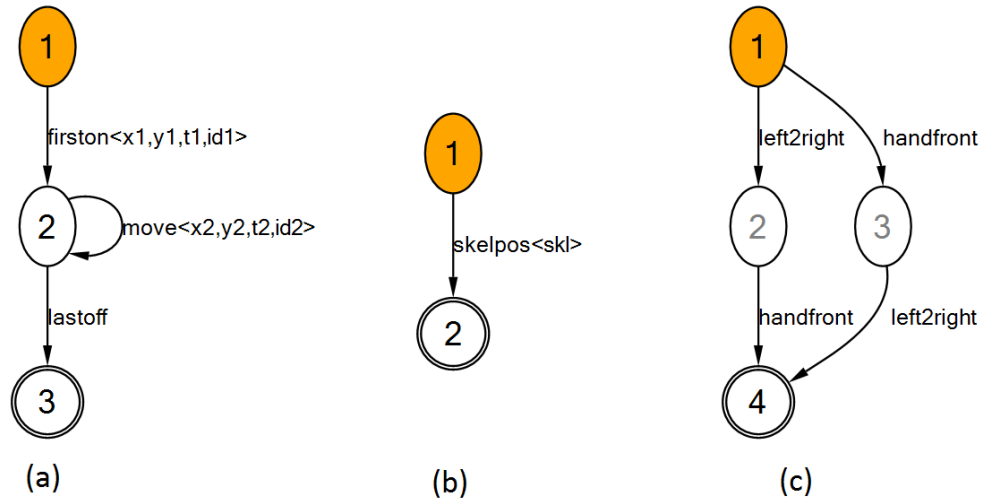


Figure 6.4: FSA-based representation of the events (a) *left2right* to be triggered when the user flicks to the right, (b) *rhandfront*, to be triggered when the user put his right hand forward, and (c) *playvideo* to be triggered when (a) and (b) co-occur, in which case Couch Potato will enter into playback mode.

involved events are declared as follows.

$$\begin{aligned}
 \text{event } \textit{left2right} &= \textit{tscreen.firston}\langle x1, y1, t1, id1 \rangle \\
 &\quad \textit{tscreen.move}\langle x2, y2, t2, id2 \rangle^*; \\
 &\quad \textit{tscreen.lastoff} \\
 \text{event } \textit{rhandfront} &= \textit{kinect.skelpos}\langle skl \rangle; \\
 \text{event } \textit{playvideo} &= \textit{ce.left2right} + \textit{ce.handfront}
 \end{aligned}
 \tag{6.1}$$

These events, whose visual representation is shown in Figure 6.4, are to be handled according to the following specification.

$$\begin{aligned}
 \text{wrt } \textit{ce.left2right} & \\
 &\quad @\textit{link}(2, \textit{tscreen.move}\langle x2, y2, t2, id2 \rangle) \textit{do} \\
 &\quad \quad \textit{when } id1 = id2; \\
 &\quad \textit{triggers when } x2 > x1 \textit{ and } \textit{abs}(y2 - y1) < 0.05 \\
 \text{wrt } \textit{ce.rhandfront} & \\
 &\quad \textit{triggers when } \textit{skl.HandRight.Z} < \textit{skl.Head.Z} - 0.35 \\
 \text{wrt } \textit{ce.playvideo} & \\
 &\quad @\textit{node}(4) \textit{do} \\
 &\quad \quad \textit{call} : \textit{WinMediaPlayer.Form1.play}();
 \end{aligned}
 \tag{6.2}$$

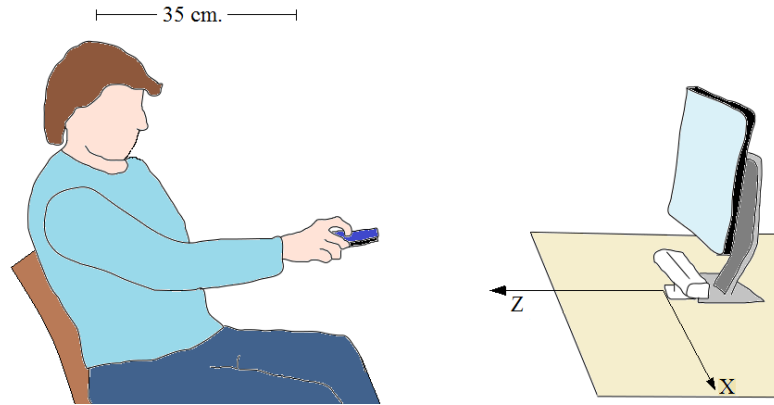


Figure 6.5: Coach Potato setup. The Z-axis of the Kinect’s coordinate space extends in the direction in which the sensor points. When the user extends his right hand, the z-coordinates of the head and the hand differs in at least 35 cm.

The composite event *left2right*, declared in Equation 6.1, occurs when the user flicks towards the right on his mobile phone screen. As specified in Equation 6.2, this event has two constraints. The semantic constraint $id1 = id2$ prevents other fingers to dirty the gesture started by the first one. The spatial constraint $x2 > x1$ and $abs(y2 - y1) < 0.05$ guarantees that the finger moves towards the right and almost in a straight line. Notice that $(x1, y1)$ was captured during the first touch, and $(x2, y2)$, during its last movement.

The composite event *rhandfront* occurs when the user is pointing forward, i.e. when his right hand is 35 cm. in front of his body (Figure 6.5).

The composite event *playvideo* occurs when the previously defined *left2right* and *rhandfront* are detected simultaneously. In this case, the function *play()* contained in the back-end application is launched.

Choosing a video through free-form touch gestures

By drawing the number N on his mobile phone screen, the user commands Couch Potato to select the N^{th} video of the list. Numbers can consist of multiple unistroke digits drawn in a quick succession.

Every time the user draws a digit, the composite event *digit* is triggered. A sequence of these events will cause the triggering of the event *searchByTouch*, which will finally cause Coach Potato to select a video. These two events are

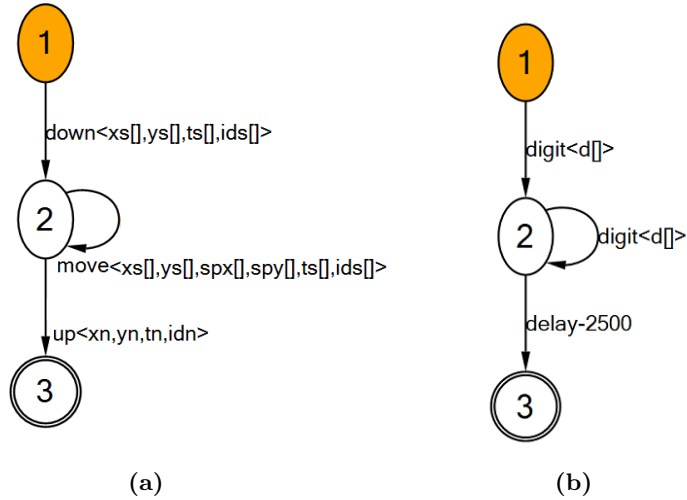


Figure 6.6: FSA-based representation of composite events (a) *digit*, triggered when a digit has been drawn, and (b) *searchByTouch*, triggered when several digits, whose names are accumulated in the array *d*, have been drawn.

defined below.

$$\begin{aligned}
 \text{event } \textit{digit}\langle \textit{getBestMatch} \rangle &= \textit{tscreen.down}\langle \textit{xs}[], \textit{ys}[], \textit{ts}[], \textit{ids}[] \rangle; \\
 &\textit{tscreen.move}\langle \textit{xs}[], \textit{ys}[], \textit{ts}[], \textit{ids}[] \rangle^*; \\
 &\textit{tscreen.up}\langle \textit{xn}, \textit{yn}, \textit{tn}, \textit{idn} \rangle \\
 \\
 \text{event } \textit{searchByTouch} &= \textit{ce.digit}\langle \textit{d}[] \rangle; \\
 &\textit{ce.digit}\langle \textit{d}[] \rangle^*; \\
 &\textit{delay-2500};
 \end{aligned}
 \tag{6.3}$$

The logic to handle the events *digit* and *searchByTouch*, whose visual representations are shown in Figure 6.6, is as follows.

$$\begin{aligned}
 \text{wrt } \textit{ce.digit}\langle \textit{getBestMatch} \rangle \\
 \quad \text{@link}(2, \textit{tscreen.move}\langle \textit{xn}, \textit{yn}, \textit{tn}, \textit{idn} \rangle) \textit{do} \\
 \quad \quad \text{call} : \textit{gest2d.utils.getBestMatch}(\textit{xs}, \textit{ys}, \textit{ts}); \\
 \quad \text{triggers when } \textit{getBestMatch} \langle \rangle \textit{'none'} \\
 \\
 \text{wrt } \textit{ce.searchByTouch} \\
 \quad \text{@node}(3) \textit{do} \\
 \quad \quad \text{call} : \textit{WinMediaPlayer.Form1.chooseVideo}(d);
 \end{aligned}
 \tag{6.4}$$

As shown in Equation 6.4, Hasselt UIMS exploits the method *getBestMatch* to identify the digit depicted by the user's stroke. This method belongs to a

DLL that was imported into Hasselt UIMS for the Couch Potato project. The method *getBestMatch* returns a string containing the name of the depicted digit or ‘none’ if the stroke did not match with any digit template. The arrays *xs[]*, *ys[]*, and *ts[]*, containing the time-ordered series of 2d points contacted by the gesture stroke are passed as input parameters to this method.

As to the event *searchByTouch*, this is composed out of a stream of digit events, e.g. *digit*⟨‘two’⟩, *digit*⟨‘six’⟩, ..., that finishes 2.5 seconds after the last received *digit* event. The composite event *searchByTouch* collects the parameters carried by its constituent events into an array, e.g. $d = [‘two’, ‘six’]$, that will be passed to the method *chooseVideo*. This method will select the video whose ordinal number is encoded in its input parameter, e.g. the 26th video.

Changing the volume

In playback mode, the user can increase/decrease the volume level by flicking up/down on his mobile phone screen. After performing N flicks in a quick succession, the volume level will change in N^2 levels. This rule takes into account the intensity with which the user wants to change the volume. Three *quick flicks* will not have the same effect as three flicks widely spaced in time. The former will increase the volume in 9 levels; the latter, in 3 levels only. In this way, Couch Potato users can change the volume linearly, which is the usual way, or exponentially, which is an innovation of Couch Potato.

The events *volumeUp*, *volumeDown*, seen in Figure 6.3d, have the same pattern as the event *repeflicks* discussed in Equation 5.6, in the previous chapter.

Waving hello

The *wave* gesture was defined as the sequence of two events: (1) *fromR2L*, to be triggered when the user moves his right hand from the right to the left, and (2) *fromL2R*, to be triggered when the user moves his right hand from the left to the right.

A first strategy to implement *fromR2L* was to collect three consecutive skeleton frames and check whether the x-position of the right hand decreases from frame to frame (cf. X-axis in Figure 6.5). Although this strategy is correct in theory, the imprecision of Kinect measurements causes poor event-pattern recognition in practice. Instead of defining the event *fromR2L* in terms of the imprecise Kinect measurements, the event *xrighthand* had to be declared (Equation 6.5). Whereas the built-in event *kinect.skelpos* carries the right hand’s position of the last frame fired by Kinect, the event *xrighthand*

was defined to carry a weighted average of the right hand's positions throughout the whole interaction. This is implemented as follows:

$$\begin{aligned}
 \text{event } x\text{righthand}\langle xpos \rangle &= \text{kinect.skelpos}\langle j \rangle \\
 \text{event } fromR2L &= \begin{array}{l} ce.x\text{righthand}\langle xpos1 \rangle \\ ce.x\text{righthand}\langle xpos2 \rangle \\ ce.x\text{righthand}\langle xpos3 \rangle \end{array} \quad (6.5)
 \end{aligned}$$

$$\begin{aligned}
 \text{wrt } ce.x\text{righthand}\langle xpos \rangle \\
 @node(0) \text{ do} \\
 \quad \text{assign : } prevxpos = 0; \\
 @link(1, \text{kinect.skelpos}\langle j \rangle) \text{ do} \\
 \quad \text{assign : } xpos = 0.8 * j.\text{HandRight}.X + 0.2 * prevxpos; \\
 @node(2) \text{ do} \\
 \quad \text{assign : } prevxpos = xpos; \quad (6.6)
 \end{aligned}$$

$$\begin{aligned}
 \text{wrt } ce.fromR2L \\
 \text{triggers when } xpos1 > xpos2 \text{ and } xpos2 > xpos3
 \end{aligned}$$

In Equation 6.6, the variable $xpos$ carried by the composite event $xrighthand$ contains the x-position of the right hand after passing through an exponential smoothing filter (see Appendix A.1). Since the variable $prevxpos$ is attached to the special node 'zero', it will retain its value throughout the whole runtime; it condenses historical information. Normally, variables are clear up right upon a composite event is detected.

Finally, the event $fromR2L$ verifies that the value $xpos$ decreases in three consecutive $xrighthand$ events. The fundamental difference with the aforementioned original strategy is that the x-positions are now smoothed by a filter, which produces less error recognitions in practice.

Analogously, an event $fromL2R$ declares the hand's movement to the right as one composite event that must be detected by Hasselt UIMS. The waving event was defined as simply $fromR2L ; fromL2R$ (Figure 6.7c) restrained to a time interval of 2.5 seconds.

6.2.3 Passive inputs

The event $leaveoff$ depicted as an outgoing link in the node 2 of Figure 6.3d, triggers in response to the built-in event $kinect.useroff$ or to the speech input $speech.bye$ included in the recognition grammar file used for this project, i.e. $\text{event } leaveoff = \text{kinect.useroff} | \text{speech.bye}$. The fact that $kinect.useroff$ is launched by Hasselt UIMS when the user disappears from the Kinect's field of view means that $leaveoff$ can be triggered passively or actively.

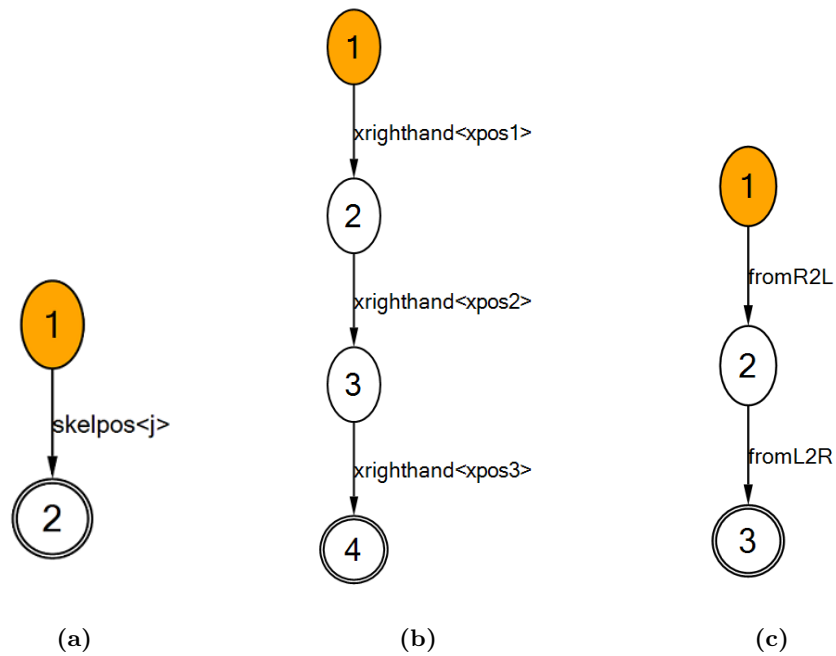


Figure 6.7: FSA-based representation of the composite events (a) *xriighthand*, which filters the information provided by MS Kinect, (b) *fromR2L*, to be triggered when the user's right hand moves to the left in three consecutive frames, and (c) *wave*, to be triggered when the wave gesture occurs.

An interesting possibility is to change Coach Potato so that upon entering into Selection mode, it uses the distance between the user and the Kinect to set the font size of the items shown in Figure 6.3, b. This would require to change the event *rhandfront* (Equation 6.1) so that it can carry one parameter storing the z-coordinate of the user's head. This parameter can be later passed to the back-end application that controls the video player.

Another example of passive inputs was already given in the previous chapter when a prototype for supervising the training session of a user was discussed (Figure 5.4 on page 89). There the end user is not commanding the prototype explicitly; rather, the prototype is an assistant, that can proactively encourage the end user to keep up the pace whenever he starts to slow down.

6.3 Limitations of HMD2L

- There can only be one HMD2L model per project, which may raise difficulties when modeling real-world systems. It was found that many industrial applications may be modeled with 50-1000 transitions [Jensen 97], which, with our HMD2L, will degenerate into a spaghetti of wires. The complexity of such chaotic state diagrams may be alleviated, at least to some extent, by the fact that each arc of HMD2L model represents a FSM itself, so there is some implicit hierarchy of FSMs even in one single HMD2L model.
- Creating and maintaining HMD2L models is not as fast as writing code. The precision required to wire the diagram and the persistent habit of rearranging the layout of a visual model almost every time a new node is added or an existing one is removed slows down the process of maintaining visual models. The concern for the aesthetics is higher when maintaining visual model than when maintaining textual models. This is something we observed during the user studies to be discussed in a later chapter.

6.4 Summary

This chapter has brought Part I to a close. The study of the Composite Event Definition Language (CEDL) and the System Response Definition Language (SRDL), studied in the preceding chapters, was herein complemented with the study of a visual notation called Human-Machine Dialog Definition Language (HMD2L).

HMD2L allows describing a dialog model on top of the interaction model created with CEDL and SRDL. The dialog model is depicted, in a graphical editor, as a finite state machine (FSM) whose arcs refer to previously declared composite events. This model represents the potential context-of-uses of the intended system and the interactions allowed in a given context-of-use. A Hasselt project can dispensed with a dialog model, in which case, the user actions will be responded in the same way through the whole runtime. The syntax of HMD2L and the differences between the user-defined FSMs created with HMD2L and the FSMs auto-generated from CEDL and SRDL code have been discussed. At the end of the chapter, a comprehensive example that involves all members of the Hasselt family was presented. Coach Potato, a video player controlled by remote, multimodal signals, proves that on top of the old-fashioned speak-and-mouse interactions described in the first chapters, Hasselt can also combine more ‘modern’ modalities such as touch and body movements. As proven with Couch Potato, Hasselt allows prototyping different interaction styles, namely multimodal, multitouch, and cross-device interactions; these interactions do not only involve active but passive modalities too. Despite its declarative nature, relatively complex tasks, such as joint filtering, a technique for smoothing imprecise input data, can be described with Hasselt without need of invoking low level back-end code.

The following Part II will concentrate on the evaluation of the proposed tools from two different viewpoints: code inspection and user studies.

Part II

Assessment of Hasselt

Chapter 7

Code comparison of two different paradigms

Whereas Part I described how to create interaction models and dialog models with Hasselt, Part II will present a two-fold evaluation to assess the code complexity and programming efficiency of Hasselt.

This chapter concentrates on code complexity; it investigates whether the difficult-to-read callback soup resulted from implementing multimodal interactions with event-callback code can be simplified by the use of Hasselt. For this purpose, a set of prototypes that were implemented with both Hasselt and event-callback code were subjected to code inspection. The comparisons are grouped into two categories.

The first category of comparisons is intended to investigate whether Hasselt (CEDL/SRDL) brings advantages with respect to event-callback code when it comes to implement interaction models. The composite event-based models obtained with Hasselt and the models obtained with event-driven languages are compared twice: for a multimodal interaction and for a multitouch interaction. For each interaction style, the resulting source codes are compared based on many dimensions, some of which, were taken from the Green's Cognitive Dimensions [Green 89].

The second category of comparisons aims at identifying whether Hasselt (HMD2L) simplifies the description of dialog models. A multimodal dialog system will be described with the visual model of Hasselt and with event-callback code. To restrict the comparison to the high-level dialog model, it will be assumed that in both cases there is a common interaction model over which the dialog model has to be specified.

7.1 Cognitive Dimensions

Cognitive Dimensions of Notations (CDs) is a theoretical framework for describing the usability of notational systems [Green 89]. The CD sets out a vocabulary of terms designed to capture and discuss about the cognitively-relevant aspects of a notation as well as the tradeoffs among these aspects. There are 14 dimensions in the CD framework, but we only used those that can be discussed with a moderate level of precision. Other dimensions, in our opinion, are too difficult-to-measure and any conclusion drawn about them may be too subjective (e.g. one dimension is expected to measure the extent to which the notation make things complex or difficult to work out in the programmer's head, but nothing is said about how to measure that).

The cognitive dimensions considered in our study are as follows:

- *Viscosity* refers to the resistance to change a program when a given language is used. A viscous system needs many user actions to accomplish one goal.
- *Diffuseness* refers to the fact that some notations can be annoyingly long-winded, or occupy too much valuable space within a display area. Diffuse languages reduce the available working area.
- A program is said to have *hidden dependencies* when the full understanding of its semantics requires unveiling a piece of code that is hidden by the programming environment.
- *Abstract gradient* refers to the minimum and maximum level of abstraction allowed by a language, i.e. to the possibility of encapsulating and reusing fragments of code.

For the sake of completeness, the other 10 dimensions are: visibility, premature commitment, role-expressiveness, error-proneness, secondary notations, closeness of mapping, hard mental operations, provisionality, and progressive evaluation. Readers may refer to [Green 89] for the definitions of these dimensions and to [Green 96] for an example of its applications for the evaluation of visual languages.

Our comparative study also includes other dimensions that we defined ourselves and can be measured more precisely by inspecting source code.

7.2 Interaction models

This section compares the code required by Hasselt and by C# to describe the multimodal interaction *put-that-there* and the multitouch gesture *pinch*.

7.2.1 Code inspecting a multimodal interaction

The multimodal interaction to be implemented is the *put-that-there* described in previous chapters: End users are allowed to move virtual objects by speaking ‘*put that there*’ while disambiguating the pronouns ‘*that*’ and ‘*there*’ with mouse clicks on the target object and on its intended position, respectively.

Put-that-there with an event language

When implementing the interaction *put-that-there* with an event language, one has to implement a presentation part (a windows form hosting several objects as shown in Figure 4.3a) and to write the code for handling the interaction. When using those Integrated Development Environments (IDEs) that used to accompany mainstream event languages, the presentation part can be created with an interface builder. This presentation part has to be supported by the code outlined in Algorithm 3, which was reverse engineered from a C# program we created ourselves.

In Algorithm 3, the boolean variables, declared in line **1**, are used to check the occurrence of relevant events. The datetime variables, shown in line **2**, are used to timestamp the event notifications. The variables $x1, y1, x2$, and $y2$, shown in line 3, are used to store the (x-y)-points on which the mouse was clicked. We distinguish all the variables according to the role they play in the program. Whereas the event parameters (line 3) carry information brought by the input devices, the state variables (lines 1-2) are meant to encode the interaction state.

The method `HASPUTTHATTHEREOCCURRED`, declared in line **13**, is used to combine the information carried by the boolean and datetime variables (i.e. state variables), which is necessary to determine whether the interaction *put-that-there* has been completed. Finally, the method `PUTTHATTHERE`, shown in line **18**, is intended for moving the selected object. Only a small portion of the code (method `PUTTHATTHERE`) implements the functionality that the end user wants to invoke, i.e. to move the selected object. The majority of the code is in charge of detecting the occurrence of the *put-that-there*.

Algorithm 3 Put-that-there interaction with event-callback code

```

1: boolean bPut, bThat, bThere, bClick1, bClick2           ▷ State variables
2: datetime dPut, dThat, dThere, dClick1, dClick2       ▷ State variables
3: int x1, y1, x2, y2                                   ▷ Event parameters

4: procedure SPEECHRECOGNIZED(e)                       ▷ Detecting speech inputs
5:   if e.Text = 'put' then bPut ← true, dPut ← Now()
6:   if e.Text = 'that' then bThat ← true, dThat ← Now()
7:   if e.Text = 'there' then bThere ← true, dThere ← Now()
8:   if HasPutThatThereOccurred() then PutThatThere(x1, y1, x2, y2)

9: procedure MOUSECLICK(e)                             ▷ Detecting mouse clicks
10:  if dClick1 is null then x1 ← e.X, y1 ← e.Y, dClick1 ← Now()
11:  else x2 ← e.X, y2 ← e.Y, dClick2 ← Now()
12:  if HasPutThatThereOccurred() then PutThatThere(x1, y1, x2, y2)

13: procedure HASPUTTHATTHEREOCCURRED()                 ▷ Detecting event pattern
14:  if bPut & bThat & bThere & bClick1 & bClick2 then
15:    if dPut < dThat < dThere then
16:      if dThat - dClick1 < 500 & dThere - dClick2 < 500 then
17:        return true

18: procedure PUTTHATTHERE(x1, y1, x2, y2)             ▷ Handling event pattern
19:  for all o ∈ Controls do
20:    if o.contains(x1, y1) then o.Location = new Point(x2, y2)

```

7.1: Put-that-there interaction with Hasselt

```

1: event moveObject = speech.put ;
2:   speech.that + mouse.down(x1, y1) ;
3:   speech.there + mouse.down(x2, y2)

```

(7.1)

```

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

```

Put-that-there with Hasselt

In order to create the *put-that-there* interaction with Hasselt, we first had to implement an externally developed, back-end application including the presentation part and the method `PUTTHATTHERE`. The back-end application was developed with C# and imported into Hasselt UIMS. As in the previous case, its presentation part consists of a windows form hosting button boxes. Its method `PUTTHATTHERE` referred to in the line 6 of the code snippet 7.1 is exactly the same as the method shown in line 18 of Algorithm 3. As to the interaction code, describing the *put-that-there* interaction with Hasselt requires programmers (1) to define the composite event *moveObject* as the stream ‘*put that (click) there (click)*’, as shown in code snippet 7.1, lines 1-3, and (2) to bind *moveObject* to the method `PUTTHATTHERE`, as shown in lines 4-6.

Comparing the source codes for the *put-that-there*

The event-callback code, sketched in Algorithm 3, will be compared against the Hasselt code, shown in code snippet 7.1, based on the dimensions shown in Table 7.1. The data contained in the table will be discussed below.

1. The programming effort required to build the presentation part and to implement the application-specific functionality, encoded in the method `PUTTHATTHERE`, is the same regardless of whether the interaction will be later described with Hasselt or with event-callback code.
2. The variables $x1$, $x2$, $y1$, and $y1$, shown in both Algorithm 3 and code snippet 7.1, serve the same purpose in both cases: storing the coordinates of the two mouse clicks required for the *put-that-there* interaction.
3. When using event-callback code, one has to maintain several state variables (Algorithm 3, lines 1-2) that altogether encode the interaction state. Hasselt code does not require such state variables since the interaction state is internally tracked by Hasselt UIMS.
4. Hasselt code does not include conditional statements to check whether the method `PUTTHATTHERE` has to be launched or not. When the programmer refers to *node(8)* (snippet 7.1, line 5), he is defining, in a declarative fashion, the moment when `PUTTHATTHERE` has to be launched. In contrast, with event-callback code, one has to interrogate several state variables –see method `HASPUTTHATTHERE OCCURRED` in Algorithm 3– in order to determine if it is the right time to call the `PUTTHATTHERE` method.

Dimension	Hasselt	Event-callback code
1. Presentation and application code	Same amount of programming effort	
2. Number of event parameters	Same amount of event parameters	
3. Number of state variables	None	Many
4. To-fire-or-not-to-fire condition	Not required	Required
5. Scope of event parameters	Scoped by composite event	Scoped by event handler
6. Verification of time constraints	Automatic	Manual timers/time variables
7. Ease of fine-tuning	Limited	High
8. De-synchronization problem	Naturally robust	Robustness requires programming effort
9. Space occupied by code (Diffuseness)	Concise notation	Wordy (compared to Hasselt)
10. Effort for changing interaction code (Viscosity)	Low for adding partial feedback	Moderate for partial feedback
11. Hidden dependencies	Yes	No

Table 7.1: Dimensions used for comparing Hasselt with equivalent event-callback code based on the two equivalent implementations of the multimodal interaction *put-that-there*, i.e. Algorithm 3 and code snippet 7.1.

5. When using event languages, the *put-that-there* interaction triggers two event handlers (Algorithm 3, lines 4, 9). Important information about the interaction, e.g. spoken words and (x,y)-positions of mouse clicks, is carried by the parameters of the event handlers or calculated inside the handlers, e.g. timestamps or interaction state flags. But this information is only visible from within the handlers although it often has to be put together, for instance, to determine the interaction state (lines 14-16). One way to make this information visible to all the functions involved in the management of the *put-that-there* is by putting it in a wider scope, as global variables (lines 5-7, 10-11). The trick of moving information that has local scope to a global scope is not needed with Hasselt. Hasselt uses a new type of scope that is not as wide as the global scope, in which global variables live, and not as reduced as the local scope, in which event parameters exist. Hasselt variables are scoped by composite event. Every event parameter and user-defined variables created within a composite event will be visible throughout the whole lifecycle of that composite event, but not visible from other composite events. The variables *x1* and *y1* (Code snippet 7.1) containing the position of the first click will be still visible, without being overwritten, even after the second click arrives. At the same time, these variables cannot be referred to from other composite events, which may thus use the same variable names for other purposes.
6. With event-callback code, one requires datetime variables (Algorithm 3, line 2) in order to verify (1) whether the selection and movement of the object have been performed in this sequential order (Algorithm 3, line 15), and (2) whether the speech inputs and mouse clicks are issued simultaneously –read within a time frame of 500 ms– (Algorithm 3, line 16). In Hasselt, the symbols FOLLOWED BY (;) and AND (+) allow programmers to specify sequential and simultaneous relations among events, in a declarative manner. We are aware that the datetime variables used in Algorithm 3 can be dispensed by using timers¹. But this latter option would introduce more event handlers in which the state variables had to be maintained.
7. Mainstream event languages such as C# or Java allow more fine-tuning. The time interval within which two inputs are considered simultaneous (i.e. the 500 ms shown in Algorithm 3, line 16) can be changed to the

¹<https://msdn.microsoft.com/en-us/library/system.timers.timer%28v=vs.110%29.aspx>

whims of the programmer, and it may be different for each couple of parallel inputs (e.g. it is possible to give to the selection of the object a time interval of 500 ms, and another of 600 ms, to the selection of the object's new position). In Hasselt, in contrast, the length of such time intervals can also be set to any value through a configuration file (Section 3.1.3), but it will be the same for every pair of events joined by the AND (+) operator. This prevents from specialized treatments for those input modalities whose recognition is too slow.

8. Algorithm 3 works fine in an ideal scenario. Some situations presented during the interaction may cause that the temporal constraints, specified in lines 15-16, will never be met, and thus the `PUTTHAT THERE` method will never be called (e.g. when the new position of an already selected object could not be established within a time threshold of 500 ms due to a speech recognition error). Hasselt handles simultaneity in a more robust way. If only one input arrives in a moment when two simultaneous inputs were expected, Hasselt UIMS will wait for some time before acknowledging the end user that he has a new chance to issue both inputs again (Section 5.4.1). This is a remedy against the stagnation of the interaction tracking process, which was, one of the limitations of HephaisTK [Dumas 10], as we were told by Dumas when he visited our research lab.
9. A final, evident observation is that Hasselt code is shorter than the equivalent event-callback code. The same functionality encoded in Algorithm 3, lines 1-17 (i.e. to invoke the method `PUTTHAT THERE` in the right moment of the interaction) is concisely specified the code snippet 7.1, lines 5-6.
10. Adapting the Hasselt code so that the selected object can be highlighted only requires binding some kind of `HIGHLIGHT OBJECT` method to the *node*(5). In contrast, the same change would require to implement conditional statements, similar to those in lines 14-16 of Algorithm 3, when using event language.
11. The SRDL code depends on the finite state automata generated from the composite events: the system responses are associated to the nodes and links of these automata. However, due to the design of its code editors, Hasselt UIMS only displays one automaton at a time; the others remain hidden. On the other hand, the event-callback code is self-contained, it

can be fully interpreted without need of additional references; it does not have dependencies.

7.2.2 Code inspecting a multitouch interaction

By following the structure of the preceding section, here we show the implementation of a touch interaction with both event-callback code and Hasselt. The interaction to be implemented consists on the user placing two fingers on a touch-sensitive screen and moving them closer together/further apart in order to see an online contraction/enlargement of the screen content.

To avoid to unintentionally disfavor event languages in our comparison, this section will present the algorithm elaborated by Kin et al. to discuss the difficulties of implementing a two-finger rotation gesture with support of an event-callback framework [Kin 12b]. For the same reason, we will also try to reproduce the opinions of these authors as accurately as possible.

Pinch gesture with event-callback code

The event-callback code required to handle the two-finger pinch gesture is shown in Algorithm 4. The only difference with the original of Kin et al. is that our algorithm explicitly shows that before resizing the screen content, the system has to first verify that the distance between the touches has changed. Concretely, the lines 12-14 were collapsed in one single line in [Kin 12b].

The same analysis made in [Kin 12b] for the rotation gesture applies for the pinch gesture too. As Kin et al. state: “As the user interacts with a multitouch surface, the framework continuously generates a stream of low-level touch events corresponding to *touch-down*, *touch-move* and *touch-up*. To define a new gesture, the developer must implement one callback for each of these events, *touchesBegan()*, *touchesMoved()* and *touchesEnded()*. It is the developer’s responsibility to track the state of the gesture across the different callbacks.”

Similar to the multimodal case, tracking the interaction state (i.e. the state of the gesture) requires updating and interrogating a set of global variables (Algorithm 4, lines 1-2) across different event handlers (lines 3, 9, and 13). Once again, many lines of code are intended to detect the sequence of events corresponding to the pinch gesture; the application-specific functionality the end user wants to invoke (i.e. resizing the content screen) is implemented in line 14.

In the given pseudocode, the state variables are used to count the number of touches and to track the stage of the two-touch pinch gesture. Although

there are only two state variables, “these adds significant complexity to the recognition code, even for simple gestures”. As Kin et al. mention, maintaining multitouch gestures is not possible “without fully understanding how the different callback functions work together”. “As the number of gestures grows” –the authors finished–, “understanding how they all work together and managing all the possible gesture states becomes more and more difficult”.

Pinch gesture with Hasselt

With Hasselt UIMS, one has to implement an external application containing the presentation part and the function for resizing the touchscreen content. This latter function is equivalent to the one collapsed in Algorithm 4, line 14. The attribute *avgDistanceToFirst*, referred to in code snippet 7.2, returns the distance between touches. This distance is calculated internally by Hasselt UIMS whereas with event languages, it requires user-defined code, as the code collapsed in Algorithm 4, line 12. In an efficient implementation, the resizing should only be performed if the distance between touches has changed. This verification is made in the line 11 of the Hasselt code (and has its equivalent counterpart in the line 13 of Algorithm 4).

7.2: Pinch gesture with Hasselt

```

1:  event zoom = tscreen.move(x0, y0, spX, spY, t0, id0, scr0);
2:          tscreen.move(x1, y1, spX, spY, t1, id1, scr1)*;
3:          tscreen.lastoff

4:  wrt ce.zoom
5:  @link(1, tscreen.move(x0, y0, spX, spY, t0, id0, scr0)) do
6:    assign : prevDist = scr0.avgDistanceToFirst;
7:    when scr0.numberOfFingers = 2;
8:    @link(2, tscreen.move(x1, y1, spX, spY, t1, id1, scr1)) do
9:      call : rotation.Form1.resize(scr1.avgDistanceToFirst);
10:     assign : prevDist = scr1.avgDistanceToFirst;
11:     when scr0.numberOfFingers = 2 and
        scr1.avgDistanceToFirst <> prevDist;

```

Comparing the source codes for the pinch gesture

Two equivalent implementations of the pinch gesture will be compared. The data contained in Table 7.2 is discussed below.

Algorithm 4 Pinch gesture with event-callback code, as shown in [Kin 12b]

```
1: boolean touchCount = 0                                ▷ State variables
2: state gestureState = null                            ▷ State variables

3: procedure TOUCHESBEGAN( )
4:   touchCount = touchCount + 1
5:   if touchCount = 2 then
6:     gestureState = began
7:   else if touchCount > 2 then
8:     gestureState = failed

9: procedure TOUCHESMOVED( )
10:  if touchCount = 2 and gestureState! = failed then
11:    gestureState = continue
12:    ## a. calculate distance between touches ##
13:    ## b. compare current vs. previous distance ##
14:    ## c. resize the screen content ##

15: procedure TOUCHESENDED()
16:   touchCount = touchCount - 1
17:   if touchCount = 0 and gestureState! = failed then
18:     gestureState = ended
```

Dimension	Hasselt	Event-callback code
1. Presentation and application code	Same amount of programming effort	
2. Number of event parameters	Many primitive variables	One TouchEventArgs object
3. Number of state variables	None	Few
4. To-fire-or-not-to-fire condition	Required	Required
5. Scope of event parameters	Not applicable	Not applicable
6. Verification of time constraints	Not applicable	Not applicable
7. Ease of fine-tuning	Not applicable	Not applicable
8. De-synchronization problem	Not applicable	Not applicable
9. Space occupied by code (Diffuseness)	Concise but difficult-to-read	Wordy (compared to Hasselt)
10. Effort for changing interaction code (Viscosity)	Low for the case of adding fingers	
11. Hidden dependencies	Yes	No

Table 7.2: Dimensions used for comparing Hasselt with equivalent event-callback code based on the implementation of the pinch gesture for resizing screen content.

1. The programming effort made for the presentation part and the application part is equivalent with both languages.
2. Hasselt code exhibits several event parameters containing information about the touches (e.g. position, speed, etc.) whereas event-callback code only uses two state variables, but not event parameters. This is due to the fact that the pseudocode copied from [Kin 12b] is not explicitly showing the parameters of the touch events. Beyond this, we admit that Hasselt could do better if the list of 7 parameters right after the *touch-move* event (code snippet 7.2, lines 1-2) would have been encapsulated into one single object. These many parameters hinder the readability of Hasselt code when *touch-move* events are involved.
3. In Algorithm 4, the variables *gestureState* and *touchCount* have to be updated in line with the ever-changing gesture state. State variables are not needed when using Hasselt since the gesture state is tracked automatically by Hasselt UIMS.
4. With event languages, the same as with Hasselt, there is a conditional statement that determines when to resize the screen content. The aforementioned condition is hidden in the line 13 of Algorithm 4, whereas, in Hasselt, an equivalent verification is made in the line 11 of the code snippet 7.2.
5. Algorithm 4 was written at a too high level of abstraction (in [Kin 12b]) and it does not show the event parameters of the touch events. Therefore, it is fair not to make any comparison about this respect.
6. No hard time constraints are imposed in this example. Any of the two touches performing the pinch gesture can arrive before the other.
7. The enhanced fine-tuning allowed by event languages is not appreciated here as in the previous multimodal case.
8. The touches used to pinch may arrive at the same time or at different times. Any of these two cases is to be considered a problem.
9. Hasselt code is shorter than event-callback code –even if you just compare the real Hasselt code (snippet 7.2) versus the simplified pseudocode delineated in Algorithm 4.
10. Hasselt code can be adapted to a three or four-touch pinch gesture just by changing the condition shown in code snippet 7.2, lines 7,11 (e.g.

scr0.numberOfFingers = 3). The same upgrade with event-callback code would mainly require a modification in the calculation of the distance between touches (Algorithm 4, line 12).

11. The same discussion of hidden dependencies made in the previous section applies here. The inability of observing all the finite state automata at once is caused by the Hasselt's code editor and will be present in every interaction model.

7.3 Dialog models

The present section compares one HMD2L model with equivalent event callback for the case of a dialog-based Put-That-There system.

In order to constraint the comparison to the high-level dialog model, it will be assumed that there is already a baseline system from which a dialog system has to be built on. The baseline system supports three interactions: it allows users to create virtual objects on a canvas by issuing the command '*create green box here*'. The virtual objects can be moved with the command '*put that there*'. And the canvas, which is initially empty, can be cleared up with the voice command '*remove objects*'. The behavior of this baseline system does not depend on the context-of-use: this system can create, move, or remove objects whenever the end user asks it to do so.

What will be evaluated here is how much effort has to be involved to extend such a system with dialog management capabilities, to upgrade it to a dialog system with the following behavior: (1) the boxes can only be moved if there are more than three of them on the canvas, and (2) the canvas can only be cleared up after the displacement of at least one object. Neither in the baseline system, nor in the required dialog system, is partial feedback generated during the interactions.

7.3.1 Implementation of the baseline system

When implemented with Hasselt, the baseline system consists of three composite events, namely *createObject*, *putThatThere*, and *removeObjects*, which are bound to the functions `CREATEOBJECT(COLOR,X,Y)`, `PUTTHATTHERE(X1,Y1,X2,Y2)`, and `REMOVEOBJECTS()` of a back-end application. The events *createObject*, *putThatThere*, and *removeObjects* fire whenever the end user asks the system to create, move, or delete objects, respectively.

When implemented with event-callback code, the baseline system includes a supervisory mechanism able to invoke the functions `CREATEOBJECT(COLOR,`

x, y), PUTTHAT THERE(x_1, y_1, x_2, y_2), REMOVEOBJECTS() in a timely manner and in response to the appropriate sets of user inputs.

Dialog-based Put-that-there system with an event language

One of the simplest ways to augment the code of the baseline system in order to convert it into the required dialog system is by adding the conditional statements shown in Algorithm 5.

The ability to manage human-machine dialogs is implemented by maintaining two global variables, *state* and *N*, which must be interrogated before launching a system response (i.e. creation, movement, or deletion of objects). In this way, the dialog system can be sure that the responses are going to be conveyed only in the right context-of-use and under the right conditions, which are encoded in *state* and *N*.

The variable *state* can accept three different values, each representing one of the three relevant contexts-of-use that the system has to distinguish. The variable *state* is assigned with 1 whenever the canvas is empty; the value 2 is to represent that there are virtual objects on the canvas but none of them has yet been moved; and the value 3 is to represent that some of the objects has already been moved. On the other hand, the variable *N* stores the number of objects on the canvas, which is relevant to check whether a transition from *state* = 2 to *state* = 3 is valid. These variables determine whether the system must respond and, at the same time, every system response alters the values of these variables.

Dialog-based Put-That-There system with Hasselt

With Hasselt, one can extend the baseline system into the required dialog-based Put-That-There system by depicting the HMD2L model shown in Figure 7.1. Here the node labelled as 1 represents the state where the canvas is empty; the node 2 represents the state where there is at least one object on the canvas; and the node 3, the state where at least one object has been moved. These states have the same interpretation of those states considered above, when the dialog was modeled with event-callback code.

According to the HMD2L model, the system moves from the initial state 1 to state 2 upon the creation of the first object. It also moves from state 2 to state 3 after the first displacement of an object. The variable *N* is used (a) to count the number of objects in the form –when this is relevant–, and (b) to condition the displacement of objects, which should only be possible if there

Algorithm 5 Event-callback code for dialog-based Put-That-There system

```
1: global: state=1, N=0

2: procedure CREATEOBJECT(color, x, y)
3:   if state = 1 then
4:     N = 1, state = 2
5:   else if state = 2 then
6:     N = N + 1
7:
8:     # code to create an object on (x,y) #
9:
10: end procedure

11: procedure PUTTHATTHERE(x1, y1, x2, y2)
12:   if (state = 2 and N > 3) or state = 3 then
13:     state = 3
14:   else
15:     return
16:
17:     # code to move object from (x1,y1) to (x2,y2) #
18:
19: end procedure

20: procedure REMOVEOBJECTS( )
21:   if state = 3 then
22:     state = 1
23:   else
24:     N = 0
25:     return
26:
27:     # code to remove all objects #
28:
29: end procedure
30:
31: # code for the supervisory mechanism that invokes the #
32: # three aforementioned functions in a timely manner #
33:
```

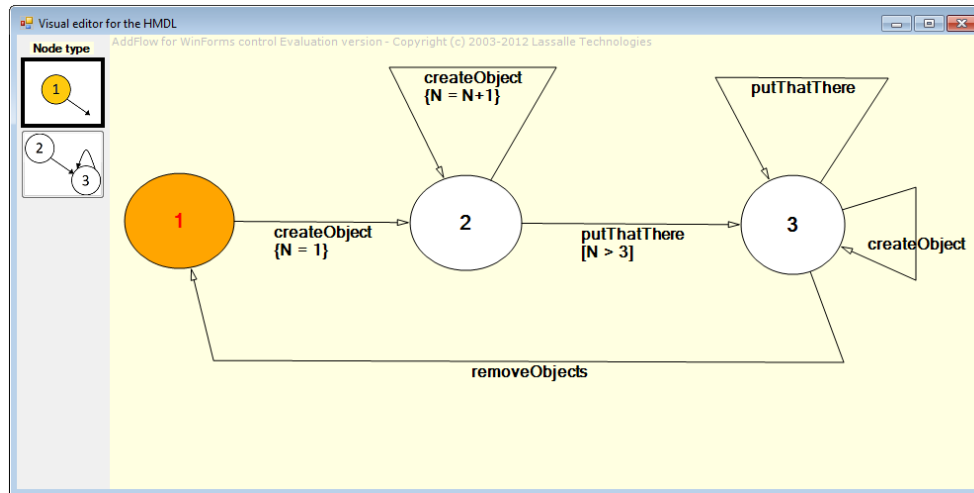


Figure 7.1: HMD2L model representing the dialog supported by the Put-That-There system described in Section 7.3.

are more than 3 objects on the form –notice the label $[N > 3]$. Finally, the removal of objects sets the system to its initial state: the node labelled as 1.

Comparing the code/model for the dialog-based Put-That-There

The comparison criteria are shown in Table 7.3.

1. When using event callback code, programmers are required to manually identify the context-of-use of the system, a task that is performed by maintaining state variables, such as *state* in Algorithm 5, line 1. Hasselt programmers, in contrast, can delegate the identification of the ever changing context-of-use to Hasselt UIMS. This is possible because in a HMD2L model, the contexts-of-use are explicitly represented (as nodes) instead of being encoded in variables. As shown in in Algorithm 5, the maintenance of state variables involves conditional convoluted logic (nested if-else statements). Although, in the system under analysis, the presence of these if-else statements seems harmless, this may not be the case for more complex systems. The conditional logic required to identify the context-of-use of an interactive system is insidious because it appears to work fine initially, but does not scale up as the system grows in complexity [Samek 09]. The HMD2L model also has conditional clauses (e.g. $[N > 3]$), but since it does not need state variables, these conditional

Dimension	Hasselt	Event-callback code
1. Number of state variables	None	One
2. Number of extended state variables	One	One
3. Viscosity	High	Low
4. Space occupied by code (Diffuseness)	Model is compact	Code is very spread out
5. Abstract gradient	One level of abstraction	Multiple level of abstractions
6. Hidden dependencies	Yes	No

Table 7.3: Dimensions used for comparing Hasselt with equivalent event-callback code based on the implementation of the dialog-based Put-That-There system.

clauses are always simpler than the clauses written with event-callback code.

2. The context-of-use of a system represents a qualitative aspect of the system. However, in a given context-of-use, many changes can be experienced by the system, without this leading to a new context-of-use [Samek 09]. The node 2 of the Put-That-There system, for instance, represents the context-of-use in which there are objects on the canvas, none of which has yet been moved. The number of objects does not characterize this context-of-use; it only represents one quantitative aspect of the context-of-use. The number of not-yet manipulated objects on the canvas is called *extended state variable* [Samek 09]. Both Hasselt and event callback code require the variable N to keep track of one quantitative feature (i.e. number of objects) of a certain context-of-use (i.e. state 2).
3. Small variations in the proposed dialog model may require more changes with HMD2L. If the studied Put-That-There had to generate a beep upon the creation of the first object, for instance, the changes with each language would be as follows. With event callback code, one would just have to add one statement for generating the beep, which can be done right next to the statement $N = 1$ (Algorithm 5, line 4). With

Hasselt, in contrast, one would have to create a new composite event *createFirstObject* (in the CEDL editor) whose system responses would be the invocation of `CREATEOBJECT` and the generation of a beep (to be declared in the SRDL editor). Later, with the HMD2L editor, one has to include the event *createFirstObject* between node 1 and node 2. The separation of concerns, which confers a clean organization to Hasselt projects, has a negative impact on the viscosity of Hasselt, when visual models are involved.

4. With event callback code, the conditional logic (if-else statements) implementing dialog management is split across many methods, which may be so spread out that one may need scrolling several screens in order to read and update this code. In contrast, HMD2L models include fewer and simpler, conditional clauses, all of them centralized in one at-a-glance diagram.
5. There can only be one HMD2L model per Hasselt project, which means that the dialog model has to be specified in one single level of abstraction. This may be a serious drawback for medium-large scale systems, which may require some degree of modularization. The tools supporting mainstream event languages, on the other hand, allow programmers to work at multiple levels of abstraction. The functions `CREATEOBJECT`, `PUTTHAT THERE`, and, `REMOVEOBJECTS`, shown in Algorithm 5, can be subjected to other higher-level constraints that can be verified outside these functions, at a higher level of abstraction.
6. The event callback code is self-contained; it completely describes how the system manages the dialog with the end user. But HMD2L models, in contrast, refer to composite events whose definitions are not visible in the same editor. The full understanding of a dialog model requires switching between two editors, but always one of these, the text editor for CEDL/SRDL or the visual editor for HMD2L, will be hidden.

7.4 Wrapping up the results

7.4.1 About interaction models

In comparison with event callback code, Hasselt programs are more concise (i.e. they occupied less lines of code) and simpler (i.e. they do not have maintain the interaction state). But on the other hand, Hasselt has limitations to

fine tune existing code. This is partly due to its declarative nature as well as to poor design decisions (e.g. the time interval under which two inputs are considered simultaneous is the same for every pair of inputs regardless of their recognition speeds). Since Hasselt is expected to be used in the rapid prototyping phase, given the current version of Hasselt UIMS, fine tuning would have to be postponed to the implementation of the final system. Another non-negligible problem of Hasselt is that it has many hidden dependencies: due to the design of the Hasselt editors, the automata required to fully understand SRDL code are displayed one at a time.

7.4.2 About dialog models

The dialog models elaborated with Hasselt include fewer, simpler, and better centralized conditional clauses than the ones described with event-callback code. This is because the conditional clauses of Hasselt models do not include state variables. But, on the other hand, Hasselt dialog models have to be described in one level of abstraction only, which eliminates the well-known advantages of modularization: reusability and enhanced readability, both being well-appreciated, especially in medium-large systems. Besides, some small changes in the dialog system may require Hasselt programmers to navigate across the three Hasselt editors, which may be tiresome, error-prone, and time-consuming.

7.5 Threats to validity

It is important to mention the factors that may jeopardize the validity of this comparative study.

- It is difficult to know the extent to which the presented sample is a good representation of the universe of multimodal interactions that one may want to prototype. The chosen sample was simple enough so that it can be subjected to detailed analysis, but despite its relative simplicity, it exhibits the fundamental problems of the event callback model, problems that the proposed composite event-based approach tries to simplify.

On the one hand, when modeling multimodal and multitouch interactions, the event callback model leads to a ‘callback soup’, in which the code for tracking the state of the interaction is split across several event handlers [Kin 12b]. This problem is observed in the code of the two studied interaction models: *put-that-there* and *pinch*. On the other hand,

when modeling human-machine dialogs, the event callback model often leads to deeply nested if-else constructs, which are difficult to get right even by experienced programmers [Samek 09]. These nested if-else statements, which are required to constrain the system responses to specific contexts-of-use, appeared in the dialog system studied above.

- Some may claim that the event callback code shown in this chapter reflects our own programming style, but it is not necessarily an ‘average’ programming code, i.e. other programmers can implement it in a different way. To reduce the effects of our personal style, the event callback code was described at a very high-level of abstraction, a level in which the particular ways of tweaking some very specific aspect of a problem are blurred, and only the overall structure of the code remains visible to analysis. Furthermore, to mitigate the effects of personal bias (i.e. unconsciously coding to favor our approach), we used, at least in one of three studied cases, code proposed by another group of researchers: as mentioned above, Algorithm 4 is almost a perfect copy of the pseudocode shown in [Kin 12b].

7.6 Summary

This chapter compared the code required to create interactive prototypes with both Hasselt and event-callback code. The comparisons were grouped into two categories.

In the first category, two interaction models were compared. Concretely, we inspected the source codes of one multimodal interaction and one multitouch interaction, each implemented with both Hasselt and event-callback code. The source codes were compared based on eleven dimensions, some of which, were taken from a well-known theoretical framework, called Cognitive Dimensions [Green 89]. Among other results, it was found that Hasselt makes it possible for programmers to get rid of the code for maintaining the interaction state, which, was qualified –with sound reasons– as difficult-to-read and difficult-to-maintain [Kin 12b].

In the second category of comparisons, one dialog model was compared based on several criteria. The results showed that Hasselt simplifies the amount and complexity of the conditional statements required to implement dialog management. But on the other hand, some small changes in an existing dialog model may be more time-consuming with Hasselt than with event-callback code. This may be partly to the graphical nature of the visual editor

as well as to the possibility of having to switch between the visual and textual editors.

The following chapter will investigate whether the simpler, shorter programs brought about by CEDL/SRDL as well as the fewer, simpler, and better centralized conditional clauses brought about by HMD2L are reflected in higher programming efficiency.

Chapter 8

User Study

To the best of our knowledge, none of the UIMSs mentioned in the Section 2.2.1 has been evaluated in user studies. Nonetheless, outside the multimodal domain, we found two user studies that guided us in the design of our experiments.

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 and InterState. It was verified that InterState models are faster to modify than equivalent event-callback code written in RaphaelJS [Oney 14].

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 12a].

Since software projects often require programmers not only to comprehend but to write programming code, we followed the schema of Oney et al [Oney 14]. We asked participants to modify an existing prototype with both Hasselt and equivalent event-callback code.

8.1 Hypotheses

Based on the results of a pilot test and the code inspection presented in the previous chapter, this study will test two sets of hypotheses.

First, we hypothesized that the adaptation of a multimodal interaction model requires (1) less time, (2) less code testing, and (3) less mental effort with Hasselt (CEDL and SRDL) than with C#.

Second, we hypothesized that the adaptation of a dialog model requires (1) less time, (2) less code testing, and (3) less mental effort with Hasselt (HMD2L) than with C#.

Each set of hypotheses will be 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/model until he informs the researcher about the completion of the task. The amount of testing involved during the experiment will be measured as the number of times the participant enters into runtime mode. The mental effort required by a programming task will be obtained with a subjective post-task questionnaire in which participants must indicate the perceived difficulty of the programming task.

8.2 Method

8.2.1 Study Design

The presented study had two parts. In the first part, we asked participants to modify a multimodal interaction model that was described with both CEDL/SRDL and C#. In the second part, we asked them to perform equivalent modifications to a dialog model described with both HMD2L and C# (Figure 8.1a).

Participants took part in the study one by one; each of them participated in both experiments. Each experiment is a within-subject experiment that started right after a short training session. In the experiment, the participant was shown 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 changes to the prototype (the changes required for each experiment are described in Section 8.2.3 and Section 8.2.4). Each participant had to sequentially perform the changes in both Hasselt and C# (Figure 8.1b). We shuffled the order of the language to

be used first so that the aggregated experience bias can be neutralized. The changes to be performed were explained orally, but also written on a sheet that the participant could check during the experiment.

While the participant modified the code, the researcher was observing using a second monitor showing the same information as the participant's screen. This way, the researcher could watch how the participant navigated through the code/visual model, count how many times the partial changes were tested in the runtime environment, and measure the completion time of the programming task.

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 programming task. In total, each participant filled four of these questionnaires –one for Hasselt and one for C# in each experiment.

At the end of the user study, i.e. after both experiments, the participant was asked to fill a usability questionnaire and was interviewed by the researcher. The study design is sketched in Figure 8.1.

8.2.2 Participants

We recruited 12 participants, all of which are male. The overall programming experience of the participants ranges from 4 to 13 years; and their C# experience, between 1 and 8 years (Figure 8.2). The pool of participants was quite varied. It includes master and PhD students, post-docs, and industry programmers, from different universities and countries, with different backgrounds (computer science and engineering), with and without knowledge in finite state automata (FSA).

8.2.3 First Experiment. CEDL/SRDL versus C#

Before starting the first experiment, each participant was given a tutorial about CEDL and SRDL (see Appendix D.1). With this tutorial, participants were able to describe a simple multimodal interaction by following step-by-step instructions. The completion of the tutorial took approximately 10-15 minutes, during which participants could ask questions to the researcher in case of doubts. In this way, the participant 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 prototype to be modified allowed end users to create and move virtual objects from a canvas. New objects could be created, in random positions,

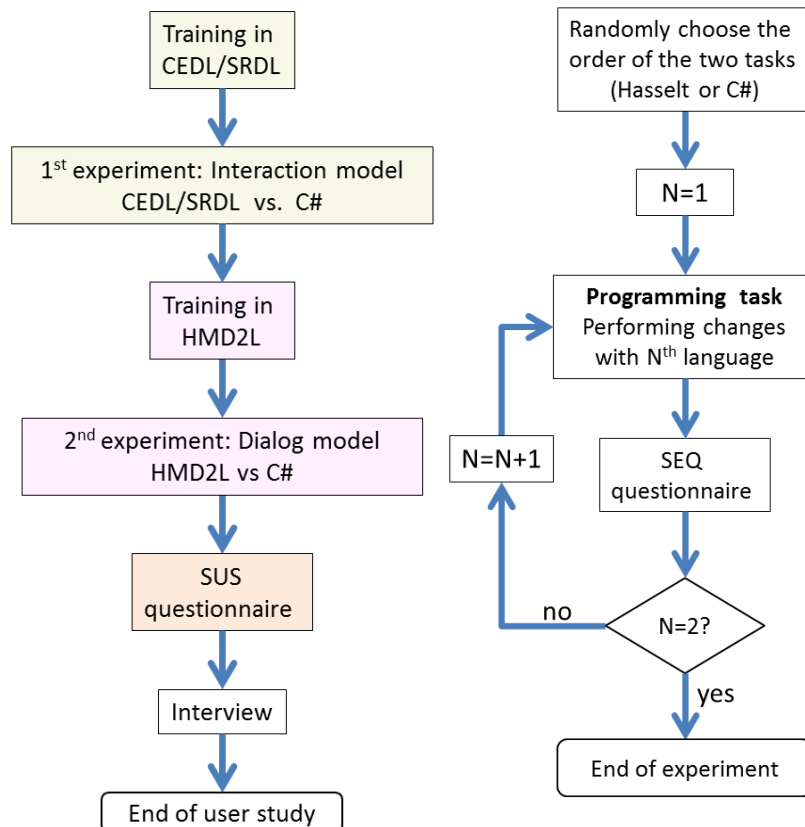


Figure 8.1: (a) Sequence of steps followed by each participant in the user study. The first and second experiments referred to in this sequence are carried out in an analogous fashion. (b) Common flow of activities followed during the first and second experiment. Here Hasselt would mean CEDL and SRDL for the first experiment; and HMD2L for the second experiment.

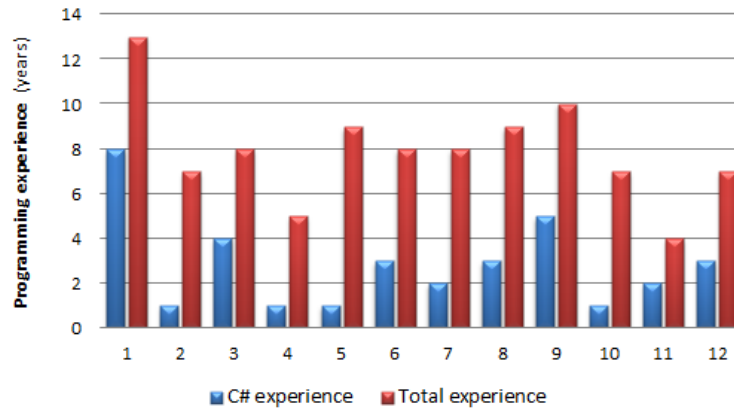


Figure 8.2: Programming experience of the 12 participants.

through the voice command *‘create object’*. Existing objects could be moved by issuing *‘put that there’* while clicking on both the target object and its new position. The front-end of the prototype is shown in Figure 4.3a.

The participant was asked to adapt 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. The changes only required modifying the interaction code, not the application-specific code (Appendix D.2).

The aforementioned prototype was described with both C# and Hasselt. Each participant had to modify both sources within a time limit of 30 minutes per language. The order of the languages to be used was balanced over the participants so that the aggregated experience bias can be neutralized.

Finally, we report that the C# code was partitioned into regions, i.e. collapsible blocks of code. Right before starting to perform changes with C#, participants were shown and explained the purpose of these regions of code. They were explicitly informed that there was no need to modify the configuration code or the back-end code, that just maintaining the global variables across the event handlers would suffice to change the interaction. Indeed, we ensured that all regions containing code for configuration of input sources and back-end functions were collapsed during the task. We provided this information in an attempt to make the comparison as fair as possible. With Hasselt, the code for configuring input recognizers as well as the back-end code is not visible either. The former is enclosed into Hasselt UIMS, the latter into a canned EXE file.

8.2.4 Second Experiment. HMD2L versus C#

Before starting the second experiment, each participant was given a 10-minutes tutorial about HMD2L (Appendix D.1). Participants had to describe a simple human-machine dialog by following step-by-step instructions. In this way, the participant got acquainted with the visual editor, debugging tools, and runtime environment of Hasselt UIMS. There was no training in C# since all participants had experience with this language.

For the experiment, the participant was presented with a system similar to the one shown in Figure 4.3a. It allows users to create and remove virtual objects from a canvas in response to multimodal input. In the version given to participants, the objects can be created or removed at any time, after which the end user is acknowledged with voice feedback.

Participants were asked to change the system so that it can handle two contexts-of-use: the command to remove objects must only be processed if there are objects on the screen; otherwise, it should be ignored (Appendix D.2). The required changes can be made by modifying the human-machine dialog model only; and participants were orally informed about this fact.

As in the previous experiment, the prototype was described with both C# and Hasselt. Each participant had 30 minutes to modify each source code. Once again, the order of the language to be used was balanced over the participants.

8.3 Measures

8.3.1 Observations

While the participant performs the required modifications with a certain language, the researcher is observing how the participant navigates through the source code. This is to try to identify whether there is a pattern, an order in which different parts of the code are changed, whether some parts of the code require more mental effort than others, and whether some programming errors are repeated by many participants. The researcher also monitors the participant's working time and counts the number of times the code is tested.

8.3.2 Single Ease Question (SEQ) questionnaire

Right after completing the changes with each language, each participant was asked to complete the Single Ease Question (SEQ) questionnaire (Figure 8.3), a rating scale ranging from 1 (anchored with "Very difficult") to 7 (anchored

Overall, this task was?

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

Figure 8.3: Single ease question (SEQ) questionnaire. Each participant filled the SEQ four times –namely after using C# in experiment 1, after using Hasselt in experiment 1, after using C# in experiment 2, and after using Hasselt in experiment 2.

with “Very easy”). It aimed to assess the perceived difficulty (or perceived ease, depending on one’s perspective) of a task. The SEQ has been proven to be reliable, sensitive, and valid while also being easy to respond [Sauro 09].

8.3.3 System Usability Scale (SUS) questionnaire

After completing both experiments, participants had to fill the System Usability Scale (SUS) questionnaire [Brooke 96] (Figure 8.4), which has become a well-known questionnaire for end-of-test subjective assessments of usability [Lewis 09].

The SUS questionnaire consists of 10 items with 5-point scales numbered from 1 (anchored with “Strongly disagree”) to 5 (anchored with “Strongly agree”). SUS test scores are normalized to values between 0 and 100.

There is baseline information showing that the average and third quartile of 324 usability evaluations performed with SUS are 62.1 and 75.0 respectively [Lewis 09]. This information allows us to put our results in the perspective of other systems that were evaluated with SUS.

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 09]. In the taxonomy proposed by Grossman et al. [Grossman 09], this learnability falls within the category of initial learnability given that participants have been exposed to Hasselt for the first time during this experiment.

It must be mentioned that, when designing the user study, a pool of post-task

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

Figure 8.4: System Usability Scale (SUS) questionnaire and average scores per question obtained for Hasselt UIMS. Raw data can be seen in Appendix D.4.

questionnaires and usability questionnaires were evaluated (Appendix D.5). Considering that modifying a program is already a time-consuming task that demands lots of mental effort from the participants, we decided to use short and simple questionnaires, such as the SEQ and SUS, in order to avoid respondent fatigue, a well-documented phenomenon that occurs when participants become tired of the survey task and the quality of the data they provide begins to deteriorate [Lavrakas 08].

8.4 Results

8.4.1 Modifying an interaction model: CEDL/SRDL vs. C#

All 12 participants completed the experiment when using Hasselt; but only 10 succeeded with C# –the others exceeded their allotted time.

Completion time

The following results are based on those 10 participants who completed the required changes with both languages (Figure 8.5, a).

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, $W = 0$, $Z = -2.8085$).

Code testing effort

The following results are based on those 10 participants who completed the required changes with both languages (Figure 8.5, b).

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 less frequently (p-value = 0.009766, $W = 2.5$, $Z = -2.4233$).

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 (Figure 8.5, c).

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 with 95% of confidence. 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, $W = 78$, $Z = 3.0953$).

Note: The use of Wilcoxon signed-rank tests instead of paired t-tests responded to the fact that we could not guarantee the normality assumption required by the latter. The non-normality of the pair differences was observed in the Q-Q normality plots shown in Figure 8.6.

8.4.2 Modifying a dialog model: HMD2L vs. C#

All 12 participants could complete the changes with both languages Hasselt and C#. The data from observations and post-task questionnaires are synthesized in Figure 8.7. After inspecting the data, we decided to drop the only participant who had no previous experience with FSMs. He was an outlier in

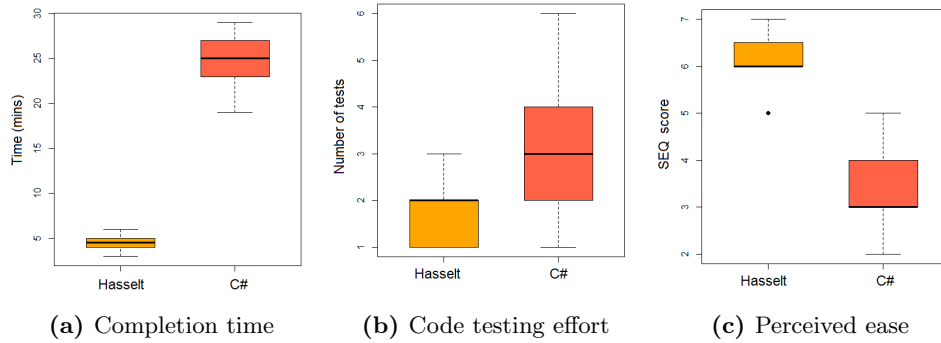


Figure 8.5: First experiment (CEDL/SRDL versus C#). Boxplots (a) and (b) summarize the measurements for the 10 participants who succeeded with both languages. Boxplot (c) includes all 12 participants. Raw data can be seen in Appendix D.4.

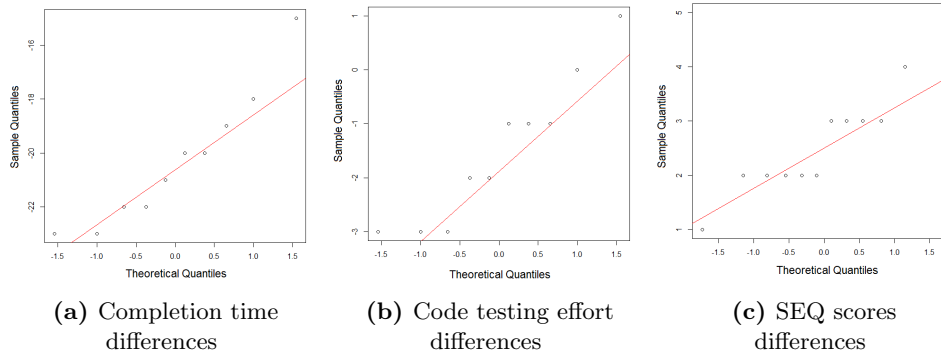


Figure 8.6: First experiment. Q-Q normality plots for the differences in (a) completion times, (b) code testing effort, and (c) SEQ scores. Plots (a) and (b) involve data of the 10 participants who succeeded with both languages; plot (c) includes all 12 participants. Those many points falling far from the straight line, depicted in red, indicate that the normality of the pair differences cannot be guaranteed, i.e. it may not be safe to apply paired t-tests.

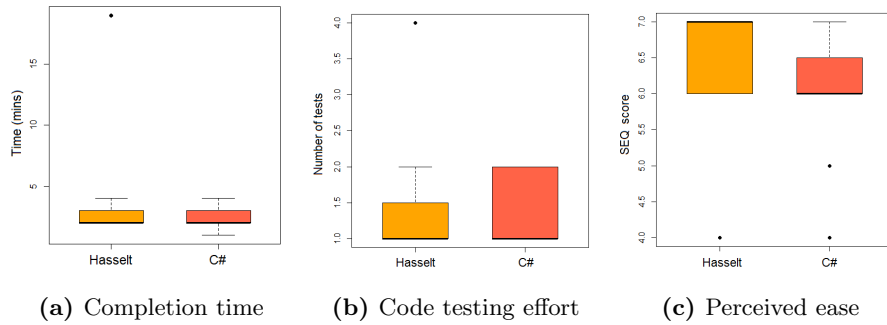


Figure 8.7: Second experiment (HMD2L versus C#). Boxplots showing the data collected from the 12 participants. Raw data can be seen in Appendix D.4.

the three plots shown in Figure 8.7. Therefore, the following results are based on the remaining 11 participants.

Completion time

On average, changes made with Hasselt took 2.4 minutes in comparison with the 2.1 minutes when using C#. It could not be proven that Hasselt completion times are lower than C# completion times: a Wilcoxon signed-rank test resulted in $p\text{-value} = 0.9688 > 0.05$ ($W = 12.5$, $Z = 1.3828$).

Code testing effort

On average, programmers tested their code 1.2 times when using Hasselt and 1.4 times when using C#. But this result was not statistically significant. We could not reject the null hypothesis in favor of the alternative hypothesis that the code testing effort is lower with Hasselt than with C#: a Wilcoxon signed-rank test resulted in $p\text{-value} = 0.25$ ($W=0$, $Z = -1.4142$).

Perceived easiness of the task

The average SEQ scores for Hasselt and C# were 6.6 and 5.9 respectively. In this case, we found that this difference in favor of Hasselt was statistically significant. A Wilcoxon signed-rank test indicated that the alternative hypothesis that the SEQ scores are higher for Hasselt than for C# can be accepted ($p\text{-value} = 0.0078$, $W=28$, $Z = 2.6153$).

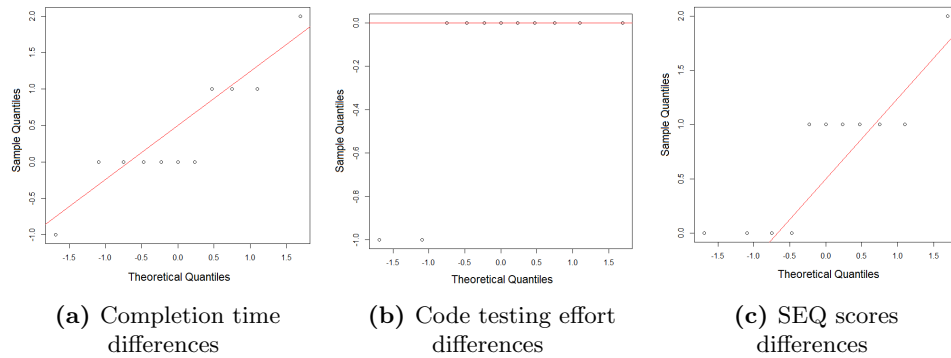


Figure 8.8: Second experiment. Normal Q-Q plots for the differences in (a) completion times, (b) code testing effort, and (c) SEQ scores, calculated for the 11 participants whose data was analyzed. Those many points falling far from the straight line, depicted in red, indicate that the normality of the pair differences cannot be guaranteed, i.e. it may not be safe to apply paired t-tests.

Note: Like in the previous experiment, the use of Wilcoxon signed-rank tests instead of paired t-tests responded to the fact that we could not guarantee the normality assumption required by the latter. The non-normality of the pair differences was observed in the normal Q-Q plots shown in Figure 8.8.

8.5 Observations

What the researcher observed in the first experiment, while the participants were modifying *C#* code, coincides with what was revealed in the interviews. Lots of time and effort are spent trying to maintain the interaction state, i.e. a set of global variables, across multiple event handlers. The participants constantly navigate from one portion of the code to another, extremely focused, trying to figure out what variables have to be maintained and where this is the case. Even right before entering into runtime mode, they are still mumbling to themselves, trying to check they did not forget any variable. In contrast, with Hasselt UIMS, participants never had to perform this time-consuming task since Hasselt UIMS internally maintains the interaction state while tracking the composite events.

In the same experiment, there was an incident that is worth mentioning. Although he was never asked for an explanation, Participant 2, one of those

who failed finishing the changes with C#, tried to justify himself by claiming that the prototype he was given never worked correctly despite of the fact he tested it himself multiple times before the experiment. We did not have any similar incident with other participants.

In the second experiment, we observed that although the participants tested Hasselt models fewer times than equivalent C# code, task completion times were lower with C# than with Hasselt. This may be due to the fact that the modification of Hasselt visual models was more time-consuming than the editing of C# textual code. There were cases where programmers had to try more than once to put a link between a pair of nodes. This is because the visual editor requires to precisely click on the center of the source node, which may be hard when one does not have much experience with the Hasselt visual editor. Although the visual editor can be upgraded in order to alleviate this issue, there is another cause of delays that cannot be solved by the UIMS developer. Many participants invest a lot of time rearranging the layout of the visual model with changes that do not have any intention of altering the semantics of the model (e.g. changing the positions of the nodes or moving the elbows of an arc). This may be partly a consequence of the concern for the aesthetics of some individuals, but we think there are also objective reasons that explain this behavior. The code editors conveniently restrict the set of choices that programmers have to make: the changes in the code editor are discrete –you can write a statement in one of a discrete number of lines–, whereas the changes in the visual editor are continuous –you can drag the nodes through every (x,y) point of the editor. Besides, the code, as any text, is linear: once you write a sentence, the next sentence must go in the following line. In contrast, the visual model, is non-linear, meaning that there is no intrinsic sense of order in the diagram: the position of the first node does not give any clue about what should be a good position for the second node. Furthermore, whereas the code editor helps programmers by performing automatic text indenting, the visual editor does not offer this type of support and let the whole design in charge of programmers.

One final observation is about the outlier of the second experiment, who was the only participant lacking knowledge on finite state automata (FSA). When using CEDL/SRDL, he managed to finish the experiment in a time that is comparable to the average completion time; however, when using HMD2L, he took much longer than average. His lack of theoretical knowledge did not prevent him from correctly interpreting the FSA auto-generated from CEDL/SRDL. However, as he admitted in the interview, that deficiency may have had a negative impact in his performance to depict HMD2L models.

8.6 Usability and learnability 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 09]. We are aware however that this result can only be used as a guideline: the SUS scores reported by Lewis et al. are over very diverse systems, and not only within the subset of over programming environments.

The average scores obtained for each of the 10 items of the SUS questionnaires were 3.5, 2.1, 4.3, 1.9, 3.1, 2.3, 4.4, 1.9, 3.9, and 1.4 (Figure 8.4).

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 two questions that define learnability [Lewis 09], may be indicating that Hasselt is perceived as easy-to-learn. This matches with the fact that all participants completed the two experiments with Hasselt even though they received little training. Larger scale experiments should confirm if this learnability is independent from or dependent on previous programming experience.

8.7 Interview highlights

With regard to the first experiment, there was unanimous consent that the required changes were completed more easily when using Hasselt than when using C#.

When participants were asked “Why do you think it is more difficult with C#?”, many 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) –other participants had at least pen-and-paper experience. This lack of knowledge did not affect his performance for the first experiment; when using Hasselt, he completed the experiment in 4 minutes. However, when using HMD2L in the second experiment, he took 19 minutes for completing the task. This is a big difference with regard to the 2.4 minutes required, on average, by the other 11 participants. This may be an indication that reading the FSA generated from CEDL code is so intuitive that programmers may not need theoretical background. But for depicting FSMs required by HMD2L, a training session may not be enough and theory about FSMs may be highly recommended.

Continuing with the second experiment, participants perceived that performing the modifications with Hasselt was simpler than with C#. Some participants mentioned that HMD2L is simpler because its visual models provide an overview of the whole dialog in one single screen. In contradiction with this subjective appreciation, C# led to better completion times than HMD2L.

8.8 Threats to validity

8.8.1 Construct validity

Construct validity is defined as the degree to which a test measures what it claims, or purports, to be measuring [Brown 95]. The construct validity of our empirical study could have been affected as follows.

- We measured the code testing effort as the number of times the participant enters in the runtime environment. This means we assumed that participants have to run the program in order to evaluate whether the source code is correctly specified. This definition may not be complete since it leaves out the effort made by the participant while reading the

program and ‘running and testing the code inside his head’.

- We cannot completely guarantee that the observed differences were due solely to Hasselt rather than to the specific application used as part of the programming task. This threat could have been mitigated by performing not only one, but many experiments with different applications. However, this was unfeasible in our case. Getting experienced programmers who can volunteer to participate in a relatively stressful experiment that lasts more than one hour was already a very difficult task. Asking them to stay for two or three consecutive experiments or to come in different sessions was simply unfeasible.
- The SUS questionnaire may have 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 and only 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.

Construct validity is not the only type of validity that must be considered when designing empirical research. An empirical study is said to have internal validity when the impact of almost all influencing factors are excluded, so the study is performed in a highly controlled setting. In contrast, external validity refers to avoiding such a strict control so that the experiment can emulate a real-world situation instead of an ideal, unrealistic one. Whereas external validity increases the chances that results can be generalized to more realistic, every-day situations, internal validity allows researchers to pinpoint the reasons of improvement or degradation, but at the cost of generalizability [Siegmund 15].

8.8.2 Internal validity

We pursued for internal validity in the following way.

- The order of the languages, so which language was to be used first, Hasselt or *C#*, was balanced over the participants so that the aggregated experience bias can be neutralized.

- Since the goal was to measure the effort for implementing multimodal interaction and not application code, we had to restrict participants to modifying interaction code only. This was easy to achieve with Hasselt since it only allows describing interactions. The code for configuring input recognizers as well as the application code is not visible in Hasselt. The former is enclosed into Hasselt UIMS, the latter into a canned EXE file. In contrast, the C# program included both application code and interaction code in the same file. To avoid that the presence of application code distracts participants and influences their performances, we decided to hide it during the experiment. By using regions (i.e. collapsible blocks of code), the application code (as well as the configuration code) was collapsed during the experiment. Only the event handler and the variables encoding the interaction state were shown.
- We offer participants a tutorial on Hasselt but no tutorial on creating multimodal applications using C#. To reduce the damage caused by this omission on the internal validity of the experiment, participants were briefly explained the purpose of each of the aforementioned code regions right before starting the changes with C#. They were explicitly said that the goal can be achieved by correctly maintaining and interrogating the global variables in the event handlers. The variables and event handlers are within a different code region.

8.8.3 External validity

In order to confer our results with high external validity, we allowed some ‘randomization’ to the experiments.

- From a methodological perspective, there was no structured interaction between the researcher and participants during the experiments. This contrasts with other approaches commonly used in empirical studies, such as the *think-aloud* protocol and the *question-suggestion* protocol [Grossman 09]. The former would require participants to speak out while programming in order to provide the researcher with insights about their programming logic. The latter would allow the researcher to give advice proactively to the participant. In our experiments, the researcher only interferes when participants ask for questions. In our opinion, this is a more realistic scenario that reflects the typical case of a programmer working by his own and eventually asking for advice to more expert programmers when he got stuck in a problem.

- The pool of participants was quite varied. As mentioned before, there were master and PhD students, post-docs, and industry programmers, from different universities and countries, with backgrounds in computer science and engineering, with and without knowledge in finite state automata (FSA). Even if the group of participants is relatively small, we are convinced that the validity of the experiments is positively influenced by the variation in the pool of participants.
- Despite the fact that event-driven languages require similar work practices to describe interactive systems (i.e. to implement a series of event handlers that are to be bound with user events), it may be risky to assume without further experiments that the advantages obtained by Hasselt can be repeated for other event-driven languages.

8.9 Lessons for the future

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

- Researchers must be careful when choosing a method for determining the duration of the programming tasks. Our decision to give participants a time limit of 30 minutes per programming task was based on a pilot test. The 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 first experiment with C#, the hardest of the four programming tasks. This made us believe that a time interval of 30 minutes (almost double) would be enough for all participants and for all programming tasks. Unfortunately, this was not an optimal decision. The 30-minute period was not enough for two participants who failed to complete the first experiment with C#. As a consequence, we lost two data points for the first experiment: there is no completion time for those who could not complete the task. Future researchers may want to consider carrying out pilot tests with several randomly chosen participants in order to obtain a more realistic idea of how long the programming tasks could take. An alternative option consists of using more complex, formal mathematical models for estimating the maximum acceptable task completion time [Sauro 05].
- 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, while designing the study, 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.

- 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 baseline information with which to compare our results.
- For the cases where the training session is designed to be short and carried out before the test, it is important to gather participants with a similar background. Otherwise, some participants can benefit more from the training than others, which may cause the appearance of outliers.

8.10 Summary

This chapter has presented the results of two experiments designed for comparing the programming efficiency achieved with Hasselt and with event-callback code.

Due to the lack of similar user studies in the domain of multimodal systems, we used two relatively recent studies, which were carried out to compare InterState [Oney 14] and Proton++ [Kin 12a], each against event-callback code. InterState was tested for mouse-and-keyboard interactions; Proton++, for touch interactions. To the best of our knowledge, Hasselt is the first language to be compared against event-callback code for the case of multimodal interactions.

In the first experiment, participants were asked to perform equivalent modifications of a multimodal interaction model with both CEDL/SRDL and C#. Objective measurements showed that the textual notations of Hasselt (i.e. CEDL/SRDL) led to higher completion rates, lower completion times, and less code testing. The SEQ questionnaire also revealed that CEDL/SRDL are perceived as easier-to-use than equivalent event-callback code.

In the second experiment, participants were asked to perform equivalent modifications of a dialog model with both HMD2L and C#. In this case, most results were not statistically significant. Only the subjective SEQ questionnaires showed better results for HMD2L, i.e. participants perceived that the changes were easier to perform with HMD2L than with event-callback code.

Finally, we close Part II of this thesis by highlighting the correlation between the results obtained in the previous and the current chapter. On the one hand, the many advantages of CEDL/SRDL over event-callback code, identified through code inspection in the previous chapter, are materialized in higher programming efficiency, as attested in the user study presented herein. On the other hand, although the code inspection unveiled slight advantages of HMD2L over event-callback code, though still not strong enough to improve programming performance.

Part III

Discussion, Conclusions, and Future Work

Chapter 9

Discussion

This chapter starts by describing how we accomplished the research goals presented in the introductory part of this dissertation. These two goals refer to the design and evaluation of a composite event-based language. The chapter then discusses the contributions and limitations of the present research.

9.1 Design of a composite-event based language

The first goal of this research (Section 1.2) was to design a composite event-based language and its supporting tool. Both artifacts have already been extensively discussed in Part I. The design decisions behind the nature, paradigm, and notations of Hasselt are described below.

9.1.1 Why textual? Why event-driven?

When surveying several multimodal interaction descriptions languages, it was noticed that many of them were visual languages and/or required using concepts such as CARE properties, transition rules, or logic-based concepts that were unknown by the author of this thesis, despite of his long experience as a developer of WIMP interactive systems with event languages. As part of the investigation, the author downloaded three tools (Icon, Squidy, and Pet-Shop) and utilized their underlying languages to describe basic interactions, but every time he got stagnant and frustrated trying to deal with some corner case, the same question always popped up in his mind: “Why would I use this language if I can already do whatever I want with event languages?”. This

repetitive question enlightened us to make the first decision about Hasselt, in a time when we did not have any idea about how Hasselt should look like: In order to create a programming language that intends to be adopted and well-regarded by its users, the language should not deviate too much from the mental models and work practices that programmers are accustomed to follow in their day to day work. It is not enough that a language can specify the behavior of the intended system; it is equally important that its intended users want to use it.

We later verified that the nuisance generated by having to deviate from the “native language” was not just a personal reaction. As will be exposed below, similar reactions were observed in different scenarios when programmers were asked to work with a new language.

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 87]. He illustrates his point by confessing that the difficulty for describing interfaces in terms of grammars caused the SYNGRAPH system [Olsen Jr 83] 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 86], 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 89].

Influence of previous programming experience

The previous cases highlighted the resistance of programmers 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 this study, a group of master and doctoral students had to elaborate a project consisting of describing a system with the language Live Sequence Charts (LSC), whose syntax as well as underlying concepts were unknown by participants, who, instead, had experience with other programming languages, mainly C++ and Java [Alexandron 12]. The results showed that previous programming experience leads programmers 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 01]), 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 UIMSs aimed at describing multimodal interactions use visual languages, it is also important to remind the experiment carried out by Oney et al. [Oney 14], in which 20 developers performed equivalent modifications with both InterState, a visual language, and RaphaelJS¹, a textual, event language.

The researchers reported that, during the interviews, the participants (experienced developers) showed skepticism about using visual languages in practice since they still feel more comfortable with standard imperative code. The authors hypothesized, and we agree with them, 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 IEEE² and by TIOBE³ 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, programmers have become accustomed to, and naturally, they will not want to lose.

Finally, it would be unfair to close this section without addressing the question: “Why would I use Hasselt if I can already do whatever I want with event languages?”. The answer is that, in our vision, Hasselt pretends to be

¹<http://raphaeljs.com/>

²<http://spectrum.ieee.org/static/interactive-the-top-programming-languages>

³<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

an event language augmented with notations for declaring composite events (Section 1.3). This thesis has just explored how these new notations could be and what benefits they could bring about. This is the first step towards our envisioned language.

9.1.2 Why these notations?

Although it may not be obvious due to its external appearance, the idea of composite events was conceived when analyzing the visual notations of SMUIML [Dumas 14]. The nested boxes used to group simple events into multimodal events were a straightforward indication that composing events was the key to achieve concise representations of multimodal interactions without departing too far from event languages. SMUIML models were the first models where we could read *if this sequence of events occur* \rightarrow *call this function*. Once we gained this insight, we saw that the intention of composing events was implicit in other tools too [Cuenca 14a]. There we decided to make it explicit, that is, to create a language that uses composite events as the main concept. It was at this point when CEDL was conceived.

After reviewing the literature of several research domains, we found that the idea of composing events already existed in two fields, namely Complex Event Processing (CEP) and active databases. The event operators of CEDL were chosen after surveying those domains. From the beginning, we had serious doubts about including a negation operator in CEDL. Although there is a negation operator in many languages of the aforementioned domains, the presence of this operator has also been put into question because “there is no way to represent the absence of an occurrence” [Mei 09]. We decided not to include such an operator, but, of course, we verified that the idea of a “did not occur” event (which is a better name than a “no event”) can be represented with the CEDL operators (Section 5.5.1). To know more about CEP or active databases and why Hasselt UIMS did not just exploit these existing tools, readers are invited to refer to Appendix B.

With regard to the SRDL, it must be said again that it did not exist in the first version of Hasselt. There the composite events were attached to one event handler (Figure 4.4 on page 70), over understanding that such a handler had to be fired upon the full detection of a composite event. Once the need of invoking event handlers along the whole interaction (e.g. to generate partial feedback) was noticed, SRDL was created. Initially, it was intended to enable programmers to annotate every possible node of the auto-generated FSAs, i.e. every possible interaction state. But even after this upgrade, a new big

problem surfaced when we were running multiple interactions in parallel: the ending of the *drag-and-drop* always fired the *mouse click* event. This could not be solved without using conditional clauses that restrict the firing of the mouse clicks to certain timing conditions. We investigated different types of transition networks (e.g. non-deterministic finite state automata, recursive transition network, etc.) until we noticed that the Augmented Transition Network (ATN) [Woods 70] can be the solution to our problem. In order to augment the expressiveness of our model, from a set of parallel FSAs to an ATN, the SRDL was upgraded to the version presented in this thesis. An informal proof that our models have the same expressiveness as an ATN is given in Appendix A.2; a formal proof still needs to be elaborated.

Once it was quite clear that the CEDL/SRDL led to a reduction of the code required to describe multimodal interactions, we decided to investigate whether similar gains could be replicated at a higher level of abstraction. This was the motivation to create HMD2L. To avoid HMD2L to interfere with the gains already obtained with CEDL/SRDL, HMD2L was created to have a loose relation with CEDL/SRDL. As mentioned in Section 2.2.3, there were not too many tools where the dialog models were separated from the multimodal interaction models, being SMUIML and CoGenIVE two of the few cases. It was decided to use state diagrams, the same as in the two aforementioned cases, to create Hasselt dialog models. As mentioned before, HMD2L models are not mandatory in a Hasselt project.

9.2 Evaluation of a composite event-based language

The second goal of this research (Section 1.2) was to evaluate the composite event-based language by both analytical methods (code inspections) and empirical methods (user study). Both types of evaluations were presented in Part II and the results will be integrated herein. The same as it was done with the code inspections and the user study, the loosely coupled interaction models and dialog models will be discussed separately.

9.2.1 Interaction models

Both the code inspections and the user study revealed that the advantages of using composite events instead of traditional event-callback code are clearer during the elaboration of interaction models, understanding an interaction model as a description of how the system responds to the coordinated sets of

actions that the end user performs in order to accomplish a single task (e.g. moving a virtual object, enlarging a map).

In the context of developing multimodal systems, three important tasks that have to be implemented with event-callback code are simpler or not required at all when using composite events.

First, with event-callback code, the maintenance of the interaction state requires several manual updates of a multitude of state variables. Some participants of our user study referred to the maintenance of the interaction state as one source of difficulty for modifying multimodal prototypes (Section 8.7). In contrast, Hasselt programmers do not have to declare or maintain state variables: Hasselt UIMS automatically updates the interaction state while tracking the user-defined composite events (Table 7.1, dimension 3). Second, when using event-callback code, one has to write conditional clauses in order to identify the moment when the system responses are to be conveyed. These conditional clauses can be more or less complex depending on the number of state variables that need to be interrogated. With Hasselt, in contrast, the moments when the system must respond can be referred to directly, in an explicit manner, e.g. in this state \rightarrow call this function, without need of interrogating state variables (Table 7.1, dimension 4). Third, with event callback code, the information about the user actions (e.g. skeleton joint positions or angle of a touched point) is carried by the parameters of the event handlers, which have local scope, i.e. they can only be referred to from within the event handlers. Therefore, this information may have to be saved in global variables in order to make it visible to other event handlers that will also deal with the same multimodal interaction. Hasselt, in contrast saves programmers from littering the code with global variables. In Hasselt, the event parameters have a wider scope and can be referred to at any moment of the composite event lifecycle (Table 7.1, dimension 5).

The enhanced simplicity of Hasselt in comparison with event-callback code was noticed in practice by twelve participants, who were asked to modify a multimodal prototype with both languages. They unanimously agree, in both interviews (Section 8.7) and SEQ questionnaires (whose individual scores are shown in Figure D.1a on page 230), that the required modifications were more easily performed with Hasselt than with C#. This subjective perception is in line with the objective fact that, during the same experiment, Hasselt led to higher completion rate, lower completion times, and less code testing (Section 8.4.1).

On the other hand, the creation of multimodal prototypes may be hindered by the low range of fine-tuning allowed by Hasselt (Table 7.1, dimension 7).

Some functionalities offered by Hasselt UIMS are “hermetically sealed” and cannot be tweaked, which restricts Hasselt programmers 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, calling the back-end methods asynchronously, and setting different tolerance periods for each pair of simultaneous events are operations that cannot be implemented with Hasselt, but can be coded with an event language like C#. It must be very clear, however, that this limitation is not caused by the concept of composite events that this thesis proposes. Rather, it is caused by the limited sets of functions provided by Hasselt, e.g. a keyword *call-async* could be defined to launch functions in asynchronous mode and the event $(A+B)\{500\}$ could be used to define a tolerance period of 500 ms. for a particular pair of inputs.

9.2.2 Dialog models

By means of code inspection, it was noticed that the interaction models described with HMD2L have fewer and simpler conditional clauses than those required with event callback code. This was due to the fact the conditional clauses written with event callback often include state variables, which do not exist in HMD2L models. Apparently, this gain was not enough to give HMD2L advantages in programming efficiency (Table 7.3).

The user study did not reveal a clear winner at the level of dialog models. Only one statistically significant result could be drawn from the second experiment. Based on the SEQ scores, it was found that the required changes were more easily performed with Hasselt visual models than with event-callback code. This result may be in line with the (non-statistically significant) fact that Hasselt models were tested fewer times than event-callback code: less code testing means that the right solution was found with a fewer number of attempts, which may lead participants to believe that the task was simpler. But the result is a bit uncorrelated with the (non-statistically significant) fact Hasselt led to higher completion times. The modifications with Hasselt were (perceived as) simpler, did not required much testing, but took longer! Based on our observations, we think that the slight advantage in completion time in favor of C# (2.4 mins for Hasselt versus 2.1 mins for C#) may be due to the Hasselt visual editor.

There is more decision making involved in drawing a HMD2L model than in writing code. First, each node can be placed in a continuous space of (x,y)-points, which is always larger than the discrete number of lines where a statement can be written. Furthermore, when a node is drawn, its size has

to be defined by dragging the mouse –and it is often the case that the nodes are drawn to have similar sizes. When writing code, in contrast, one does not have to set the font size for every statement to be written. Finally, there is no sense of right order when depicting a visual model: the position of one node does not restrict the position of the next nodes to be drawn; each new node can be correctly placed at any point of the screen, which invites one to think *where?* On the contrary, textual languages impose an intrinsic sense of linear order: once you write a statement in its semantically right position, it cannot be moved a single line on top of the previous statement or a single line below the next statement because that would change the program semantics.

There are also specific issues with the visual editor. To create an HMD2L model, one has to draw arcs and write annotations on the arcs. Each of these operations requires clicking on specific points of the diagram. This precision sometimes leads to errors and thus, to repeat the failed operation (Section 8.5). Besides, when a pair of nodes is connected by two arcs pointing in opposite directions, by default, the arcs will be overlapping. To avoid confusions with the arc annotations, one has to manipulate the elbows of the arcs, which is, again, an error-prone task due to the small size of the elbows.

The accumulated time dedicated to make the many small decisions mentioned above plus the time used to redress the issues of the visual editor may explain why the modifications of the visual models took longer. But, of course, all this requires to be proven with another user study.

Finally, it must be highlighted that the weak points found during the code inspection, namely hidden dependencies and abstract gradient, did not have an impact in the second experiment since the second programming task never required changes in the underlying interaction model and it was simple enough to be solved at one single level of abstraction.

9.3 Engineering problems tackled by CEDL/SRDL

Spano et al. pointed out three important problems in the engineering and development of gestural interfaces [Spano 13b]. We noticed that these problems are also present in the domain of multimodal systems. As will be exposed below, Hasselt allows overcoming these three problems and two other problems that we found ourselves.

The *event granularity problem* appears when a framework can only launch callback functions after the full detection of a gesture. This limitation prevents from providing partial feedback while the gesture is performed [Spano 13b]. Hasselt tackles this problem by allowing programmers to annotate any node

and link of the auto-generated finite state automata (FSA) with system responses, which, in practical terms means, that the system responses can be launched at any stage of the gesture lifecycle.

The *spaghetti code problem* refers to the fact that the implementation of multimodal/multitouch interactions requires splitting code across multiple event handlers [Spano 13b]. This problem is because event-driven frameworks restrict programmers to handle events one by one, without an option to defining a handling functions for a full sequence of events. In contrast, Hasselt programmers do not have to handle each event separately. Rather, they can bind one single handling function to arbitrary sequences of events that, at runtime, will be automatically tracked by Hasselt UIMS without having programmers to write a single line of code for the tracking process.

The *selection ambiguity problem*, the third problem identified by Spano et al. [Spano 13b], deserves further discussion since, in our opinion, it has not yet been completely explored by their proponents.

9.3.1 The selection ambiguity problem

Spano et al. defined the *selection ambiguity problem* as the situation in which multiple gestures may start with the same initiating sequence of events [Spano 13b]. We believe that there are actually two types of problems behind this definition.

Traditional problem

One manifestation of the selection ambiguity problem is when many gestures have a common prefix and the gestures are larger than this common prefix. This is the problem analyzed by D. Spano in his doctoral thesis [Spano 13a]. The present section will discuss this ‘traditional’ version of the problem by comparing the solutions proposed by Spano against our solution; the next section will unveil a ‘recursive’ version of the selection ambiguity problem.

The traditional version of the selection ambiguity problem can be seen when, for instance, one has to implement two-stroke gestures like the *equal* symbol (=) and the *plus* symbol (+), each of which starts with the same prefix, a single-stroke horizontal line. This entails that, at runtime, the detection of the common prefix can affect the gesture state of both the *plus* and the *equal* symbol, even though, at the end, only one of these two gestures is going to be performed. Some researchers have tried to avoid this problem at runtime by performing heavy computations at design time (e.g. by identifying and merging the definitions of all conflicting gestures [Kin 12b]). In contrast,

GestIT, the tool developed by Spano et al., does not try to avoid the problem at runtime; rather, it tracks all conflicting gestures in parallel and once it is clear which gesture is going to be performed, only that one gesture continues the recognition process. It may be subtle but there is an underlying assumption behind this approach: it is assumed that only one gesture can be performed at a time, which is not necessarily true since one may want to rotate and enlarge an interface object at the same time, for instance. That is why Hasselt enables programmers to decide on a case-by-case basis whether a common prefix is really a “problem” or not.

At runtime, similar to GestIT, Hasselt UIMS tracks all conflicting gestures in parallel. Once the common prefix is passed and it is clear which gesture is going to be performed, programmers can decide whether to eliminate the influence of the common prefix in the state of the other gestures or not. Concretely, Hasselt programmers can decide to stop or to continue tracking all or some of the other conflicting events, and to perform some actions like invoking a rollback function, if needed. The price to pay for this flexibility is that Hasselt requires more programming effort than GestIT, which only requires joining the conflicting gestures with a *choice* operator (e.g. *plus [] equal*). To have an idea of the code required to handle ambiguities with Hasselt, the specification for the case of the *plus* and *equal* symbols is shown in Appendix C.4. There the composite event *plus* is specified so that a user-defined event, *evtPlusDetected*, is raised when the vertical line of the *plus* is detected, thus making it clear that the end user is depicting the plus symbol. The user-defined event must cancel the tracking of the analogous composite event *equal*, i.e. it must immediately move the composite event *equal* to its final state without providing any type of feedback. Similarly, during the detection of the event *equal*, a user-defined event *evtEqualDetected* is fired when a second horizontal stroke is detected⁴. With Hasselt, both the raising of a user-defined event from a composite event and its capture and processing in the other conflicting composite event have to be manually described.

Recursive version of the problem

The second manifestation of the selection ambiguity problem occurs when one gesture is the initial part of another gesture. In this case, the common prefix would be the “smaller” of the two gestures. The well-known *double-tap* is the simplest example of this problem: when double tapping on a touchscreen device, one does not want the frameworks to notify the first tap but the se-

⁴Selection ambiguity problem with Hasselt: <https://youtu.be/rMwZbXStGRw>

quence of two taps only. More complex cases are those that require specifying arbitrarily long sequences of similar gestures, e.g. a number gesture can be specified as an arbitrarily long sequence of 0-9 digit gestures arriving in a quick succession, just as the two taps in the *double-tap*. In these cases, the selection ambiguity problem appears multiple times, e.g. a 0-9 digit gesture has the same starting sequence of events as a 2-digits number gesture, and these two gestures, in turn, have the same starting sequence as a 3-digits number gesture, and so forth. To the best of our knowledge, this second scenario has not been considered within the definition of the selection ambiguity problem. A distinction, however, is important because this second problem requires a different treatment than the first one. Once the common prefix is passed, there is no other gesture that will disambiguate in favor of the “smaller” gesture, except for a period of user inactivity, whose end can be naturally implemented as a timeout event. In the remainder of this thesis, we will refer to this second problem as the *recursive version* of the selection ambiguity problem.

The GestIT specification of the *double-tap*, shown in [Spano 13a], uses time constraints instead of timeout events. Concretely, GestIT must evaluate a user-defined time constraint in order to determine whether the two taps are close enough in time. The problem with this specification is that the constraint is checked only when the second tap is performed, which may be too late (e.g. maybe the user just wanted to tap only once) for notifying the first tap as an autonomous, self-sufficient *single-tap* event. The only way to implement both a single tap and a double tap with GestIT is by writing one definition per gesture. This may not be a problem for the single-and-double-tap case, but, when one needs to specify number gestures (unknown amounts of 0-9 digit gestures), for instance, writing one specification for each possible “gesture size” may be unfeasible.

Unlike GestIT, Hasselt allows describing one arbitrarily long sequence of gestures of the same type (e.g. taps, flicks, 0-9 digits) with one specification only. This is because of its capability to generate and handle timeout events. In the concrete example of the *double-tap*, Hasselt programmers can indicate Hasselt UIMS to activate a timer (of a user-defined length) once the first tap is detected. In the same specification, programmers can also indicate Hasselt UIMS to notify the *single-tap* unless the second tap manages to make its appearance before the expiration of the timer, in which case the *double-tap* should be fired. Instead of just passively waiting to verify a time constraint, Hasselt UIMS can be more proactive and activate timers that will eventually fire timeout events that Hasselt UIMS itself will recognize. When properly exploited, the capability of Hasselt UIMS for managing timeout events leads

to efficient specifications for the recursive version of the selection ambiguity problem. One example of such a specification was explained in Section 5.3.2 for the case of a touch gesture composed of an arbitrary number of horizontal flicks arriving in a quick succession. The execution of such a specification and other similar recursive specifications can be seen in a publicly available video⁵.

9.3.2 The problem of dual-faced gestures

By investigating about the *selection ambiguity problem*, we came across another complex problem. We found this problem when trying to specify a system response to the gesture *plus* (+), which was defined as a composition of a horizontal stroke followed by a vertical stroke. Although each stroke fires a particular system response when performed individually, the whole *plus* gesture should not fire such system responses; rather, it should activate its own distinct system response.

In this example, the selection ambiguity problem is solved by preventing conflicts between the horizontal stroke and the gesture *plus* (both have the same starting prefix), but it does not say anything about the damage that can be caused when the vertical stroke is detected. If this vertical stroke was meant to be part of the gesture *plus* but the framework fails to notice it, the response associated with the vertical stroke will be conveyed, which may potentially disturb the completion of the *plus* gesture. The response to the vertical stroke should not be conveyed unless the framework is sure that this is not part of a *plus* gesture, neither at the beginning, nor at the end. Whereas the traditional version of the selection ambiguity problem refers to conflicts in the starting sequence of events of multiple gestures, this problem refers to conflicts throughout the whole lifecycle of a composed gesture. Whereas the recursive version of the selection ambiguity problem refers to gestures composed of similar “pieces” (e.g. several instances of a tap or a 0-9 digit), in this problem, the composed gesture may be formed by different pieces, each with a different gesture specification (e.g. a horizontal stroke and a vertical stroke).

We solved the problem of the *plus* gesture by viewing each of its constituent strokes as a gesture that can fire different responses depending on whether it is performed alone or as part of the *plus*. When the horizontal (vertical) stroke is performed alone, a voice message ‘*horizontal stroke*’ (‘*vertical stroke*’) is synthesized by the system. But, when the horizontal stroke is quickly followed by a vertical stroke, the horizontal stroke does not fire

⁵Selection ambiguity problem (recursive) with Hasselt: <https://youtu.be/J0-0BhbC2Lk>

any response whereas the vertical stroke fires ‘*plus gesture*’. This approach can be implemented by combining two techniques: raising user-defined events (as used to solve the traditional version of the selection ambiguity problem) and raising timeout events (as used to solve the recursive version of the selection ambiguity problem). The unambiguous specification of the strokes and *plus* gestures is presented in Appendix C.5; its execution can be watched in a publicly available video⁶.

The problem of elaborating conflict-free specifications for a composed gesture and for all its constituent gestures will be referred to as the dual-faced gesture problem. This name is because our solution views each elemental gesture as if it has two different behaviors (when alone and when part of a composed gesture), but it is not excluded that other approaches can succeed too.

Besides being a challenge to other specification languages, solving the dual-faced gesture problem can bring about practical benefits too. Since a set of gestures can be composed in many different ways (n gestures can be combined in $n!$ different manners), a small set of simple, easily recognizable gestures can be exploited so that not only each gesture of the set can be associated with a system response, but also every possible composition of these gestures can trigger its own distinct system response. In this way, the gesture vocabulary of a system can be increased, not necessarily by adding more and more, difficult-to-specify, difficult-to-recognize gestures in the repertoire of the system, but also by considering each composition as a new, non-conflicting gesture.

9.4 Contributions of the thesis

The creation and evaluation of a framework capable of detecting and handling user-defined event patterns resulted in several contributions for future UIMS developers. These contributions are classified as follows:

9.4.1 Contributions in the tooling

At the level of tooling, this research produced the following novelties.

A composite event-based language

Hasselt is a family of languages that allows declarative descriptions of a wide variety of interactions (Chapters 4-6). Hasselt and its UIMS permit creating

⁶Dual-faced gestures problem with Hasselt: <https://youtu.be/Z09vn2eWrXk>

prototypes capable of generating different types of feedback (e.g. lexical and semantic), handling different types of user errors (e.g. lapses, slips, mistakes, and violations), and supporting different interaction styles (e.g. multimodal, multitouch, cross-device) subjected to different types of constraints (e.g. spatial, temporal, and semantic).

Unlike other similar languages, Hasselt retains the textual and event-driven nature that characterizes mainstream event languages such as Java and C#. This design decision was to give Hasselt a low threshold of use, to permit programmers to keep using the same concepts and follow similar work practices than the ones acquired along the long dominion of the event-driven paradigm. With Hasselt, the same as with mainstream event languages, programmers implement interactions with textual notations, by writing event-handling callbacks that are to be bound to a set of events so that, at runtime, the callbacks will be automatically fired when their associated events occur. The difference is that the one-to-one mapping of events to event-handling callbacks supported by existing event languages is extended by Hasselt to a many-to-many mapping: Each composite event, which is a combination of many events, can be bound to many event-handling callbacks that will be fired automatically in the right moment of the interaction.

A multipartite editor

Since the CEDL code and the SRDL code are strongly coupled, we integrated the CEDL editor and the SRDL editor into one single windows form (see Figure 3.4 on page 43). These code editors were accompanied by an auto-refreshing frame, which always displays the finite state automaton (FSA) associated with the composite event under definition. In this way, within one single window, programmers can see three important aspects of a human-machine interaction: the user actions (encoded in the CEDL editor), the system responses (encoded in the SRDL editor), and the interaction states (encoded in the FSAs). Our multipartite editor “glues” three logically separated concepts into one single physical space. Whereas the logical separation of concerns supported by Hasselt allows reasoning about each aspect of the interaction separately, which often makes programming less complex [Hürsch 95], the possibility of observing all these aspects within the same window minimizes the effort of switching attention between them.

Online feedback through parallel, visual animations

Many UIMSs [De Boeck 09, Navarre 09, Dumas 10] generate textual log files, which register the user events occurred at runtime. The analysis of these logs is valuable when trying to find bugs in the interaction models at a post-runtime stage. On top of this textual feedback, Hasselt UIMS also provides a new type of debugging tool, which is more convenient for rapid prototyping.

The automata view, shown in Figure 3.8, AV on page 50, provides online feedback that shows how the tracking of the composite events is going on. It shows how the execution state of the system changes as the human-machine interaction evolves. Thus, programmers can detect problems in their interaction specifications, at runtime, instead of having to wait for a post-runtime analysis, which is when log files are more useful. Second, the simultaneous animations managed by the automata view are better aligned with the intrinsic parallelism of multimodal interactions. In contrast, log files, as any text document, are inherently linear, i.e. read in a linear direction, and cannot describe parallelism in a natural way. Assuming that the user clicked the mouse and this altered the state of many interactions involving mouse clicks, the automata view will show all these changes in parallel: a series FSAs will have a new node as their current state. By using log files, the impact of the same mouse click can be discovered by navigating line by line, downstream through the text, until finding a series of lines that have the same or a similar timestamp, which is the proof of their co-occurrence. Related to the above, log files used to accumulate all events in one single document whereas the automata view with its multiple FSAs can show the impact of each event on several interactions, separately. Finally, the automata view allows debugging both the interaction model and the dialog model simultaneously. Since the dialog model is a user-defined FSA annotated with composite events, we decided to include it in the automata view so that programmers do not have to switch attention between separate online debugging tools, which would signify more mental workload for programmers and higher chances for bugs to slip away.

Not surprisingly, most of the errors we made ourselves when creating interaction models with Hasselt were more easily and quickly identified with the automata view.

9.4.2 Contributed algorithms

The algorithm that we want to highlight as a contribution herein is Algorithm 1, which transforms composite events into finite state automata (FSA), as exposed in Section 3.3.1 on page 52. Given the specification of the user

actions during his interplay with the system, the aforementioned algorithm infers all the states that will be relevant during the interaction so that programmers can specify in a direct, declarative fashion: *in this state* \rightarrow *call this function*. This is an important difference with other existing approaches.

With mainstream event languages, the interaction state has to be encoded on a multitude of variables and flags that must be maintained in a self-consistent manner and across several handlers, which requires lots of programming effort. With some research languages, instead, the interaction states can be explicitly depicted in the interaction models (e.g. as nodes of a FSM-like diagram). In these models, the system responses to be conveyed can be associated to both the incoming event and the interaction state, which no longer has to be deduced from the state variables (i.e. programmers no longer have to write conditional clauses to identify each possible interaction state). But even in this latter case, programmers still have to make a mental effort while designing the interaction: they have to identify, in their minds, the relevant states of the interaction, and draw them in the model, a task that is automatically performed by the aforementioned algorithm.

9.4.3 Contributions in engineering

In the domain of gestural interfaces, we noticed that there were two different problems hidden within the broad definition of *selection ambiguity problem* [Spano 13b], and that only one of these two problems had been discussed by their proponents. Additionally, we found and solved one instance of a new type of problem that may be more complex than the two aforementioned ones. Besides of being a challenge for specification languages, these problems can also serve as a guidance to keep tool developers aware of the problems that must be overcome by their tools.

Recursive version of the selection ambiguity problem

The selection ambiguity problem, as proposed in [Spano 13b], consists of elaborating unambiguous specifications for a set of gestures that start with the same sequence of events. The case addressed and discussed by the authors is the case when the conflicting gestures have the same prefix and the gestures are larger than this common prefix.

We made explicit another case that occurs when the common prefix is one of the conflicting gestures –think of the *single-tap* and *double-tap* as two conflicting gestures starting with the same prefix, a tap. In this case, unlike

the previous one, once the prefix is passed, no other event will come to disambiguate in favor of this gesture. This different scenario requires a different solution, and tool developers must be able to distinguish these two problems so that they can work on specialized solutions for each case. The solution we used, by the way, was by firing and catching timeout events, as already discussed in Section 9.3.1.

Dual-faced gestures problem

This problem consists of elaborating unambiguous, conflict-free specifications for a set of elementary gestures and a composition of these gestures. Each elementary gesture must fire some specific system response when performed individually, but this response must be inhibited when the elementary gesture is performed as part of the composed gesture, which has its own, distinct system response. The name of the problem is because the elementary gestures have a dual behavior, which depends on whether they are performed alone or as part of another gesture. This scenario can potentially raise conflicts at the beginning, in the middle, and/or at the end of the composed gesture lifecycle. In our case, the solution of this problem required combining the techniques used for the two versions of the selection ambiguity problem, namely user-defined events and timeout events.

9.4.4 Contributions in user study design

To the best of our knowledge, none of the languages developed for prototyping multimodal systems (Section 2.2.1 of Related Work) was evaluated in user studies. With regard to the domain of gesture recognition, only Proton++ [Kin 12a] was evaluated, but even in this case, participants were not asked to write code, just to interpret it. Since this thesis aims at identifying the benefits and limitations that can be brought about by a composite event-based language in comparison with mainstream event languages, the user study was elaborated to compare the efficiencies of programming with versus without composite events.

As discussed in Chapter 8, our user study combines observations, standardized questionnaires, and interviews in order to measure the completion rate, completion time, code testing effort, and perceived difficulty of a programming task as well as the perceived usability and perceived learnability of the programming environment.

Furthermore, the same chapter identifies the threats that may jeopardize the validity of our results and enumerates the lessons learned during the study.

This information is to allow future researchers to find potential pitfalls and improve our experiment design.

9.5 Limitations of Hasselt UIMS

The limitations of each individual language, i.e. CEDL, SRDL, and HMD2L, were already exposed at the end of Chapter 4, Chapter 5, and Chapter 6, respectively. This section discusses the limitations of the UIMS as well as the limitations of the approach.

9.5.1 Fixed set of atomic events

Hasselt UIMS serves to prototype a specific subset of multimodal systems, namely single-user multimodal systems using a fixed set of input modes such as mouse clicking, key pressing, speech, touch gestures, and body movements.

For each of these modalities, Hasselt provides a predefined set of events, each with a predefined set of event parameters, as was shown in Table 4.2 on page 60. Hasselt programmers cannot extend this set of atomic events (e.g. to include haptic events) by using external configuration files, DLLs, or add-ins. Hasselt programmers cannot do too much to redress this limitation. They are restricted to define their interactions with this relatively small set of events or to ask the UIMS developer to add more atomic events. It is true that Hasselt allows creating user-defined events (Section 5.1.5), but the limitation we are discussing herein is about inputs events, events generated by hardware.

Moreover, the predefined atomic events are also limited at the level of event parameters. For instance, whereas the speech events (raised by a commercial event-driven framework) usually comes with a list of strings reflecting the potential utterances pronounced by the user, the speech events of Hasselt only include the most likely utterance.

We never felt a strong necessity of providing Hasselt programmers with the possibility to extend the set of atomic events from configuration files because it was not related with the main goals of this research. This research tries to prove the advantages of combining events, which can be well evaluated with the 18 events provided in the current version of Hasselt, given that these events are generated by a group of hardware devices, each with a clearly different behavior from the other. It is obvious that in order to let Hasselt UIMS grow to a commercial product, making the set of input events extensible by the programmer is a necessity.

9.5.2 Inability to describe two-handed multitouch gestures

Simple multitouch gestures, such as two touches flicking in a vertical direction, can be implemented by modeling one composite event per finger. Other more complex multitouch gestures, such as a variable number of touches enlarging an object, can be easily implemented by using aggregate data (e.g. average distance between touches), which Hasselt UIMS calculates based on all the touches. But other more complex gestures like rotating two objects with two hands, each using a variable number of fingers cannot be implemented. This would require refining the idea of aggregate information, exposed in Section 5.3.3 on page 85, to enable Hasselt to generate aggregate information per cluster of touches.

9.5.3 Negative consequences of separating interaction code from application code

Unlike the previous ones, the following limitations are strongly related to the separation of application code and interaction code.

Inability to create EXE files

As mentioned in the Introduction, Hasselt UIMS can load a running system by “glueing” the interaction code, described with Hasselt, with the application code, described in a .NET language, but these two essential pieces of code cannot be merged into an autonomous, executable file. CoGenIVE [De Boeck 09], the supporting tool of NiMMiT, can generate executable files, but these files only encode the interaction code, i.e. the executable file still has to invoke the back-end application from time to time. In order to generate autonomous, executable files, Hasselt has to be extended to include general purpose constructs (sequence, selection, iteration) so that programmers can define both application and interaction code within Hasselt UIMS. Obviously, this requires the challenging task of extending the compilers of Hasselt UIMS so that they can interpret application code.

Searching for bugs in two different places

The debugging tools of Hasselt UIMS allow tracking the execution state of Hasselt programs, but not back-end applications. This is because the back-end applications are developed externally, without Hasselt support. Therefore, the variables of the back-end application are not visible to Hasselt UIMS.

Analogously, the back-end application can be independently inspected with the debugging tools provided by MS Visual Studio; but these tools cannot track the variables declared in Hasselt programs.

It would be convenient to develop debugging tools that can inspect both environments at once. That is, tools where the variables of both the interaction code and the application code can be shown at runtime within one single window.

Information between Hasselt UIMS and back-end applications have to be serialized

Passing information from Hasselt to the back-end applications and vice versa is not always a simple task.

First, in the direction from Hasselt UIMS to the back-end application, Hasselt UIMS can only pass (lists of) primitive values to the back-end applications. The data structures used by Hasselt to carry information about the skeleton joints (Table 4.3) or the aggregated data of the touchscreen device (Table 4.4) can still be passed to the back-end application, but as sets of primitive values (e.g. several x - y - z tuples, one for each joint), which may damage programming efficiency, a crucial aspect in the prototyping phase. This limitation is because the back-end application ignores (thus, it is not ready to receive) the types of internal objects that Hasselt UIMS handles.

In the other direction, the back-end application can only return primitive values but not instances of user-defined datatypes (e.g. class Student, class Country), because Hasselt UIMS has no clue about the structure of such classes. In order for the back-end applications to pass objects of such classes, the objects must be serialized, that is, their information has to be encoded as a string that will be later broken up by Hasselt programmers (see Section 5.1.4 on page 78). It may be technically possible for Hasselt UIMS to capture objects of user-defined datatypes as if these were instances of a generic Object class, but this is not implemented in the current version.

Back-end applications play a passive role

The interaction states, represented in the auto-generated FSA, are defined by taking only the user inputs into account. This unveils one underlying assumption of the proposed tool: Hasselt UIMS assumes that the back-end applications play a passive role in the interaction. If the back-end application had to perform a long computation (e.g. 10 mins) along which it will fire events that are supposed to affect the course of the interaction, Hasselt programmers

would have to coordinate with the application developer in order to reengineer the system. This issue would not exist if the events triggered by the back-end applications were visible to Hasselt UIMS, in which case, these events could be used as part of the composite events definitions.

9.6 Summary

The present chapters started by describing how the two goals proposed in this thesis were accomplished. With regard to the first goal, i.e. the design of a composite event-based language, the design decisions made to create Hasselt were exposed (why is it textual? Why is it event-driven? Why do the CEDL, SRDL, and HMD2L are as they are?). As to the second research goal, i.e. the assessment of the language, the results obtained from both code inspections and user study were integrated.

The chapter then presents the list of contributions made during this research. These were grouped into different categories, namely contributions in tooling, in algorithms, in engineering, and in user study design.

Finally, in order to give an idea of the scope of the research, the limitations of Hasselt UIMS were presented. Many of these limitations were strongly related with the fact that the interaction code (Hasselt code) is separated from the application code (.NET code).

Chapter 10

Conclusions and Future Work

As discussed in the previous chapter, this PhD research managed to design, implement, and evaluate a language that simplifies the creation of multimodal prototypes by saving programmers from the error-prone task of maintaining the interaction state, which is mandatory when using event languages. This last chapter summarizes the current status of the research, i.e. what has been done and what still remains to be done.

10.1 Conclusions

This research has designed, implemented, and evaluated Hasselt, a family of declarative languages that allows programmers to bind user-defined event sequences, herein called composite events, to one or more event handlers. This many-to-many mapping of events to event handlers is a generalization of the one-to-one mapping allowed by mainstream event languages, such as Java and C#, which restrict programmers to bind one event to one event handler only.

As discussed in Part I, Hasselt is composed out of three languages: The Composite Event Definition Language (CEDL) enables programmers to define composite events through a set of operators that can be applied in a recursive manner. Both the constraints among the constituents of a composite event and the event-handling callbacks to be launched in response to the (partial) detection of composite events are declared with the System Response Definition Language (SRDL). Finally, the higher-level Human-Machine Dialog Definition Language (HMD2L) can be optionally used to declare human-machine

dialogs, high-level models of interaction where the system responses depend on the current context-of-use. Hasselt comes accompanied with a supporting tool, Hasselt UIMS, that provides the code editors, compilers, runtime environment, and debugging tools required to write, syntax-check, run, and test Hasselt programs.

Hasselt enables composing several types of events (speech, touch, timeout, or body movement events), which can be subjected to different types of constraints (temporal, spatial, semantic). Despite its declarative nature, Hasselt allows implementing relatively complex functionalities, such as the smoothing of imprecise inputs (e.g. by filtering the inputs generated by Kinect). The expressivity of Hasselt also allows tackling difficult engineering problems such as the selection ambiguity problem and the dual-faced gesture problem, proposed herein. The prototypes created with Hasselt can support different interaction styles (multimodal, multitouch, and cross-device interaction), generate different types of feedback (lexical and semantical), handle several types of user error (lapses, slips, mistakes, and violations), and provide specialized treatments to parallel user inputs depending on whether these are complementary, redundant, or equivalent.

In Part II, through code inspection, it was revealed that the combined use of CEDL and SRDL save programmers from using state variables, and therewith, from the difficult, error-prone task of maintaining the interaction state across different event handlers. But, in the down side, it was shown that CEDL and SRDL offer a low-range for code refinement, in partly due to their declarative nature, and in partly due to suboptimal design decisions. The same code inspection also revealed that HMD2L models include fewer, simpler, and better centralized conditioned clauses than equivalent dialog models described with event-callback code. But HMD2L models suffer from hidden dependencies and high viscosity.

The evaluation of Hasselt also included empirical studies: two experiments were carried out to compare the programming efficiency of Hasselt and C#. The results of the first experiment showed that the modification of an interaction model with Hasselt leads to higher completion rates, lower completion times, and less code testing than when using C#. In a second experiment, after comparing the efficiency of HMD2L and C# for modifying multimodal dialog models, we could only conclude that the perceived ease of use was a bit higher for HMD2L than for C#.

Finally, Part III discussed the design decisions behind Hasselt, identified the contributions of this PhD research, and enumerated the limitations of Hasselt UIMS.

10.2 Future Work

This research can evolve in the following four directions, namely, evaluation methods, applied research, exploration of alternative directions, and upgrading towards a full-fledged general-purpose composite event-driven language.

10.2.1 Evaluation methods

User studies for gesture interaction

We managed to prove that describing context-independent speak-and-mouse interactions (i.e. first experiment) with Hasselt led to higher completion rates, lower completion time, and less code testing. It would be interesting to evaluate whether these programming benefits are repeated when prototyping multitouch gestures and body gestures.

More user studies for dialog modeling

The visual HMD2L created for elaborating high-level models of human-machine dialogs did not meet our expectations. The use of this declarative language did not show any significant gain over prescriptive event-callback code. We think that the main reason why no clear winner emerged from this study is that the task was too simple given the programming experience of the participants. Thus, it would be convenient to repeat the second experiment (HMD2L versus C#) with more complex programming tasks.

10.2.2 Applied research

At this moment, a plan is being elaborated to use Hasselt in the ClaXon project¹, an applied research aiming at optimizing the interaction between robots and humans by enabling safer operations and higher efficiency.

In the plan, the cobots (co-working robots) will be commanded through different hand gestures; each gesture is intended to command the cobot so that it can support the worker with some difficult/dangerous task (e.g. handing parts which are heavy or at a high temperature). Since not every worker is supposed to have the same level of authority over the cobots, it is important to recognize who is performing the hand gestures. This will require combining face recognition with hand gesture recognition. Furthermore, since safety is a critical issue in human-robot environments, it is part of the plan to use cameras

¹<http://www.iminds.be/en/projects/2015/03/11/claxon>

to make the robots aware of the ever-changing positions of their human co-workers. In this way, the robots can slow down or stop in reaction to the close presence of workers, for instance. As shown in this thesis, Hasselt has proven to be effective for combining modalities and for recognizing passive modalities, both are required to implement the aforementioned scenarios.

10.2.3 Exploring alternative directions

A new member of Hasselt?

In a survey study [Cuenca 14a], we classified several UIMSS by the capabilities of their frameworks (Figure 10.1). In that survey, the potential functionalities of a framework were: (1) recognition of user inputs, (2) fusion of inputs, (3) management of human-machine dialog, and (4) fission of outputs. It turns out that Hasselt UIMS fits into the second category, called State-based. Hasselt UIMS incorporates several software recognizers that perform recognition of inputs. Based on the CEDL and SRDL code, Hasselt UIMS can fuse the information carried by several multimodal events (e.g. to evaluate guard conditions, to accumulate it into arrays, or to pass it to back-end applications). Through the HMD2L, programmers can delegate the management of a human-machine dialog to Hasselt UIMS. However, Hasselt does not offer notations for fission of outputs: Programmers cannot delegate the coordination of multiple outputs to Hasselt UIMS; they have to write some multi-threading code in the back-end application for this task. But with a new Composite Response Definition Language (CRDL), programmers would be able to declare how multiple system responses are to be conveyed in a coordinated manner, e.g. `speak:'Cusco' + call:highlight('Cusco');` `speak:'belongs to';` `speak:'Peru' + call:highlight('Peru')`. The need of such language was predicted in the first stage of this PhD project, right before Hasselt started to be built, as attested in [Cuenca 13a]. The CRDL can still be considered as a part of a future project.

CEDL as a formal language?

After the CEDL and its supporting tooling were implemented, we noticed the existence of formal notations for describing interactions in concurrent systems, e.g. process algebra [Baeten 05]. It is not clear whether these formal notations would generate a larger interaction space than CEDL, i.e. whether they would allow describing interactions that are impossible to describe with CEDL. But, what is clear is that unlike our CEDL, such formal notations are supported

	Recognition of inputs	Fusion of inputs	Human-machine dialog management	Fission of outputs
Icon	✓	✗	✗	✗
OpenInterface	✓	✗	✗	✗
Squidy	✓	✗	✗	✗
Mengine	✓	✓	✓	✗
CoGenIVE	✓	✓	✓	✗
HephaistK	✓	✓	✓	✗
PetShop	✓	✓	✓	✓
Hinckley	✓	✓	✓	✓

■ Flow-based ■ State-based ■ Token-based

Figure 10.1: Functionalities that can be delegated to a UIMS [Cuenca 14a]. Has-selt UIMS would fit in the state-based group.

by a mathematical apparatus that has been elaborated after decades of study. If such notations were used, instead of CEDL, to describe interactions, in theory, they would permit formal reasoning about interaction descriptions. This potential gain has to be confirmed by future research.

10.2.4 Towards a composite event-based language

Augmenting SRDL with general-purpose constructs

The fundamental constructs of general-purpose languages (i.e. sequence, selection, and iteration) should be used to enrich the expressiveness of SRDL.

(1) SRDL does not allow defining sequential instructions. Declaring interleaved assignment statements and functions calls cannot be performed with SRDL; all system responses associated with a node or with a link are executed in parallel. (2) SRDL only allows one basic case of selection: all the system responses associated with a node or with a link are executed or not depending on a guard condition. More sophisticated methods of selection are not possible, e.g. to map different sets of systems responses to different guard conditions. (3) SRDL does not allow iterating over a set of statements: the system responses associated with a node or with a link are executed only once.

After adding these three fundamental constructs into SRDL, each node and link of a FSA (i.e. each interaction state) no longer has to be associated with a limited set of predefined actions (e.g. calling a function, raising an event, assigning variables), but with general-purpose code.

Overcoming Hasselt's event-blindness

Currently, Hasselt can call the methods included in the imported back-end applications; but it cannot sense the events generated by these applications. Redressing this event-blindness may solve two of the limitations mentioned in Section 9.5.

First, the set of atomic events, which is fixed in the current version of Hasselt, could be extended from existing libraries. New events (e.g. `ReadingChanged`) from new devices (e.g. accelerometers) can be used in the composite events definitions if Hasselt UIMS were capable of “seeing” the events declared in an existing DLL (e.g. Accelerometer API). Furthermore, by redressing the event-blindness of the presented tool, programmers were able to incorporate their favorite APIs (e.g. Kinect libraries) to refer directly to the events they are used to work with, e.g. event `SkeletonFrameReady`, instead of having to refer to the built-in events provided by Hasselt, e.g. event `kinect.skelpos`. The latter may cause that, eventually, programmers will stop using the predefined events of Hasselt to use their favorite libraries instead. This scenario will allow us to remove all predefined events and thus to lighten the Hasselt's core.

Second, the back-end application can start playing a more active role in the systems described with Hasselt. So far, if a back-end application triggers events during its invocation, those events will not be captured by Hasselt UIMS, and therefore, they cannot alter the course of the interaction. Hasselt views back-end applications as entities meant to perform computations and return a value; in formal terms, Hasselt prototypes fall into the computer-as-a-tool paradigm [Beaudouin-Lafon 04]. The limitation of Hasselt UIMS to capture the internally-defined events of the back-end applications can also be solved by a more comprehensive solution: by allowing programmers to encode application code into Hasselt UIMS, as discussed below.

Application code within Hasselt UIMS

Many of the limitations mentioned in Section 9.5 were strongly related with the fact that the interaction code and application code were separated. The ultimate solution for this problem is to augment Hasselt UIMS with a code editor for writing application code, which has to be compiled by Hasselt UIMS. In this way, programmers no longer have to import externally developed back-end applications whose user-defined data types, internally-defined events, and internally-defined variables are difficult to “see” from Hasselt UIMS or, in the best case, can be “seen” after some special treatment that increases the com-

plexity of the Hasselt's core. If Hasselt UIMS were able to compile application code, it would entail that every class, method, and variable, which are now included in the back-end applications, can be directly referred from the CEDL/SRDL code and vice versa. Hasselt was designed in line with this vision.

10.3 Long term vision

10.3.1 The need of a guiding star. Stopping the Babel-like confusion of languages

In a seminal paper [Myers 00], Myers et al. mentioned the possibility that a new paradigm may be needed to overcome the inadequacy of event languages for describing interfaces using speech and gestures. To the best of our knowledge, after 15 years, nobody has given a clue about how this new paradigm would look like. We think that this is a major omission since having a target would have helped researchers to channel their efforts in the same direction. Instead, the community has produced a series of reciprocally-different multimodal interaction description languages that vary from block diagrams to Petri nets, from visual to textual, from event-driven to logic-based, etc. It has been almost a rule that once a new language is developed and presented to the academic community, it falls into disuse and a new language starts to be developed from scratch to repeat the same cycle. In order to finish with the brainstorming stage in which the community has been immersed for 15 years, researchers must make an effort and try to start delineating the new programming paradigm that is to be guiding star for future language developers.

Some clues about the new paradigm can be learnt from the past.

10.3.2 Extrapolating the past for envisioning the future

Several UIMSs and event languages, including MIKE [Olsen Jr 86], University of Alberta UIMS [Green 85], and Sassafras UIMS [Hill 86], were proposed by academia in the middle 1980s, when imperative code was commonly used [Myers 92, DeMarco 89]. These UIMS and their specification languages inspired the creation of commercial event languages like Visual Basic [Myers 00], released in 1991.

We want to highlight two important facts of the then-newly created commercial event languages. First, these new languages reduced the accidental complexity of creating WIMP systems with the imperative paradigm. For instance, commercial event languages and their supporting frameworks gave

programmers the opportunity to develop WIMP systems without having to implement the message loop² which was pivotal when using C or other imperative languages. Second, the new event-driven programming languages did not dispense with the then-dominating imperative paradigm; imperative code was still used within the event handlers.

We think that the history is going to be repeated again and a new type of programming language will emerge from the languages and UIMSs studied in the Related Work section. As in the past, the new languages are going to be built on top of event-driven languages, which are now widely used for implementing interactive systems. That is why we suggest future UIMS developers to stop insisting with describing multimodal interactions with visual languages. These are obstructing the vision of the new paradigm, which is going to be built on top of the event-driven paradigm in the same way as this was built on top of the imperative paradigm in the past. Visual languages can still be useful to create high-level models.

10.3.3 Composite event-based programming. The guiding star

An event language with support for composite events, as outlined in Figure 10.2, b, could be the shadow of a language of the new paradigm. First, composite events reduce accidental complexity. They eliminate the need of tracking sequences of events, the error-prone, difficult task required when creating multimodal, multitouch systems with event languages. Second, the event-callback model is not thrown away in our envisioned composite event-based language, it is just extended. At a high level of abstraction, WIMP interactions can be viewed as one-to-one mappings of events to event handlers. With composite events, multimodal interactions are described as many-to-many mappings of events to event handlers.

This thesis has just provided one example of how to create tools and languages able to automate the detection and handling of user-defined composite events. But the future must see new ventures searching for new event operators, new ways of event binding (maybe dispensing with FSA), new notations (e.g. Petri nets instead of ATNs), and, in general, aspiring to identify and deal with the still unfound, deeper, and more subtle consequences introduced when extending the concept of event to composite event.

Future research on composite event-driven programming languages can nurture from the theory developed in other well-established fields such as Com-

²<https://msdn.microsoft.com/en-us/library/windows/desktop/ms644928%28v=vs.85%29.aspx>

```

/* binding user events to callback functions */
bAccept.Click += new EventHandler(bAccept_Click)

```

```

/* callback functions */
private void bAccept_Click(object sender, EventArgs e)
{ /* application-specific C# code */ }

```

(a) Event-driven language: one-to-one mapping of events to event handlers.

```

/* declaring composite events */
event putThatThere = speech.move ;
    speech.that + mouse.down<x1, y1>;
    speech.there + mouse.down<x2, y2>

```

```

/* binding composite events to callback functions */
wrt ce.putThatThere
    @node(5) do
        call:highlightHere (x1,y1);
    @node(8) do
        call:putThatThere (x1,y1,x2,y2);

```

```

/* callback functions */
private void highlightHere(int x, int y)
{ /* application code */ }

private void putThatThere(int x1, int y1, int x2, int y2)
{ /* application code */ }

```

(b) Composite event-driven language: many-to-many mapping of events to event handlers.

Figure 10.2: (a) Each user event can be bound to one event handler. (b) Each composite event, which is a combination of many events, can be bound to multiple event handlers.

plex Event Processing (CEP) and active databases (Appendix B). As with our research, both of these fields use composite events as the main concept.

A concentrated effort, which has not happened to date, may eventually lead to a new type of programming language and programmers no longer have to continue developing multimodal interactive systems with the tooling designed for the previous WIMP paradigm.

10.4 Summary

This thesis has presented a software solution that reduces the complexity of describing multimodal interactions with current event-driven languages. This solution consists of a family of declarative languages that empower programmers to define compositions of events, which can be later bind to one or more event handlers. Although the language we envisioned included not only notations for composite event binding, but also the general purpose constructs required to implement the application part of the intended system (Section 1.3), we had to prove first that the idea of automating the detection and handling of composite events was feasible and beneficial. We have done that with the present thesis.

There were two findings that encourage us to continue going in the direction of our envisioned language. First, when using event-driven programming as a baseline paradigm, composite events do bring about benefits in code complexity and programming efficiency, as attested by the code inspections and user study respectively. Second, the major limitations of the proposed UIMS comes from the fact that, as in any UIMS, the interface code and the application code are separated into two autonomous software pieces that have difficulties to exchange data and to refer to the internal variables of each other. These two facts push us in the same direction: to extend the proposed notations so that these can also describe application code, that is, to continue walking towards a full-fledge event language with support for composite events.

Appendices

Appendix A

Theoretical background

The present appendix starts by presenting the filter applied in the section 6.2.2. The said filter was completely implemented with Haskell showing that it has the syntax required to smooth the effects of noisy data.

Then, we show that the transition networks generated from CEDL code altogether fall within the definition of an Augmented Transition Network (ATN).

Finally, the appendix shows the algorithm used for transforming composite events into semantically equivalent finite state automata.

A.1 Exponential Smoothing Filter

An exponential smoothing filter, also known as an exponentially weighted moving average (EWMA), is a popular filter in many different fields. The exponential filter output is given by:

$$\hat{X}_n = \alpha X_n + (1 - \alpha) \hat{X}_{n-1}$$

Where α is called the dampening factor and $0 \leq \alpha \leq 1$. By substituting \hat{X}_{n-1} this can be expanded to obtain:

$$\hat{X}_n = \sum_{i=0}^n (1 - \alpha)^i X_{n-i}$$

Therefore, filter output at time n is a weighted average of all past inputs, where weights $a_i = \alpha(1 - \alpha)^i$ decrease exponentially with time (more precisely, with geometric progression, which is the discrete version of an exponential

function). Also, all previous inputs contribute to the smoothed filter output, but their contribution is dampened by increasing power of parameter $1 - \alpha$. Since \hat{X}_n depends on all past inputs, an exponential filter is said to have an infinite memory of all past inputs.

A.2 Augmented Transition Network (ATN)

The internal data structure with which Hasselt UIMS handles the human-machine interactions meets the characterizing features of an Augmented Transition Network (ATN) [Woods 70].

(I) *Arcs with arbitrary conditions.*- In an ATN, one can add to each of the transition arcs an arbitrary condition which must be satisfied in order for the arc to be followed [Woods 70].

The proposed SRDL, through the keyword *when*, allows attaching guard conditions to any arc of the transition network. This feature was exploited in Equation 5.5 for instance. There we only processed touch events if these were generated by the first finger.

(II) *Arcs with instructions for maintaining variables.*- Information about ATNs is encoded in registers, which are updated when the arcs are followed. The registers can be updated in terms of their previous values, the values of other registers, the parameters of the current input, and/or the output of a function. They may also be interrogated by conditions on other arcs [Woods 70].

The proposed SRDL, through the keyword *assign*, allows instantiating and maintaining arbitrary sets of variables. One example is given in the definition of *mclick* in Equation 5.3. There both variables *t0* and *t1* are set with the value of a function, *Now.TotalMilliseconds*, while traversing an arc; they are also subsequently evaluated in a triggering condition.

(III) *Recursiveness.*- The labels of any arc may include not only terminal symbols but also nonterminal symbols. When an arc labelled with a nonterminal symbol is about to be traversed, the control will jump to the subnetwork represented by that nonterminal symbol. Reaching the final state of this subnetwork will finally cause the transition represented by the arc [Woods 70].

Hasselt exhibits the aforementioned behavior every time it has to handle composite events defined in a compositional manner. For instance, in the *put-that-there* shown in Figure 5.1, every time the end user is expected to click the mouse, the control is temporarily transferred to the subnetwork *mclick* and returned to *moveObject* after the final state of *mclick* is reached.

Appendix B

Composite events in other domains

During the development of this research, we notice that the concept of composite events was already discovered and studied in other domains such as active databases and complex event processing (CEP).

Active databases appeared because the “passive” nature of traditional, relational databases (e.g. they only provide information once they are queried) damaged the efficiency of those applications requiring real-time information. Similarly, CEP engines responded to the constant appearance of a large number of distributed applications that required continuous and timely processing of information from the periphery to the center of the system [Cugola 12b].

B.1 Active databases as composite event processors

Timely recognition of complex event patterns may be helpful in a variety of situations: to detect frauds from credit card transactions, to warn of natural disasters by analyzing variations in climate, to discover misuse of an application by tracking its event log, etc. Those series of related happenings that reveal an occurrence of interest have been referred to as composite events in the field of active databases for more than two decades [Gehani 92, Gehani 94, Chakravarthy 94, Adaikkalavan 06]. More precisely, active databases, unlike traditional relational databases, are able to react to specific circumstances of relevance for an application [Paton 99].

The maturity reached after many years of academic research led to the development of commercial software that performs processing of complex events,

e.g. SAP Event Stream Processor¹, Oracle Event Processor², and IBM InfoSphere Streams³. For a company, composite events may signify threats or opportunities that require timely responses from the business.

B.2 Complex Event Processing (CEP) as a service

Complex event processing have also been implemented in middleware platforms that allow consumers to subscribe to certain event patterns. For instance, Sanchez et al. [Sánchez 03] implemented a middleware that notifies the occurrence of user-defined event patterns to avionics applications. The publish/subscribe system Cayuga [Demers 06] was able to search for patterns in a stream of RSS entries, e.g. blog entries, news headlines, etc. Anicic et al [Anicic 09] created a middleware along with its underlying language and tested it with real market stock values for the IBM company. And TRex [Cugola 12a] was tested with simulated data (e.g. air temperature, presence of smoke, humidity, etc.) from which the presence of fire had to be inferred.

There are also composite event detection servers beyond academia. Nowadays, companies can avoid the hassles of deploying, maintaining or scaling infrastructure by subscribing to SaaS services that embed composite event detection under the covers. RuleCore⁴ and Interstage Big Data Complex Event Processing Server⁵ are two of these servers, but the one that will be the greatest focus of attention will possibly be Google Cloud Dataflow. Google recently expanded its Cloud Platform with this new service⁶, which helps users to get actionable insights from examining real-time streams of data (events).

We found a major problem that prevent implementing multimodal interactions by exploiting existing CEP engines or active databases: these tools notify their subscribers only after the full detection of an event pattern. This is not enough to handle multimodal interactions. Here a system must be notified during the whole lifecycle of a complex event so that partial feedback can be provided. Moreover, delegating event pattern detection to a CEP engine or to an active

¹<http://www.sap.com/pc/tech/e/software/sybase-complex-event-processing/index.html>

²<http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/complex-event-processing-088095.html>

³<http://www-03.ibm.com/software/products/en/infosphere-streams>

⁴<http://www.rulecore.com/>

⁵<http://www.fujitsu.com/global/products/software/middleware/application-infrastructure/interstage/solutions/big-data/bdcep/>

⁶<https://cloud.google.com/>

database would require recording user events into a database first. This may damage the response time expected from an interactive system, and especially if the user events are generated at a high rate, e.g. when using Kinect or accelerometers.

Appendix C

Source code

This appendix shows the Hasselt code of a representative set of applications discussed throughout the thesis. It also includes the C# code required to create the back-end application used in the put-that-there example.

C.1 Feedback about the error recognition inputs

Specialized feedback can be generated for each type of error recognition input, as proven by a prototype implemented with the following code. The prototype was discussed in Section 5.4.1 on page 91. Its behavior can be seen in the following video¹.

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

wrt ce.moveObject
@node(1) do
  assign: msg2='what-do-i-have-to-move?';
@node(2) do
  speak: msg2;
@node(3) do
  assign: msg2='you-gotta-click-on-an-object', msg5='where-to-move?';
```

¹<https://www.youtube.com/watch?v=01ETIsCoMq8>

```

@node(4) do
  assign: msg2='i-didnt-hear-anything', msg5='where-to-move?';
@node(5) do
  call:PTT.frmPTT.HighlightObjectHere(x1,y1);
  speak: msg5;
@node(6) do
  assign: msg5='you-had-to-click-on-the-new-position';
@node(7) do
  assign: msg5='i-did-not-hear-anything';
@node(8) do
  call:PTT.frmPTT.PutThatThere(x1,y1,x2,y2);

```

C.2 Rolling back

Here we show the code of a prototype capable of undoing the highlighting and creation of an arbitrary number of objects in response to the reset command. The prototype can be seen in action in a publicly available video². More details about roll backing were discussed in Section 5.4.2 on page 92.

```

event mclick<x,y> = mouse.down; mouse.up<x,y>

event putThatThere = speech.move;
                    speech.that + ce.mclick<x1,y1>;
                    speech.there + ce.mclick<x2,y2>

event createObj = speech.create;
                 (speech.here + ce.mclick<x,y>)*;
                 delay-1500

event removeMany = speech.remove;
                  (speech.here + mouse.down<x,y>)*;
                  delay-1500

wrt ce.mclick<x,y>
  @link(1, mouse.down) do
    assign: t0 = Now.TotalMilliseconds;
  @link(2, mouse.up<x,y>) do
    assign: t1 = Now.TotalMilliseconds;

```

²Roll-backing with Hasselt: <https://youtu.be/zcKFgZTaFhw>

triggers when $t1 - t0 \leq 200$

```
wrt ce.putThatThere
  @node(0) do
    assign:comefromfinal=0;
  @node(1) do
    call:PTT.frmPTT.undoLastAction();
    when comefromfinal=0;
  @node(5) do
    speak:'where?';
    call:PTT.frmPTT.HighlightObjectHere(x1,y1);
  @node(8) do
    speak:'done!';
    assign:comefromfinal=1;
  call:PTT.frmPTT.PutThatThere(x1,y1,x2,y2);
```

```
wrt ce.createObj
  @node(0) do
    assign:comefromfinal=0;
  @node(1) do
    assign:wasLastUnstable=0;
    call:PTT.frmPTT.undoLastAction();
    when comefromfinal=0;
  @node(2) do
    call:PTT.frmPTT.CreateObjectHere(x,y);
    when x is not null and wasLastUnstable=0;
  @node(3) do
    assign:wasLastUnstable=1;
  @node(4) do
    assign:wasLastUnstable=1;
  @link(3, ce.mclick<x,y>) do
    assign:wasLastUnstable=0;
  @link(4, speech.here) do
    assign:wasLastUnstable=0;
  @node(5) do
    speak:'done!';
    assign:comefromfinal=1;
    call:PTT.frmPTT.fogetLastActions();
```

```
wrt ce.removeMany
  @node(0) do
    assign:comefromfinal=0;
  @link(1, speech.remove) do
    assign:wasLastUnstable=0;
  @node(1) do
    call:PTT.frmPTT.undoLastAction();
    when comefromfinal=0;
  @node(2) do
    call:PTT.frmPTT.RemoveThisObject(x,y);
    when x is not null;
  @node(3) do
    assign:wasLastUnstable=1;
  @link(3, mouse.down<x,y>) do
    assign:wasLastUnstable=0;
  @link(4, speech.here) do
    assign:wasLastUnstable=0;
  @node(4) do
    assign:wasLastUnstable=1;
  @node(5) do
    speak:'done!';
    assign:comefromfinal=1;
    call:PTT.frmPTT.fogetLastActions();
```

C.3 Management of of redundant inputs

This section show the code required to describe the interaction shown in a video³ discussed in Chapter 5.

The composite event *deleteNow* is triggered when the end user utters ‘*remove*’ and presses the key DEL at the same time. The user-defined event *evtAsk4Confirm* is triggered when only one of these two equivalent commands is detected. In this case, the system will continue waiting for a confirmation and the composite event *deleteConf* will therefore be completed.

The FSA of the *deleteNow* and *deleConf* were shown in Figure 5.6 on page 98.

```
event deleteNow = speech.remove + keyboard.keydown<c>
```

³<https://www.youtube.com/watch?v=B4Zwmw6MleI>

```
event deleteConf = ce.evtAsk4Confirm; speech.any<answer>

event createobjs = speech.create; speech.object

wrt ce.deleteNow
@node(0) do
  assign: finalreached = 1;
@node(1) do
  raise: evtAsk4Confirm;
  when finalreached = 0;
@node(2) do
  assign: finalreached = 0;
@node(3) do
  assign: finalreached = 0;
@link(1, keyboard.keydown<c>) do
  when c = 46;
@link(2, keyboard.keydown<c>) do
  when c = 46;
@node(4) do
  call:PTT.frmPTT.RemoveAll();
  assign: finalreached = 1;

wrt ce.deleteconf
@node(2) do
  speak:'Do you confirm?';
@link(2, speech.any<answer>) do
  when answer = 'of course';
@node(3) do
  call:PTT.frmPTT.RemoveAll();

wrt ce.createobjs
@node(3) do
  call:PTT.frmPTT.CreateObject();
```

C.4 Ambiguity of gestures *plus* and *equal*

The following code shows that Hasselt allows tackling the selection ambiguity problem, which appears when two or more gestures start with the same chain


```
wrt ce.plus
  @link(2, ce.top2down) do
    assign:isplussymbol=1;
    raise:evtplusdetected;
  @link(2, ce.evtequaldetected) do
    assign:isplussymbol=0;
  @node(3) do
    speak: 'plus symbol';
    when isplussymbol=1;
```

The composite events *left2right* and *top2down* are defined in separated files. What is important to know from these two events is that they will be triggered whenever the end user performs a horizontal click or a vertical flick, respectively.

As to the composite event *equal*, this is described as a sequence of two horizontal flicks (Figure C.1a). The description also includes the user-defined event *evtplusdetected*, which is manually fired from the composite event *plus* in order to notify that the symbol *plus* has just been drawn. As to the composite event *equal*, this is described as a sequence of two horizontal flicks (Figure C.1a). The description also includes the user-defined event *evtplusdetected*, which is manually fired from the composite event *plus* in order to notify that the symbol *plus* has just been drawn. At runtime, the event *equal* will move to *node(2)* every time a horizontal flick is perceived. And it will correctly go back to its initial node if *evtplusdetected* is detected at this stage: the occurrence of *evtplusdetected* will be indicating that the horizontal flick was just part of the symbol *plus* and this the event *equal* should have never moved to *node(2)*. Because the user-defined event *evtplusdetected* cannot throw *equal* directly to its initial state, it does that indirectly through the final state (every time a composite event reaches the final state, it will automatically goes back to the initial state). The flag *issymbolequal* is just to distinguish if the final state was reached because the symbol *equal* was fully performed or because the symbol *plus* was detected in the interim.

As to the composite event *plus*, its description is analogous to *equal*.

C.5 Dual-faced gestures problem

Section 9.3.2 described a complex problem therein called the dual-faced gestures problem, which was illustrated with an example that involves two gestures, namely a horizontal and a vertical stroke, which can be composed to

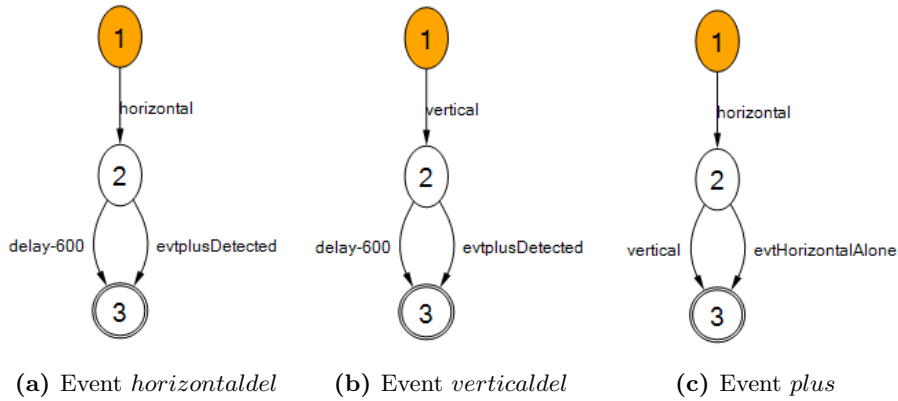


Figure C.2: Unambiguous specifications of three conflicting gestures, namely vertical flick, horizontal flick, and plus symbol

delineate a *plus* (+) gesture. The problem consists of specifying a vertical and a horizontal stroke so that each stroke can fire a different system response depending on whether it is performed individually or as part of the *plus* gesture. Our specification for this simple instance of the dual-faced gesture problem is shown below. This code uses a library containing the definition of *horizontal* and *vertical*, which are triggered when the horizontal and vertical strokes occur, respectively.

```

event horizontaldel= ce.horizontal; delay-600 | ce.evtplusDetected

event verticaldel= ce.vertical; delay-600 | ce.evtplusDetected

event plus= ce.horizontal ; ce.vertical | ce.evtHorizontalAlone

wrt ce.horizontaldel
  @link(2, delay-600) do
    speak:'horizontal-line';
    raise:evtHorizontalAlone;

wrt ce.verticaldel
  @link(2, delay-600) do
    speak:'vertical-line';
    raise:evtVerticalAlone;

wrt ce.plus

```

```
@link(2, ce.vertical) do
  speak: 'plus-symbol';
  raise: evtplusDetected;
```

As to the composite event *horizontaldel*, it will be fired 600 milliseconds after the occurrence of a horizontal flick unless the *plus* gesture occurs within this interim (Figure C.1a). The time interval of 600 milliseconds is an arbitrary period used to decide whether ‘something else’ will come after the horizontal line or not. If no events are detected after 600 milliseconds (i.e. if *evtplusDetected* never arrives), a timeout event will be triggered (i.e. the arc labelled *delay-600* in Figure C.1a will be traversed), and thus, the message ‘horizontal line’ will be synthesized and the user-defined event *evtHorizontalAlone* will be fired, as annotated in the timeout event.

As to the composite event *verticaldel*, it is analogous to *horizontaldel*.

As to the composite event *plus*, it will be fired when the vertical flick is detected right after the horizontal flick (Figure C.2c), in which case, the voice message ‘plus symbol’ (annotated in the arc labelled *vertical* in Figure C.2c) will be synthesized. But if there is a ‘silence’ longer than 600 milliseconds after the horizontal flick, as explained before, the user-defined event *evtHorizontalAlone* will be fired. This will move the event *plus* to its final state through an alternative way, lacking of system responses –the system responses are annotated in the arc labelled *vertical*, but not in the arc *evtHorizontalAlone* or in the final node.

C.6 Couch Potato

Part II closed with a comprehensive example showing that Hasselt UIMS can be used to prototype multimodal, multitouch, and cross-joint interactions. The composite events used to describe the interactions supported by Couch Potato (see Chapter 6) are shown below. Some of these composite events were already discussed in the thesis.

```
event xrighthand<xpos>= kinect.skelpos <j>

event fromR2L = ce.xrighthand<xpos1>;
               ce.xrighthand<xpos2>;
               ce.xrighthand<xpos3>

event fromL2R = ce.xrighthand<xpos1>;
```

```
        ce.xrighthand<xpos2>;
        ce.xrighthand<xpos3>

event wave = ce.fromR2L ; ce.fromL2R

event hi = ce.wave | speech.any<str>

event finish = kinect.useroff

event left2right =  tscreen.firston<x1,y1,t1,id1>;
                   tscreen.move<x2,y2,spx,spy,t2,id2>*;
                   tscreen.lastoff

event right2left =  tscreen.firston<x1,y1,t1,id1>;
                   tscreen.move<x2,y2,spx,spy,t2,id2>*;
                   tscreen.lastoff

event top2down =  tscreen.firston<x1,y1,t1,id1>;
                 tscreen.move<x2,y2,spX,spY,t2,id2>*;
                 tscreen.lastoff

event down2top =  tscreen.firston<x1,y1,t1,id1>;
                 tscreen.move<x2,y2,spX,spY,t2,id2>*;
                 tscreen.lastoff

event volumeDown = ce.top2down; ce.top2down*; delay-500

event volumeUp = ce.down2top; ce.down2top*; delay-500

event tap = tscreen.down<x1,y1,t1,id1>; tscreen.up<x2,y2,t2,id2>

event handfront = kinect.skelpos<sk1>

event playvideo = ce.left2right + ce.handfront

event stopvideo = ce.right2left + ce.handfront

event pausevideo = ce.tap + ce.handfront
```

```

event searchBySpeech = speech.any<str>

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

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

wrt ce.xrighthand<xpos>
    @node(0) do
        assign: prevxpos = 0;
    @link(1, kinect.skelpos<j>) do
        assign: xpos = 0.8 * j.HandRight.X + 0.2 * prevxpos;
    @node(2) do
        assign: prevxpos = xpos;

wrt ce.fromR2L
    @link(1, ce.xrighthand<xpos1>) do
        when xpos1 > 0;
    triggers when xpos1 > xpos2 and xpos2 > xpos3

wrt ce.fromL2R
    @link(1, ce.xrighthand<xpos1>) do
        when xpos1 < 0;
    triggers when xpos1 < xpos2 and xpos2 < xpos3

wrt ce.wave
    @link(1, ce.fromR2L) do
        assign: t1 = Now.TotalMilliseconds;
    @link(2, ce.fromL2R) do
        assign: t2 = Now.TotalMilliseconds;
    @node(3) do
        when t2 - t1 < 2000;

wrt ce.top2down
    @link(2, tscreen.move<x2,y2,spX,spY,t2,id2>) do
        when id1=id2;
    triggers when y2 > y1 and x2 - x1 < 0.05 and x2 - x1 > -0.05

```

```
wrt ce.down2top
  @link(2, tscreen.move<x2,y2,spX,spY,t2,id2>) do
    when id1=id2;
triggers when y2 < y1 and x2 - x1 < 0.05 and x2 - x1 > -0.05

wrt ce.volumeDown
  @link(1, ce.top2down) do
    assign:N=1;
  @link(2, ce.top2down) do
    assign:N=N+1;
  @node(3) do
    call:WinMediaPlayer.Form1.volumeDown(N);

wrt ce.volumeUp
  @link(1, ce.down2top) do
    assign:N=1;
  @link(2, ce.down2top) do
    assign:N=N+1;
  @node(3) do
    call:WinMediaPlayer.Form1.volumeUp(N);

wrt ce.hi
  @link(1, speech.any<str>) do
    when str in ('hello', 'hi');
  @node(2) do
    call:WinMediaPlayer.Form1.makePlayerVisible();

wrt ce.finish
  @node(2) do
    call:WinMediaPlayer.Form1.makePlayerInisible();

wrt ce.left2right
  @link(2, tscreen.move<x2,y2,spx,spy,t2,id2>) do
    when id1 = id2;
triggers when x2 > x1 and y2 - y1 < 0.05 and y2 - y1 > -0.05

wrt ce.right2left
  @link(2, tscreen.move<x2,y2,spx,spy,t2,id2>) do
```

```
    when id1 = id2;
triggers when x2 < x1 and y2 - y1 < 0.05 and y2 - y1 > -0.05

wrt ce.tap
  @link(1, tscreen.down<x1,y1,t1,id1>) do
    assign: tini = Now.TotalMilliseconds;
  @link(2, tscreen.up<x2,y2,t2,id2>) do
    assign: tfin = Now.TotalMilliseconds;
triggers when id1=id2 and tfin-tini <= 200 and x1=x2 and y1=y2

wrt ce.handfront
triggers when skl.HandRight.Z < skl.Head.Z - 0.35

wrt ce.playvideo
  @node(4) do
    call:WinMediaPlayer.Form1.play();

wrt ce.stopvideo
  @node(4) do
    call:WinMediaPlayer.Form1.stop();

wrt ce.pausevideo
  @node(4) do
    call:WinMediaPlayer.Form1.pause();

wrt ce.searchBySpeech
  @node(2) do
    call:WinMediaPlayer.Form1.scrollVideos(str);

wrt ce.digit<getBestMatch>
  @link(2, tscreen.up<xn,yn,tn,idn>) do
    call:gest2d.utils.getBestMatch(xs,ys,ts);
triggers when getBestMatch <> 'none'

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


C.7 Back-end application for the put-that-there

The back-end windows form used in the *put-that-there* example and whose GUI is seen in Figure 4.3a, was encoded as follows. Notice that this application does not require sensing the external environment since is done by Hasselt UIMS. Programmers only have to implement application-specific functionality.

A high percentage of the presented C# code is auto-generated by MS Visual Studio.

Algorithm 6 C# back-end code for the put-that-there system

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms

namespace ptt {

    [Description("Back-end of a PutThatThere system")]
    public partial class frmPTT:Form {

        public frmPTT()
        {
            InitializeComponent();
        }
        public void PutThatThere(x1, y1, x2, y2)
        {
            Button b = (Button) this.GetChildAtPoint(new Point(x1, y1));
            if(b!= null)
            {
                b.Location = new Point(x2, y2);
                b.FlatStyle = FlatStyle.Flat;
            }
        }
        public void HighlightObjectOn(x, y)
        {
            Button b = (Button) this.GetChildAtPoint(new Point(x, y));
            if(b!= null) b.FlatStyle = FlatStyle.Standard;
        }
    }
}
```

Appendix D

User study

The first part of this appendix will show the tutorial used for training participants before the user study. The second part will show the two instruction sheets participants used during the study. Then, we will give a brief overview to help the reader understand the statistical analysis carried out in this research. Finally, the appendix presents the raw data collected during the user study. Finally, the appendix provides

D.1 Tutorial for user studies

The tutorial consists of creating and running two Hello world-like examples with Hasselt. Similar to the user study, the tutorial is divided into two parts. The first part requires participants to use the textual languages CEDL and SRDL; the second part requires them to use the visual language HMD2L.

As it can be seen, the tutorial is quite simple. It does not give any definition about composite events or mention its advantages. It does not mention the syntax of Hasselt, e.g. that the variables within angular brackets are event parameters. The user is expected to infer this on the fly.

The goal of the tutorial is to provide participants with practical knowledge about how to edit, test, and debug Hasselt programs. Participants are expected to have a feeling of how to use the auto-completion popups, how to draw the nodes, links, etc.

The content of the tutorial is shown below.

EVALUATION OF HASSELT

Hasselt is a family of interrelated languages aimed at describing multimodal interfaces. It consists of:

- Composite Event Definition Language (CEDL)
- System Response Definition Language (SRDL)
- Human-Machine Dialog Definition Language (HMD2L)

In the first part of the experiment we are going to evaluate the CEDL and SRDL. In the second part, we will evaluate the HMD2L. The following is intended to get the user acquainted with the programming environment.

FIRST PART

A) My first program: Hello World!

Click on the option "Hasselt editor" of the "View" menu (or press Ctrl + H). The Hasselt's text editor shown in Figure 1 must be opened.



Figure 1. Hasselt's text editors for the CEDL and SRDL

- a) Use the CEDL editor to declare a composite event as follows:

```
event foo = mouse.down<x,y> ; keyboard.keydown<ch>
```

You are defining the occurrence of a mouse click followed by a keystroke as one composite event called *foo*. Click on the option "Parse Composite Events" of the menu "Debug" (or press Ctrl+1).

- b) Now, let's instruct the toolkit to synthesize the sentence 'Hello World!' after detecting the composite event *foo*. To do this, go to the SRDL editor and write

```
wrt ce.foo
  @node(3) do
    speak:'Hello world!';
```

Notice that after writing the first line, i.e. *wrt ce.foo*, a finite state automaton appears in the right bottom frame of the Hasselt's editor. The number 3 in *@node(3)* matches with the final state of this automaton. The term *wrt* stands for 'with respect to'; and *ce*, for 'composite event'.

- c) To verify the correctness of this code, click on the option "Parse Feedback Annotations" of the menu "Debug" (or press Ctrl+2). If an error window message appears, you must solve the issue.
- d) Click on the option "Run Prototype" of the menu "Debug" (or press F5).
- e) A 'green light' in the status bar will indicate that runtime mode is activated. Most importantly, every time a mouse click is followed by a keystroke, you will hear 'Hello World!'
- f) Click on "Stop Prototype" (or press Ctrl+F5). The green light will switch to 'red'.

How to debug Hasselt programs?

Open the Variable Browser (Ctrl + B), Event Viewer (Alt + V), and State Diagrams Viewer (Ctrl + M).
Now, run the program again (F5).

For the *foo* example, the variable browser will display the position of the mouse click and the ASCII code of the key that was pressed. The event viewer will list every event detected by the toolkit recognizers regardless of it is involved in the definition of *foo* or not. The state diagrams viewer displays animations indicating how the progressive detection of *foo* is going on.

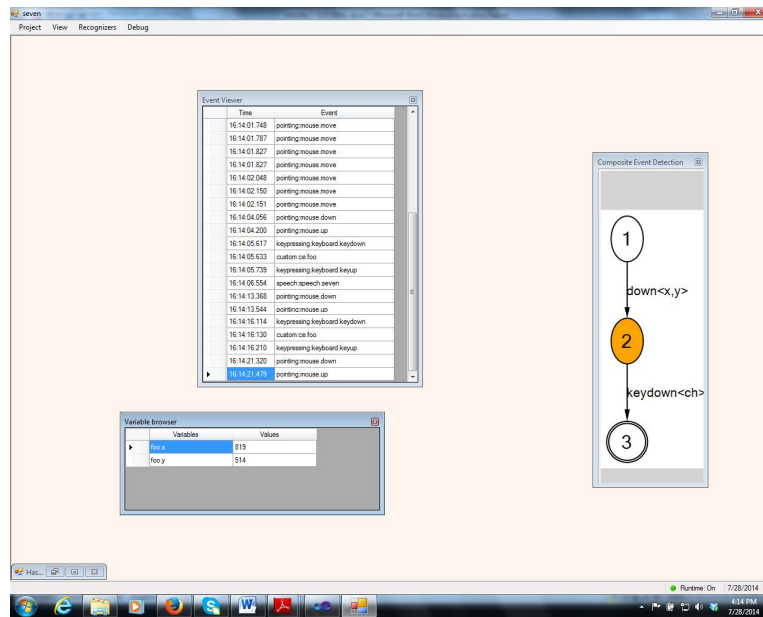


Figure 2. Runtime environment. Debugging tools.

SECOND PART

Creating context-dependent dialogs

The Human-Machine Dialog Definition Language (HMD2L) aims at describing context-dependent human-machine dialogs. To use it, click on the "Context System Visual Editor" of menu "View" (or press Ctrl + Q).

- Load the file `Hasselt_tutorial2.xml` located in the `Desktop\UserStudies\tutorial2` directory.
- Syntax-check the composite events (Ctrl + 1) and the system responses (Ctrl + 2)
- Depict the model seen in Figure 3 in the Hasselt's visual editor.
- Run the program (F5). Notice that the commands `create-object`, `move-that-there`, and `remove-everything` can only be issued in this particular order.

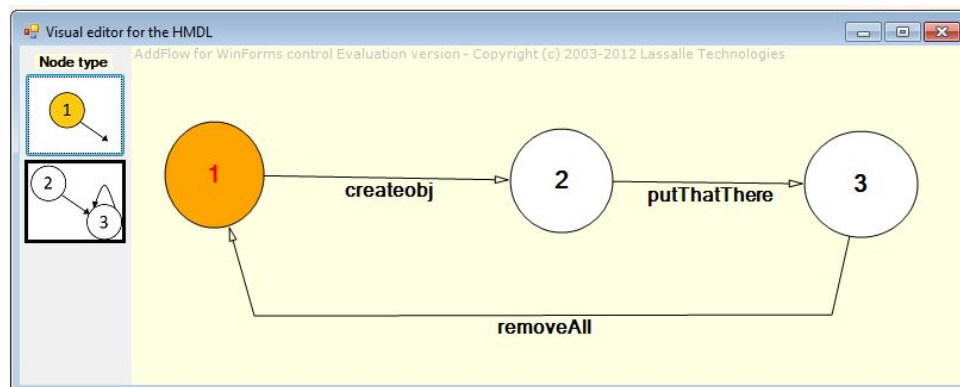


Figure 3. Hasselt visual editor.

[Test the editor. How to draw loops]

D.2 User study tasks

This section shows the instruction sheets given to participants during the user study. Participants were given one instruction sheet at a time.

The content of the instruction sheets is shown in the next pages.

D.3 Statistical tools

Some of the statistical tools used to analyze our data is described below.

D.3.1 Boxplot

In descriptive statistics, a boxplot is a convenient way of graphically depicting groups of numerical data through their quartiles. Boxplots also have lines extending vertically from the boxes (whiskers) indicating variability outside the upper and lower quartiles. Outliers are plotted as individual points. Boxplots are non-parametric: they display variation in samples of a statistical population without making any assumptions of the underlying statistical distribution.

According to the R documentation, by default, R places the plot whiskers at the lowest datum still within $1.5 \times IQR$ of the lower quartile, and the highest datum still within $1.5 \times IQR$ of the upper quartile –where the interquartile range (IQR) is the difference between the upper and lower quartiles¹.

D.3.2 Q-Q normality plots

The Q-Q normality plot is a graphical technique to identify substantive departures from normality. The package R can generate this plots with the command `qqnorm`². If the data are truly sampled from a Gaussian distribution, the points of the Q-Q plot are expected to line up on the line of Identity (i.e. $y = x$). Systematic deviation from this ideal is evidence that the data are not sampled from a Gaussian distribution.

In our comparisons, we want to use paired t-tests to evaluate our hypotheses. However, we could not guarantee the normality of the pair differences (e.g. $Completion\ time_{Hassel\#} - Completion\ time_{C\#}$), which is an assumption to be met before applying t-tests.

¹<http://127.0.0.1:28234/library/graphics/html/boxplot.html>

²<http://127.0.0.1:16862/library/stats/html/qqnorm.html>

EXPERIMENT 1. Evaluating the Hasselt CEDL and the Hasselt SRDL.

You will be presented with a windows form that supports the following commands:

- a) **Creation of objects:** By uttering 'create object', the user can create colored buttons on the form (in a random position).

- b) **Displacement of objects:** The user can utter 'move that there' while pointing on an object and on its intended position.

You will have to make modify the creation:

- The user must choose the location of the objects he/she creates. He/she must utter 'create object here'. The word 'here' must be disambiguated with a mouse click.

Note 1: Do not make any assumption about the order in which the speech input or the mouse click will be detected. The speech input 'here' and the mouse click are considered simultaneous if they are detected within an interval of 1500 milliseconds.

Note 2: Save your source code before entering into runtime mode.

EXPERIMENT 2. Evaluating the Hasselt HMD2L

The system you will have to work with can support the following commands:

a) **Create objects:** The user can create a objects in a given position by uttering

create object + (click)

b) **Remove objects:** The user can remove all the objects from the canvas by saying

remove everything

You will have to make the following modifications to the system:

- Unlike the original version, all the commands will not be active at every time. Users will only be able to issue the command remove-everything if there are objects in the canvas. The canvas is initially empty.

Note 1: Compile the composite events (Ctrl + 1) and system responses (Ctrl + 2) before editing the visual model.

Note 2: Save your source code before entering into runtime mode.

D.3.3 Wilcoxon signed-rank test

The Wilcoxon signed-rank test can be used to compare paired data as non-parametric alternatives to the paired t-test; it is used when you cannot justify a normality assumption for the differences [Elliott 07]. Nonparametric tests are used when the assumptions for a standard parametric (e.g. normality) cannot be reasonably assumed. Nonparametric tests are often less powerful than their parametric cousins and should be used only when the parametric tests is not appropriate [Elliott 07].

The package R performs this nonparametric test through the command *wilcoxsign_test*³.

D.4 Raw data

The data collected from the 12 participants in both experiments is shown below. The two zeroes in the table shown in Figure D.1, a indicate that the participants could not finish the test.

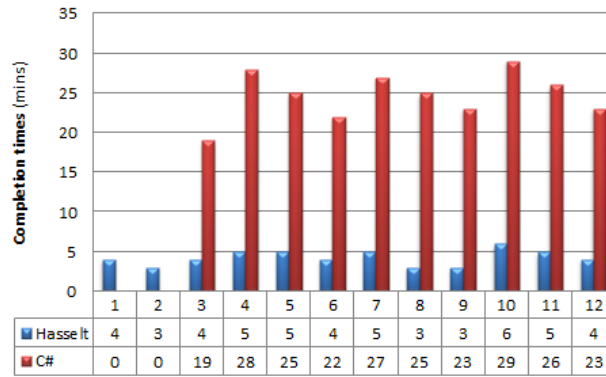
D.5 Other standardized questionnaires

As we mentioned in the thesis, we used the SEQ questionnaire to measure the difficulty of the programming tasks and the SUS questionnaire to evaluate the usability of Hasselt UIMS. These decisions were made after evaluating the following pool of standardized tests.

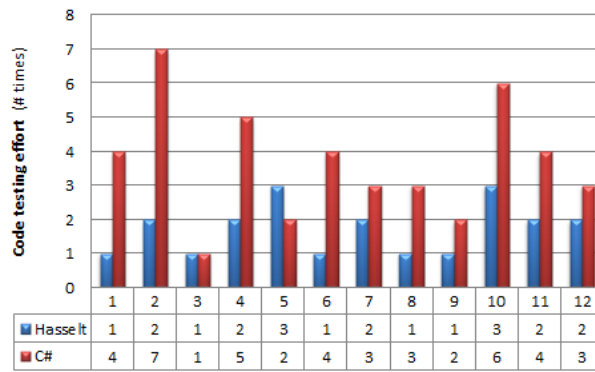
D.5.1 Standardized post-task tests

- The NASA task load index (NASA-TLX) is a multi-dimensional rating procedure that provides an overall workload score based on a weighted average of ratings in six subscales. Each dimension is rated for each task within a 100-points range with 5-point steps [Hart 88].
- The Subjective Mental Effort Questionnaire (SMEQ) is a single item questionnaire in which participants have to put a checkmark on a vertical line that serves as a continuous scale. The scale includes nine labels – from “Not at all hard to do” to “Tremendously hard to do” [Sauro 09].

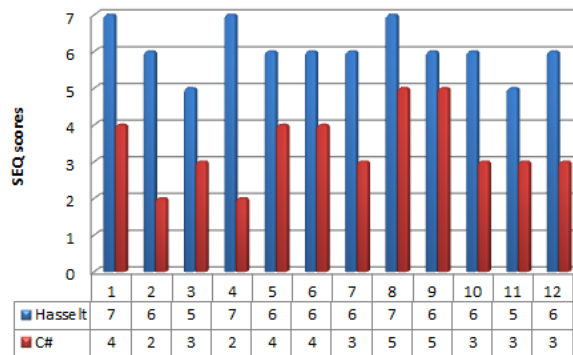
³<http://127.0.0.1:16862/library/coin/html/SymmetryTests.html>



(a) Completion time. Those participants who could not complete a programming task are shown with a completion time of 0 mins.

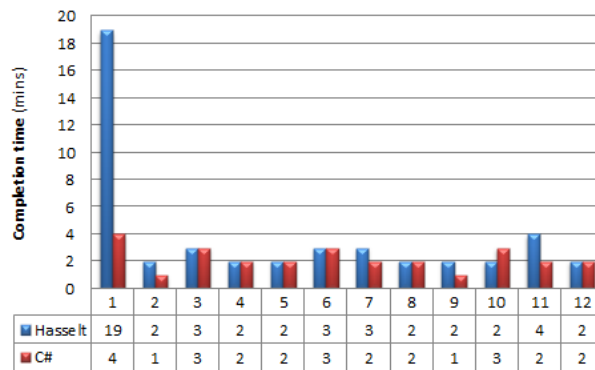


(b) Code testing effort

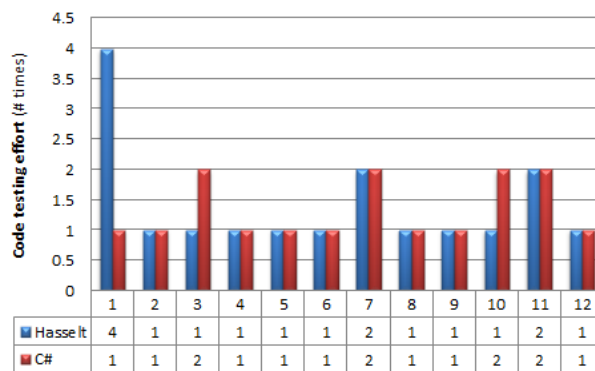


(c) Perceived ease

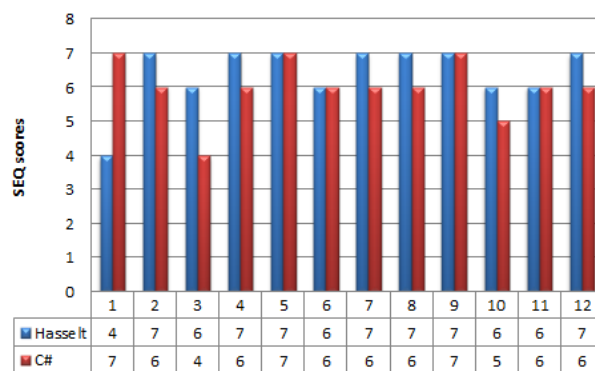
Figure D.1: Raw data collected in the first experiment.



(a) Completion time. Those participants who could not complete a programming task are shown with a completion time of 0 mins.

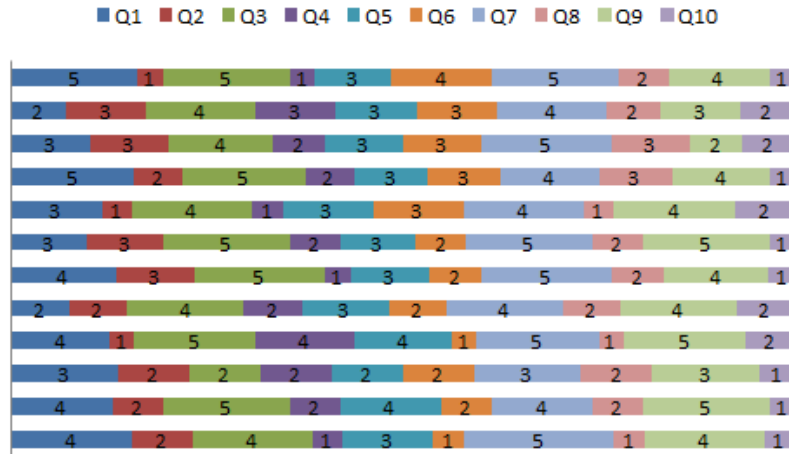


(b) Code testing effort



(c) Perceived ease

Figure D.2: Raw data collected in the second experiment.



(a) SUS scores per question

Figure D.3: Raw data collected from the SUS questionnaires for the 12 participants.

- The Single Ease Question (SEQ) questionnaire, which is a 7-point rating scale (Figure 8.3) aimed to assess the perceived difficulty (or perceived easiness, depending on one's perspective) of a task [Sauro 09].

D.5.2 Standardized usability tests

- The Cognitive Dimensions Questionnaire [Blackwell 00] consisting of five sections, one of which includes 34 open questions enclosed in 14 groups, each group referring to one of the well-known Cognitive Dimensions [Green 89]. The participants are not expected to answer all the questions; rather, they can choose by themselves depending on what aspects of the system they want to criticize.
- The System Usability Scale (SUS) consisting of 10 items with 5 response options (Figure 8.4). To have a benchmark to compare SUS scores, Lewis et al. shared statistical information that summarizes more than 300 usability evaluations. [Lewis 09].

Bibliography

- [Adaikkalavan 06] Raman Adaikkalavan & Sharma Chakravarthy. *SnoopIB: interval-based event specification and detection for active databases*. *Data & Knowledge Engineering Jour.*, vol. 59, no. 1, pages 139–165, 2006.
- [Alexandron 12] Giora Alexandron, Michal Armoni, Michal Gordon & David Harel. *The effect of previous programming experience on the learning of scenario-based programming*. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, pages 151–159. ACM, 2012.
- [Anicic 09] Darko Anicic, Paul Fodor, Roland Stuhmer & Nenad Stojanovic. *Event-driven approach for logic-based complex event processing*. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 1, pages 56–63. IEEE, 2009.
- [Baeten 05] JCM Baeten. *A brief history of process algebra*. *Theoretical Computer Science*, vol. 335, no. 2-3, page 131, 2005.
- [Bansal 13] A.K. Bansal. *Introduction to programming languages*. CRC Press, 2013.
- [Bass 88] Len Bass, Erik Hardy, Kurt Hoyt, M Reed Little Jr & Robert Seacord. *Introduction to the Serpent User Interface Management System*. Rapport technique, DTIC Document, 1988.

- [Beaudouin-Lafon 94] Michel Beaudouin-Lafon. *User interface management systems: Present and future*. In From object modelling to advanced visual communication, pages 197–223. Springer, 1994.
- [Beaudouin-Lafon 03] Michel Beaudouin-Lafon & Wendy E Mackay. *Prototyping tools and techniques*. Human Computer Interaction—Development Process, pages 122–142, 2003.
- [Beaudouin-Lafon 04] Michel Beaudouin-Lafon. *Designing interaction, not interfaces*. In Proceedings of the working conference on Advanced visual interfaces, pages 15–22. ACM, 2004.
- [Ben-Ari 01] Mordechai Ben-Ari. *Constructivism in computer science education*. Journal of Computers in Mathematics and Science Teaching, vol. 20, no. 1, pages 45–73, 2001.
- [Blackwell 00] Alan F Blackwell & Thomas RG Green. *A Cognitive Dimensions questionnaire optimised for users*. In Proceedings of the Twelfth Annual Meeting of the Psychology of Programming Interest Group, pages 137–152, 2000.
- [Bolt 80] Richard Bolt. *Put-that-there: Voice and gesture at the graphics interface*. In Proc. of SIGGRAPH’ 80. ACM, 1980.
- [Bornat 07] Richard Bornat. *Understanding and Writing Compilers—A Do-it-yourself Guide*. 2007.
- [Bourguet 02] Marie-Luce Bourguet. *A toolkit for creating and testing multimodal interface designs*. In Proc. of UIST’02, pages 29–30, 2002.
- [Bourguet 03] Marie-Luce Bourguet. *Designing and Prototyping Multimodal Commands*. In Proceedings of INTERACT’03, pages 717–720, 2003.
- [Brooke 96] John Brooke. *SUS—A quick and dirty usability scale*. Usability evaluation in industry, vol. 189, no. 194, pages 4–7, 1996.

- [Brooks 87] Frederik P Brooks & No Silver Bullet. *Essence and accidents of software engineering*. IEEE computer, vol. 20, no. 4, pages 10–19, 1987.
- [Brown 95] James Dean Brown. The elements of language curriculum: A systematic approach to program development. ERIC, 1995.
- [Cardelli 85] Luca Cardelli & Rob Pike. *Squeak: a language for communicating with mice*. In ACM SIGGRAPH Computer Graphics, volume 19, pages 199–204. ACM, 1985.
- [Chakravarthy 94] Sharma Chakravarthy & Deepak Mishra. *Snoop: An Expressive Event Specification Language for Active Databases*. Data & Knowledge Engineering Jour., vol. 14, no. 1, pages 1–26, 1994.
- [Chen 08] W W L Chen. Discrete mathematics. Macquarie University, 2008.
- [Cirelli 14] Mauricio Cirelli & Ricardo Nakamura. *A Survey on Multi-touch Gesture Recognition and Multi-touch Frameworks*. In Proc. of the Ninth ACM International Conference on Interactive Tabletops and Surfaces, pages 35–44. ACM, 2014.
- [Cohen 97] Philip R Cohen, Michael Johnston, David McGee, Sharon Oviatt, Jay Pittman, Ira Smith, Liang Chen & Josh Clow. *QuickSet: Multimodal interaction for distributed applications*. In Proceedings of the fifth ACM international conference on Multimedia, pages 31–40. ACM, 1997.
- [Coutaz 95] Joëlle Coutaz, Laurence Nigay, Daniel Salber, Ann Blandford, Jon May & Richard M Young. *Four easy pieces for assessing the usability of multimodal interaction: the CARE properties*. In InterAct, volume 95, pages 115–120, 1995.
- [Cuenca 13a] Fredy Cuenca. *The CoGenIVE Concept Revisited: A Toolkit for Prototyping Multimodal Systems*. In Proceedings of the 5th ACM SIGCHI Symposium on En-

- gineering Interactive Computing Systems, EICS '13, pages 159–162, New York, NY, USA, 2013. ACM.
- [Cuenca 13b] Fredy Cuenca, Davy Vanacken, Karin Coninx & Kris Luyten. *Assessing the support provided by a toolkit for rapid prototyping of multimodal systems*. In Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems (EICS'13), pages 307–312. ACM, 2013.
- [Cuenca 14a] Fredy Cuenca, Karin Coninx, Kris Luyten & Davy Vanacken. *Graphical Toolkits for Rapid Prototyping of Multimodal Systems: A Survey*. Interacting with Computers, 2014.
- [Cuenca 14b] Fredy Cuenca, Jan Van der Bergh, Kris Luyten & Karin Coninx. *A Domain-Specific Textual Language for Rapid Prototyping of Multimodal Interactive Systems*. In Proc. of EICS'14. ACM, 2014.
- [Cuenca 15a] Fredy Cuenca, Jan Van den Bergh, Kris Luyten & Karin Coninx. *Empirical Study: Comparing Hasselt with C# to Describe Multimodal Dialogs*. In Proceedings of the First International Workshop on Human Factors in Modeling (HuFaMo 2015), MODELS'15, pages 25–32. CEUR Workshop Proceedings, 2015.
- [Cuenca 15b] Fredy Cuenca, Jan Van den Bergh, Kris Luyten & Karin Coninx. *Hasselt UIMS: A Tool for Describing Multimodal Interactions with Composite Events*. In Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS '15, pages 226–229, New York, NY, USA, 2015. ACM.
- [Cuenca 15c] Fredy Cuenca, Jan Van den Bergh, Kris Luyten & Karin Coninx. *A user study for comparing the programming efficiency of modifying executable multimodal interaction descriptions. A domain-specific language versus equivalent event-callback code*. In To appear in the Proceedings of the sixth workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2015), PLATEAU'15. ACM, 2015.

- [Cugola 12a] Gianpaolo Cugola & Alessandro Margara. *Complex event processing with T-Rex*. Journal of Systems and Software, vol. 85, no. 8, pages 1709–1728, 2012.
- [Cugola 12b] Gianpaolo Cugola & Alessandro Margara. *Processing flows of information: From data stream to complex event processing*. ACM Computing Surveys (CSUR), vol. 44, no. 3, page 15, 2012.
- [Cuppens 04] Erwin Cuppens, Chris Raymaekers & Karin Coninx. *{VRXML}: A User Interface Description Language for Virtual Environments*. 2004.
- [De Boeck 07] Joan De Boeck, Davy Vanacken, Chris Raymaekers & Karin Coninx. *High level modeling of multimodal interaction techniques using NiMMiT*. Journal of Virtual Reality and Broadcasting, vol. 4, no. 2, 2007.
- [De Boeck 08] Joan De Boeck, Chris Raymaekers & Karin Coninx. *A tool supporting model based user interface design in 3d virtual environments*. In Grapp 2008: proceedings of the third international conference on computer graphics theory and applications, pages 367–375, 2008.
- [De Boeck 09] Joan De Boeck, Chris Raymaekers & Karin Coninx. *CoGenIVE: Building 3D Virtual Environments Using a Model Based User Interface Design Approach*. In Computer Vision and Computer Graphics. Theory and Applications, pages 83–96. Springer, 2009.
- [de Ruiter 88] Maurice M de Ruiter. *Advances in computer graphics iii*, volume 3. Springer Science & Business Media, 1988.
- [DeMarco 89] Tom DeMarco & Tim Lister. *Software development: state of the art vs. state of the practice*. In Proceedings of the 11th international conference on Software engineering, pages 271–275. ACM, 1989.
- [Demers 06] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald & Walker White. *Towards expressive publish/subscribe systems*. In Advances in Database Technology-EDBT 2006, pages 627–644. Springer, 2006.

- [Dietz 01] Paul Dietz & Darren Leigh. *DiamondTouch: a multi-user touch technology*. In Proceedings of the 14th annual ACM symposium on User interface software and technology, pages 219–226. ACM, 2001.
- [Dragicevic 04a] Pierre Dragicevic & Jean-Daniel Fekete. *Support for Input Adaptability in the ICON Toolkit*. In Proc. of the ICMI, pages 212–219, New York, NY, USA, 2004. ACM.
- [Dragicevic 04b] Pierre Dragicevic & Jean-Daniel Fekete. *Support for input adaptability in the ICON toolkit*. In Proc. of ICMI'04, pages 212–219. ACM, 2004.
- [Dumas 09] B. Dumas, D. Lalanne & S. Oviatt. *Multimodal Interfaces: A Survey of Principles, Models and Frameworks*. Human Machine Interaction, vol. 5440, pages 3–26, 2009.
- [Dumas 10] Bruno Dumas, Denis Lalanne & Rolf Ingold. *Description Languages for Multimodal Interaction: A Set of Guidelines and its Illustration with SMUIML*. Journal of multimodal user interfaces, vol. 3, no. 3, pages 237–247, 2010.
- [Dumas 14] Bruno Dumas, Beat Signer & Denis Lalanne. *A graphical editor for the SMUIML multimodal user interaction description language*. Science of Computer Programming, vol. 86, pages 30–42, 2014.
- [Elliott 07] Alan C Elliott & Wayne A Woodward. *Statistical analysis quick reference guidebook: With spss examples*. Sage, 2007.
- [Flecchia 87] Mark A Flecchia & R Daniel Bergeron. *Specifying complex dialogs in ALGAE*. In ACM SIGCHI Bulletin, volume 18, pages 229–234. ACM, 1987.
- [Floyd 79] Robert W Floyd. *The paradigms of programming*. Communications of the ACM, vol. 22, no. 8, pages 455–460, 1979.

- [Friedl 15] Susanne Friedl. *Design Solutions for Cross-Device Interaction Issues*. 2015.
- [Gehani 92] Narain H Gehani, Hosagrahar V Jagadish & Oded Shmueli. *Event Specification in an Active Object-Oriented Database*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 81–90, 1992.
- [Gehani 94] Narain Gehani, HV Jagadish & Oded Shmueli. *COMPOSE A System For Composite Event Specification and Detection*. In Book chapter in Advanced Database Concepts and Research Issues. Springer Verlag, 1994.
- [Gibbon 12] Dafydd Gibbon, Inge Mertins & Roger K Moore. Handbook of multimodal and spoken dialogue systems: resources, terminology and product evaluation, volume 565. Springer Science & Business Media, 2012.
- [Gordon 95] V Scott Gordon & James M Bieman. *Rapid prototyping: lessons learned*. IEEE software, vol. 12, no. 1, pages 85–95, 1995.
- [Green 85] Mark Green. *The University of Alberta user interface management system*. ACM SIGGRAPH Computer Graphics, vol. 19, no. 3, pages 205–213, 1985.
- [Green 89] Thomas RG Green. *Cognitive dimensions of notations*. A. Sutcliffe and Macaulay, editors, People and Computers V, pages 443–460, 1989.
- [Green 96] T. Green & M. Petre. *Usability analysis of visual programming environments: a cognitive dimensions framework*. Journal of Visual Languages & Computing, vol. 7, no. 2, pages 131–174, 1996.
- [Grossman 09] Tovi Grossman, George Fitzmaurice & Ramtin Attar. *A survey of software learnability: metrics, methodologies and guidelines*. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 649–658. ACM, 2009.

- [Hamon 13] Arnaud Hamon, Philippe Palanque, José Lu's Silva, Yannick Deleris & Eric Barboni. *Formal description of multi-touch interactions*. In Proceedings of the EICS'14, pages 207–216. ACM, 2013.
- [Harel 85] David Harel & Amir Pnueli. On the development of reactive systems. Springer, 1985.
- [Hart 88] Sandra G Hart & Lowell E Staveland. *Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research*. Advances in psychology, vol. 52, pages 139–183, 1988.
- [Hartson 88] H.R. Hartson & D. Hix. Advances in human-computer interaction. Numéro v. 2 in ADVANCES IN HUMAN-COMPUTER INTERACTION. Ablex Publishing Corporation, 1988.
- [Hill 86] Ralph D. Hill. *Supporting Concurrency, Communication, and Synchronization in Human-computer Interaction—the Sassafras UIMS*. ACM Trans. Graph., vol. 5, no. 3, pages 179–210, July 1986.
- [Hoste 11] Lode Hoste, Bruno Dumas & Beat Signer. *Mudra: a unified multimodal interaction framework*. In Proc. of ICMI'11, pages 97–104. ACM, 2011.
- [Hoste 14] Lode Hoste & Beat Signer. *Criteria, Challenges and Opportunities for Gesture Programming Languages*. Proc. of EGMI, pages 22–29, 2014.
- [Hürsch 95] Walter L Hürsch & Cristina Videira Lopes. *Separation of concerns*. 1995.
- [Jacob 99] R. Jacob, L. Deligiannidis & S. Morrison. *A software model and specification language for non-WIMP user interfaces*. ACM Transactions on Computer-Human Interaction (TOCHI), vol. 6, no. 1, pages 1–46, 1999.
- [Jensen 97] Kurt Jensen. *A brief introduction to coloured petri nets*. In Tools and Algorithms for the Construction and Analysis of Systems, pages 203–208. Springer, 1997.

- [Kasik 82] David J Kasik. *A user interface management system*. In ACM SIGGRAPH Computer Graphics, volume 16, pages 99–106. ACM, 1982.
- [Khandkar 10] Shahedul Huq Khandkar & Frank Maurer. *A domain specific language to define gestures for multi-touch applications*. In Proceedings of the 10th Workshop on Domain-Specific Modeling, page 2. ACM, 2010.
- [Kin 09] Kenrick Kin, Maneesh Agrawala & Tony DeRose. *Determining the benefits of direct-touch, bimanual, and multifinger input on a multitouch workstation*. In Proceedings of Graphics interface 2009, pages 119–124. Canadian Information Processing Society, 2009.
- [Kin 12a] K. Kin, B. Hartmann, T. DeRose & 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.
- [Kin 12b] Kenrick Kin, Björn Hartmann, Tony DeRose & Maneesh Agrawala. *Proton: multitouch gestures as regular expressions*. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI’12), pages 2885–2894, 2012.
- [König 10] Werner König, Roman Rädle & Harald Reiterer. *Interactive design of multimodal user interfaces*. Journal on Multimodal User Interfaces, vol. 3, no. 3, pages 197–213, 2010.
- [Kosar 12] Tomaž Kosar, Marjan Mernik & Jeffrey C Carver. *Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments*. Empirical software engineering, vol. 17, no. 3, pages 276–304, 2012.
- [Lalanne 09] Denis Lalanne, Laurence Nigay, philippe Palanque, Peter Robinson, Jean Vanderdonckt & Jean-François

- Ladry. *Fusion Engines for Multimodal Input: A Survey*. In Proceedings of the 2009 International Conference on Multimodal Interfaces, ICMI-MLMI '09, pages 153–160, New York, NY, USA, 2009. ACM.
- [Lavrakas 08] Paul J Lavrakas. *Encyclopedia of survey research methods*. Sage Publications, 2008.
- [Lawson 09] Jean-Yves Lionel Lawson, Ahmad-Amr Al-Akkad, Jean Vanderdonckt & Benoit Macq. *An open source workbench for prototyping multimodal interactions based on off-the-shelf heterogeneous components*. In Proceedings of the EICS'09, pages 245–254. ACM, 2009.
- [Lewis 09] James R Lewis & Jeff Sauro. *The factor structure of the system usability scale*. In Human Centered Design, pages 94–103. Springer, 2009.
- [Mei 09] Y. Mei & S. Madden. *Zstream: a cost-based query processor for adaptively detecting composite events*. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pages 193–206. ACM, 2009.
- [Murata 89] Tadao Murata. *Petri nets: Properties, analysis and applications*. Proceedings of the IEEE, vol. 77, no. 4, pages 541–580, 1989.
- [Myers 90] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish & Philippe Marchal. *Garnet: Comprehensive support for graphical, highly interactive user interfaces*. Computer, vol. 23, no. 11, pages 71–85, 1990.
- [Myers 92] Brad A Myers & Mary Beth Rosson. *Survey on user interface programming*. In Proceedings of the SIGCHI conference on Human factors in computing systems, pages 195–202. ACM, 1992.
- [Myers 94] Brad Myers. *User Interface Software Tools*. Rapport technique, Pittsburgh, PA, USA, 1994.

- [Myers 98] Brad A Myers. *A brief history of human-computer interaction technology*. interactions, vol. 5, no. 2, pages 44–54, 1998.
- [Myers 00] Brad Myers, Scott E Hudson & Randy Pausch. *Past, present, and future of user interface software tools*. ACM Transactions on Computer-Human Interaction (TOCHI), vol. 7, no. 1, pages 3–28, 2000.
- [Navarre 09] David Navarre, Philippe Palanque, Jean-Francois Ladry & Eric Barboni. *ICOs: A Model-Based User Interface Description Technique dedicated to Interactive Systems Addressing Usability, Reliability and Scalability*. ACM Transactions on Computer-Human Interaction, vol. 16, no. 4, 2009.
- [Neal 89] Jeannette G Neal, CY Thielman, Zuzanna Dobes, Susan M Haller & Stuart C Shapiro. *Natural language with integrated deictic and graphic gestures*. In Proceedings of the workshop on Speech and Natural Language, pages 410–423. Association for Computational Linguistics, 1989.
- [Newman 68] William M Newman. *A system for interactive graphical programming*. In Proceedings of the April 30–May 2, 1968, spring joint computer conference, pages 47–54. ACM, 1968.
- [Obrenovic 04] Zeljko Obrenovic & Dusan Starcevic. *Modeling multi-modal human-computer interaction*. Computer, vol. 37, no. 9, pages 65–72, 2004.
- [Olsen Jr 83] Dan R Olsen Jr & Elizabeth P Dempsey. *SYNGRAPH: A graphical user interface generator*. In ACM SIGGRAPH Computer Graphics, volume 17, pages 43–50. ACM, 1983.
- [Olsen Jr 86] Dan R Olsen Jr. *MIKE: the menu interaction kontrol environment*. ACM Transactions on Graphics (TOG), vol. 5, no. 4, pages 318–344, 1986.

- [Olsen Jr 87] Dan R Olsen Jr. *Larger issues in user interface management*. ACM SIGGRAPH Computer Graphics, vol. 21, no. 2, pages 134–137, 1987.
- [Olsen Jr 89] Dan R Olsen Jr. *A programming language basis for user interface*. In ACM SIGCHI Bulletin, volume 20, pages 171–176. ACM, 1989.
- [Oney 14] Stephen Oney, Brad Myers & Joel Brandt. *InterState: Interaction-Oriented Language Primitives for Expressing GUI Behavior*. In Proc. of UIST'14. ACM, 2014.
- [Oviatt 99] Sharon Oviatt. *Ten myths of multimodal interaction*. Communications of the ACM, vol. 42, no. 11, pages 74–81, 1999.
- [Oviatt 03] Sharon Oviatt. *Multimodal Interfaces*. In The Human Computer Interaction Handbook: Fundamentals, Evolving technologies and Emerging Applications, 2003.
- [Paterno 12] Fabio Paterno. *Model-based design and evaluation of interactive applications*. Springer Science & Business Media, 2012.
- [Paton 99] Norman W Paton & Oscar D´az. *Active database systems*. ACM Computing Surveys (CSUR), vol. 31, no. 1, pages 63–103, 1999.
- [Rowland 15] Claire Rowland, Elizabeth Goodman, Martin Charlier, Ann Light & Alfred Lui. *Designing connected products: Ux for the consumer internet of things*. ” O’Reilly Media, Inc.”, 2015.
- [Samek 03] Miro Samek. *Who Moved My State?* Dr. Dobb’s Journal, 2003.
- [Samek 09] Miro Samek. *A crash course in UML state machines*. Embedded. com, 2009.
- [Sánchez 03] César Sánchez, Sriram Sankaranarayanan, Henny Sipma, Ting Zhang, David Dill & Zohar Manna. *Event*

- correlation: Language and semantics.* In *Embedded Software*, pages 323–339. Springer, 2003.
- [Sauro 05] Jeff Sauro & Erika 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.
- [Sauro 09] Jeff Sauro & Joseph 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.
- [Scharf 13] Florian Scharf, Christian Wolters, Michael Herczeg & Jörg Cassens. *Cross-Device Interaction : Definition, Taxonomy and Application.* In Maarten Weyn, editeur, *AMBIENT 2013 : The Third International Conference on Ambient Computing, Applications, Services and Technologies*, pages 35–41, Porto, Portugal, 2013 2013. IARIA, IARIA.
- [Scholliers 11] Christophe Scholliers, Lode Hoste, Beat Signer & Wolfgang De Meuter. *Midas: a declarative multi-touch interaction framework.* In *Proc. of TEI*, pages 49–56. ACM, 2011.
- [Schöning 10] Johannes Schöning, Jonathan Hook, Tom Bartindale, Dominik Schmidt, Patrick Oliver, Florian Echtler, Nima Motamedi, Peter Brandl & Ulrich von Zadow. *Building interactive multi-touch surfaces.* In *Tabletops-Horizontal Interactive Displays*, pages 27–49. Springer, 2010.
- [Serrano 08] Marcos Serrano, David Juras & Laurence Nigay. *A three-dimensional characterization space of software components for rapidly developing multimodal interfaces.* In *Proc. of ICMIT'08*, pages 149–156. ACM, 2008.
- [Shaer 08] Orit Shaer, Robert JK Jacob, Mark Green & Kris Luyten. *User interface description languages for next*

- generation user interfaces*. In CHI'08 Extended Abstracts, pages 3949–3952. ACM, 2008.
- [Sharma 98] R. Sharma, V.I. Pavlovic & T.S. Huang. *Toward multimodal human-computer interface*. In Proceedings of the IEEE, pages 853–869. IEEE, 1998.
- [Siegmund 15] Janet Siegmund, Norbert Siegmund & Sven Apel. *Views on internal and external validity in empirical software engineering*. In Proceedings of the 37th International Conference on Software Engineering, ICSE 2015,(to appear), 2015.
- [Siek 05] Katie A Siek, Yvonne Rogers & Kay H Connelly. *Fat finger worries: how older and younger users physically interact with PDAs*. In Human-Computer Interaction-INTERACT 2005, pages 267–280. Springer, 2005.
- [Sørensen 04] Carsten Sørensen & David Gibson. *Ubiquitous visions and opaque realities: professionals talking about mobile technologies*. info, vol. 6, no. 3, pages 188–196, 2004.
- [Spano 13a] Lucio Davide Spano. *A Model-Based Approach for Gesture Interfaces*. PhD thesis. 2013.
- [Spano 13b] Lucio Davide Spano, Antonio Cisternino, Fabio Paternò & Gianni Fenu. *GestIT: a declarative and compositional framework for multiplatform gesture definition*. In Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems, pages 187–196. ACM, 2013.
- [Sturm 02] J. Sturm, I. Bakx, B. Cranen, J. Terken & F. Wang. *The Effect of Prolonged Use on Multimodal Interaction*. In Proceedings ISCA Workshop on Multimodal Interaction in Mobile Environments, Kloster Irsee. ACM, 2002.
- [Traum 03] David R Traum & Staffan Larsson. *The information state approach to dialogue management*. In Current and new directions in discourse and dialogue, pages 325–353. Springer, 2003.

- [Turk 14] Matthew Turk. *Multimodal interaction: A review*. Pattern Recognition Letters, vol. 36, pages 189–195, 2014.
- [Vo 96] Minh Tue Vo & Cindy Wood. *Building an application framework for speech and pen input integration in multimodal learning interfaces*. In Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on, volume 6, pages 3545–3548. IEEE, 1996.
- [Wagner 06] Ferdinand Wagner, Ruedi Schmuki, Thomas Wagner & Peter Wolstenholme. Modeling software with finite state machines: a practical approach. CRC Press, 2006.
- [Wahlster 01] Wolfgang Wahlster, Norbert Reithinger & Anselm Blocher. *Smartkom: Towards multimodal dialogues with anthropomorphic interface agents*. In In International Status Conference: Lead Projects HumanComputer-Interaction. Citeseer, 2001.
- [Woods 70] William A Woods. *Transition network grammars for natural language analysis*. Communications of the ACM, vol. 13, no. 10, pages 591–606, 1970.
- [Wu 03] Mike Wu & Ravin Balakrishnan. *Multi-finger and whole hand gestural interaction techniques for multi-user tabletop displays*. In Proceedings of the 16th annual ACM symposium on User interface software and technology, pages 193–202. ACM, 2003.
- [Wu 10] Chaur Wu. Pro dlr in. net 4. Apress, 2010.