A Domain-Specific Textual Language for Rapid Prototyping of Multimodal Interactive Systems

Fredy Cuenca, Jan Van den Bergh, Kris Luyten, Karin Coninx
Hasselt University - tUL - iMinds
Expertise Centre for Digital Media, Diepenbeek, Belgium
{fredy.cuencalucero,jan.vandenbergh,kris.luyten,karin.coninx}@uhasselt.be

ABSTRACT
There are currently toolkits that allow the specification of executable multimodal human-machine interaction models. Some provide domain-specific visual languages with which a broad range of interactions can be modeled but at the expense of bulky diagrams. Others instead, interpret concise specifications written in existing textual languages even though their non-specialized notations prevent the productivity improvement achievable through domain-specific ones.
We propose a domain-specific textual language and its supporting toolkit; they both overcome the shortcomings of the existing approaches while retaining their strengths. The language provides notations and constructs specially tailored to compactly declare the event patterns raised during the execution of multimodal commands. The toolkit detects the occurrence of these patterns and invokes the functionality of a back-end system in response.

Author Keywords
Multimodal systems; composite events; declarative languages.

ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

INTRODUCTION
Multimodal systems can process commands that are conveyed through a wide variety of modalities (e.g. speech, gestures, gaze, etc.) in a coordinated manner. They expand computing to accommodate to a broader spectrum of people and more adverse usage conditions than in the past [19]. However, their implementation comes with a cost: existing commercial frameworks (.NET, Java) cannot separate the concerns of event handling and event detection for multimodal systems as well as they did for traditional WIMP systems.

Whereas traditional systems respond to simple, single action commands (e.g. mouse clicks and keystrokes), multimodal systems must respond to a series of coordinated user actions that are to be regarded as a single command, a multimodal command. For those multimodal systems implemented with commercial frameworks, the execution of multimodal commands will lead to a series of event notifications that a multimodal system must process separately but without ever neglecting their interdependence. The implementation of a mechanism for detecting meaningful event patterns -hereafter called composite events- greatly complicates the creation of multimodal systems.

To ease this problem, the HCI community has proposed toolkits that support the specification and detection of composite events. Some of them provide domain-specific visual languages [3, 7, 6, 17, 8] that enable modeling a wide variety of composite events. However, these specifications easily degenerate into complex diagrams for relatively simple commands such as the *put-that-there* [5]. Other toolkits can interpret textual specifications [1, 15, 24, 13] made in some existing languages, e.g. XML or CLIPS, not specialized for multimodal systems. But the generality of these languages prevents the productivity improvement associated with the use of domain-specific ones [16].

Our research seeks to explore the potential of a distinct type of modeling language -one that combines the distinguishing features of mainstream approaches: the domain-specificity and the textual notation. The former offer substantial gains in expressiveness and ease of use, with corresponding gains in productivity and reduced maintenance costs [16]. The latter leads to concise, easy-to-scan specifications [12].

This paper presents a language that allows defining a broad range of composite events for its subsequent detection by a toolkit. This can acknowledge the partial detection of composite events through multimodal output, and react to their full detection by sending requests to a back-end system. The toolkit is ideal for rapid prototyping: the ease of editing composite events that can be attached with feedback messages allows quick iteration over different interaction techniques while the application-specific code remains unaltered at the back-end side.

Many of our ideas will be illustrated by the *put-that-there* system because (1) it requires integrating both simultaneous and sequential multimodal inputs, and (2) it can serve as a benchmark so that readers can compare different languages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is pe rmitted. To copy otherwise, or republish, to post on servers or to redistribute to lists , requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EICS 2014, June 1720, 2014, Rome, Italy.
Copyright © ACM 978-1-4503-2725-1/14/06...15.00.

Algorithm 1 Oversimplified version of the put-that-there system

```
1: boolean bPut, bThat, bThere, bClick1, bClick2          ▷ Variables capturing external information
2: datetime dPut, dThat, dThere, dClick1, dClick2
3: int x1, y1, x2, y2

4: procedure SPEECHRECOGNIZED(e)                          ▷ Detection of 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 hasPutThatThereOcurred() then putThatThere(x1, y1, x2, y2)

9: procedure MOUSECLICK(e)                                ▷ Detection of 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 hasPutThatThereOcurred() then putThatThere(x1, y1, x2, y2)

13: procedure HASPUTTHATTHEREOCURRED()                    ▷ Detection of the put-that-there event
14:    if bPut & bThat & bThere & bClick1 & bClick2 then
15:        if (tThat − tPut).Milliseconds < 500 & (tThere − tThat).Milliseconds < 500 then
16:            PUTTHATTHERE(x1, y1, x2, y2)

17: procedure PUTTHATTHERE(x1, y1, x2, y2)                ▷ Handler of the put-that-there event
18:    for all o ∈ Controls do
19:        if o.contains(x1, y1) then o.Location = new Point(x2, y2)
```

PUT-THAT-THERE: A MOTIVATING EXAMPLE

In order to highlight the difficulties encountered when developing multimodal systems with commercial frameworks, we will discuss the implementation of the well-known *put-that-there* [2] multimodal command. This allows a user to move a virtual object from its original position to a new one 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.

A simple desktop-version of the aforementioned system requires a Windows form with colored buttons on it (Figure 3a), a library for speech recognition, and the variables and subroutines outlined in Algorithm 1.

Most parts of the pseudo-code shown in Algorithm 1 aim at detecting whether the user has already issued a multimodal command: Boolean variables (line **1**) are used to check the occurrence of relevant events; datetime variables (line **2**), to timestamp the event notifications; and a dedicated subroutine (line **13**), to combine the information carried by these variables in order to verify whether the arrival of speech inputs and mouse clicks matches with the expected order of receipt. Only one subroutine (line **17**) implements the computation required to move the target object; the others are in charge of detecting the occurrence of an specific event pattern.

Variables such as those in lines **1** and **2**, and subroutines such as those in lines **4**, **9** and **13** are not individual peculiarities of the case being studied. Similar members will always be needed to implement two inherent functions of every multimodal system: the recognition and fusion of multimodal natural input [19]. On one hand, the recognition of inputs involves transforming the information captured by input devices into variables (as in procedures SPEECHRECOGNIZED and MOUSECLICK). On the other hand, the fusion of inputs entails combining the information scattered among the aforementioned variables, which requires at least one dedicated subroutine (as procedure HASPUTTHATTHEREOCCURRED).

The proposed language and its supporting toolkit will allow programmers to dispense the majority of the program outlined above. Since the toolkit can detect composite events, like the stream '*put that* (click) *there* (click)', the application-specific functionality encoded in the PUTTHATTHERE method will suffice to create a running system.

WORKFLOW OVERVIEW

The presented toolkit allows defining composite events, attaching them with partial feedback messages, and choosing the methods that will handle these events. Although the handlers have to be coded in a back-end application, they can be referenced from the toolkit.

Figure 1 gives an overview of the steps needed to define a composite event. In step ①, one specifies its constituent events and how these are temporally related with one another, as well as the signature of the subroutine that will be launched upon the detection of the composite event. Once this is done, one can check the composite event expression for correct syntax. This verification causes the generation of a finite state machine (step ②) that will be subsequently used for event tracking, i.e. each well-defined composite event has a reciprocal finite state machine-based representation. Optionally,

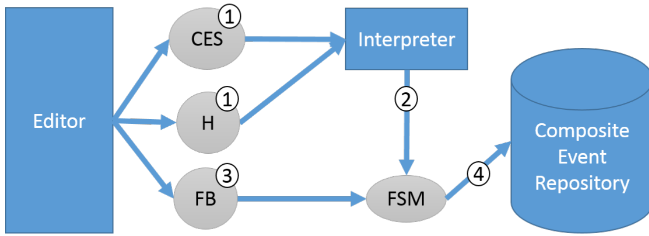


Figure 1: Components involved in the specification of composite events (CES), their handling subroutines (H), and their feedback annotations (FB).

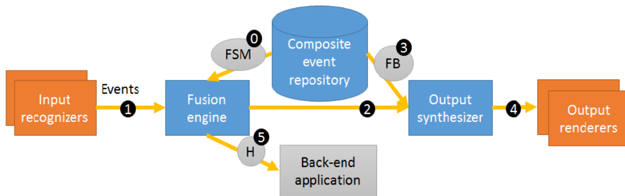


Figure 2: Components involved in the evaluation of composite events through interactive simulation.

one can then annotate the finite state machine with intermediate feedback mechanisms (step ③), after which the composite event is saved (step ④). The toolkit offers the possibility to evaluate the adequacy of a composite event by means of interactive simulation. This evaluation gives one the opportunity to experience the toolkit's recognition speed and rate at runtime, thus guiding modality allocation.

In simulation mode (see Figure 2), the user inputs will be perceived through a set of software recognizers incorporated in the toolkit. In the current implementation, these can sense speech inputs, mouse gestures, mouse inputs, keystrokes, and internally-generated timeout events. The input events are sent to a component called fusion engine (❶). This engine is in charge of detecting the occurrence of composite events, in which case it will activate the functionality of a back-end application (❺). The toolkit can provide intermediate feedback so that its user can be constantly aware of whether it is correctly detecting his/her inputs. In this case, the output synthesizer is notified (❷). Partial feedback is provided by different output renderers (❹), based on the feedback annotations (❸) that were attached to the finite state machine-based representation of the composite event.

COMPOSITE EVENT DEFINITION LANGUAGE

This section begins by explaining the underlying concept behind the composite event definition language we are proposing. It then shows the operators that permit constructing composite events in a declarative manner. Finally, the syntax required to specify event parameters is exposed and illustrated.

Composite Events

Existing commercial frameworks (.NET, Java) can notify about the occurrence of a large, predefined set of simple user events e.g. mouse clicks, keyboard presses, and menu selections. These events are considered to happen in a moment of time. Their instantaneous occurrence earned them the name -in this article at least- of atomic events.

In contrast, composite events occur over a time interval. They are series of related happenings with a unifying meaning, e.g. saying *'zoom here'* while circling on a map. A composite event is composed of several atomic events; it occurs whenever its constituent events occur in a particular order. The definition of a composite event with the proposed language requires declaring the atomic events that comprise it and the relations among them. These relations are specified through predefined event operators.

Event Operators

The operators to be presented were defined after surveying a wide assortment of graphical toolkits for prototyping multimodal systems [4]. These have been widely used to specify multimodal human-machine interaction with executable visual models. We have factored out significant features of these visual languages into a compact textual notation. The resulting operators resemble those successfully used in the field of active databases [21] to specify event-patterns.

Let A and B be two events (atomic or composite). The event operators of the proposed language, in increasing order of precedence, are:

- *FOLLOWED BY* $(;)$: This binary operator is denoted with a semicolon. The utterance $A; B$ indicates the toolkit to notify the occurrence of a composite event whenever the event B occurs after A.

- *OR* $(|)$: This binary operator is denoted with a pipe. The utterance $A|B$ indicates the toolkit to notify the occurrence of a composite event whenever either of A and B occurs.

- *AND* $(+)$: This binary operator is denoted with a plus. The utterance $A + B$ indicates the toolkit to notify the occurrence of a composite event whenever A and B occur simultaneously –read within a short time span. We use (customizable) time thresholds to consider the fact that even when the end user tries to perform simultaneous actions, there may be a short delay between them [18].

- *ITERATION* $(*)$: This unary operator is denoted with an asterisk. The utterance B^* makes the toolkit aware that zero or more occurrences of the event B can be part of a larger composite event.

Parentheses can also be used to alter the pre-established precedence, i.e. to explicitly indicate the terms that have to be evaluated first. For instance, the expressions $A; B|C$ and $(A; B)|C$ have different meanings: the former will be triggered upon the detection of event A followed by either B or C; the latter may be trigger after the consecutive occurrence of A and B, or alternatively upon the detection of C.

Being notified about the ocurrence of a predefined sequence of events may not be enough to determine an appropriate system's response. Specific information about these events may

also be required. Such information is accumulated in the parameters of the composite events.

Composite Event Parameters

The parameters of a composite event are variables that store the information carried by its constituent events.

In the proposed language, composite event parameters must go accompanied by an atomic event and within angular brackets ($\langle\ ,\ \rangle$). Not all atomic events need to have associated parameters, e.g. the detection of the voice command *print* may be enough to produce an appropiate system response.

The semantics of the parameters depend on its associated atomic event and are predefined, e.g. the event key down may be defined with one associated parameter to store the character on the key pressed if necessary. The names of the event parameters must be character strings starting with a letter; their types do not have to be explicitly declared.

To clarify matters, the composite event that will be triggered because of the *put-that-there* command can be specified as follows:

$$\begin{aligned} putThatThere \quad = \quad & speech.put\ ; \\ & speech.that + mouse.click\langle x1, y1\rangle\ ; \\ & speech.there + mouse.click\langle x2, y2\rangle \end{aligned} \tag{1}$$

This utterance indicates the toolkit to store the position of the first and second click in the variables $x1$ and $y1$, and $x2$ and $y2$ respectively. These variables will be set at runtime as the user issues the *put-that-there* multimodal command. They will permit to determine the target object and its intended position respectively.

Continuing with (1), the atomic event $mouse.click$ is predefined in the grammar of the proposed language whereas the speech inputs are dynamically incorporated. Atomic events $speech.put$, $speech.that$, and $speech.there$ are recognized as such, because they belong to the alphabet of the speech recognition grammar that the toolkit reads at startup.

The presence of parameters stems from the fact that their values may be needed by the handling subroutines. For instance, the composite event defined in (1) makes it possible to bind it to the PUTTHATTHERE$(x1, y1, x2, y2)$ method of Algorithm 1. Indeed, the toolkit to be presented in the next section allows establishing this binding.

TOOLKIT USAGE

The proposed toolkit launches one subroutine every time a composite event is detected. The handling subroutines have to be implemented -with no support from the toolkit- in a back-end application. This is a prerequisite to creating multimodal prototypes with our toolkit.

Coding the composite event handlers

The handlers to be invoked upon the detection of a composite event must be coded in a different environment, Microsoft Visual Studio, with a .NET programming language, e.g. C#, Visual Basic.

For instance, to execute a system supporting the *put-that-there* command, both an application containing a windows form with colored buttons on it (Figure 3a), and the PUTTHATTHERE method (line **17** of Algorithm 1) are needed. Since the toolkit performs voice recognition and mouse hooking, no speech recognition libraries or mouse events handlers need to be implemented in the back-end application whose path must be specified in the toolkit's configuration file.

Defining and binding a composite event

Whereas a composite event has to be defined by typing in the toolkit text editor (Figure 3a, CES), its handling subroutine has to be selected from a list (Figure 3b, ③). The text editor offers syntax highlighting, auto completion popups, and function call tips. These features aim at reducing typos, and facilitating the editing and readability of the text. The list of handlers is loaded at startup when the toolkit reads the location of the back-end application from the configuration file. Unlike coding the event handlers, defining composite events requires little programming skills.

Syntax checking a composite event

The syntax of a composite event definition must be verified (Figure 3b, ②) before registering it in the repository (Figure 3b, CER). Each well-defined composite event will be automatically transformed into a semantically equivalent finite state machine (Figure 3b, STM) that the toolkit will use at runtime for event tracking. For ease of exposition, we will not directly refer to the finite state machine but to its graphical form: the state diagram. The nodes of this diagram represent the states the toolkit may be in during event tracking; its arrows indicate the state transitions to be caused upon the detection of atomic events; and its overall topology reflects the temporal constraints among the atomic events.

Attaching feedback to a composite event

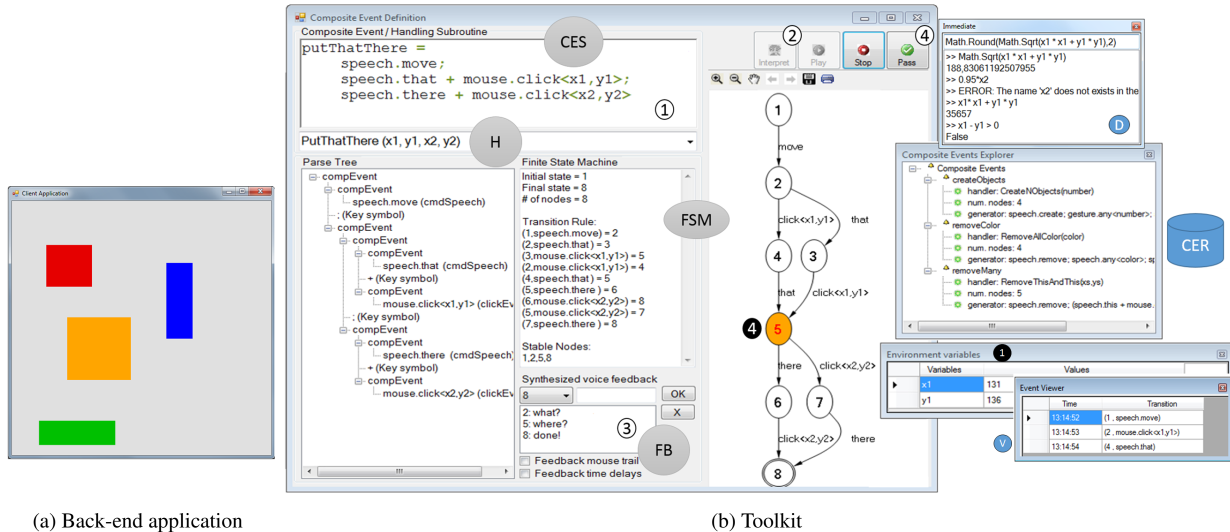
Multimodal commands may involve long series of actions. Thus, it may be desirable for the end user to be progressively notified about whether his/her actions are being correctly detected. This acknowledgment may prevent end users from frustration: no one wants to realize that his/her commands were misinterpreted after few seconds of system inaction.

Toolkit users can attach partial feedback to the nodes of the state diagram representing a composite event. For instance, in Figure 3b, FB, the toolkit is configured to synthesize the utterances *'what?'*, *'where?'* and *'done!'* after the detection of *'put'*, *'that'* (click), and *'there'*(click) respectively.

Transforming user-defined text messages into synthesized voice is not the only way to provide feedback. The toolkit can also be configured to play audio files, to show mouse gesture trails, and/or to display the progression of a timeout event through a progress bar.

Evaluating a composite event

In simulation mode, the toolkit user can convey a myriad of inputs in order to evaluate the adequacy (e.g. to test recognition speed) of a composite event. The progressive recognition of a composite event will be reflected in its reciprocal state diagram-based representation.



(a) Back-end application

(b) Toolkit

Figure 3: The application that handles the composite events (a) and the current version of the toolkit (b) during a simulation of the put-that-there composite event. The toolkit in (b) is annotated with some of the labels from Figure 1 and Figure 2.

Simulations start with the toolkit in the initial state, labeled as 1, meaning that it is ready to sense the external environment. Subsequently, it will change its state as atomic events occur. The new state after a transition is determined by the arrow indicating the name of the triggering event. Eventually, the toolkit will reach its final state, depicted as a double circle, meaning that it has detected the occurrence of the composite event under evaluation. In that moment, a handling subroutine will be launched and the toolkit will go back to the initial state waiting for the next composite event. All toolkit state transitions are graphically reflected in the state diagram, i.e. the current toolkit state is always highlighted (Figure 3b, ❹).

The animated state diagram leads to a quick identification of input recognition problems (when the toolkit becomes stagnant in a particular state). There are also other debugging tools enabling more precise analyses. The variable browser (Figure 3b, ❶) is a window showing the event parameters values; it is updated in each state transition. The interactive debugger (Figure 3b, D) is a scratchpad window in which C# statements involving the event parameters can be evaluated on the fly. The event viewer (Figure 3b, V) shows the happenings detected during the simulation along with their timestamps. Through these tools, for instance, we observed that after 35 executions of the *put-that-there* command, this was detected without fail 68.5% of the times. Mouse clicks, and inputs *'that'* and *'there'* were missed in 2, 6, and 3 occasions respectively. When receiveing partial feedback as in Figure 3b, the recognition rate increased to 94.2%.

Based on the simulation and up to the toolkit user's criteria, the evaluated composite event may be discarded, modified for re-evaluation, or registered into the toolkit. Then, he/she can repeat the process for other composite events. A dedicated window will show all those composite events registered in the toolkit (Figure 3b, CER).

Testing the final prototype

Once the toolkit user has registered all the composite events the intended prototype has to handle, this is ready for end user testing. In this phase, the end user will freely interact with the prototype, i.e. composite events will be triggered in arbitrary order. Unlike simulations, the animated state diagram will not be shown to the end user.

In the final prototype, every multimodal command starts with a pre-defined reset action (like Google Glass [9] commands are activated by first saying *'O.K. Glass'*). In this way, we guarantee that multimodal commands will be issued one at a time. The reset action also allows users to cancel their multimodal command executions at any time. This design decision, however, comes with a cost: the toolkit cannot support the prototyping of multi-user applications since these must handle several commands simultaneously.

EXPLOITING ADVANCED LANGUAGE FEATURES

This section shows how to use the toolkit to exploit language features that were not required by the *put-that-there* example.

Aside from the *put-that-there*, the studied application (Figure 3a) also supports other multimodal commands that allow the creation and deletion of arbitrary sets of objects. The simplicity of this application should not mislead the reader's judgement to underestimate the applicability of the proposed toolkit. Clearly, the same multimodal commands that can be detected during the manipulation of this simple application can also be detected for a more sophisticated one. We expect the reader to focus on the toolkit's functionalities, and distinguish them from the application's functionalities. The implementation of the latter is independent of our tools.

Variable events

End users can remove all the objects of a specific color by uttering a sentence like *'take the green out'*. This required implementing a subroutine $removeAllColor(string\ color)$

-with the obvious functionality- in the back-end application, and using the toolkit to define it as the handler of the event:

$$\begin{aligned} removeColor \;=\; & speech.take; \\ & speech.any\langle color\rangle; \\ & speech.out \end{aligned}$$

The keyword any causes the toolkit to consume an input event that is not accurately declared. In the example, unlike *'take'* and *'out'*, the declaration of the second speech input is rather flexible: it can be any word. Its textual form will be used to set the variable $color$.

The speech recognition grammar includes the words 'take', 'out', and several color names, but not the article 'the', which it is ignored by the toolkit.

Equivalent events

End users can create, from one to nine, objects on the canvas of the application. The number of objects to be created can be specified by means of speech or mouse gesture. This was done by implementing the said functionality into a subroutine, $createNObjects(int\ N)$, and binding it to the composite event:

$$\begin{aligned} createObjects \;=\; & speech.create; \\ & speech.any\langle N\rangle \mid gesture.any\langle N\rangle; \\ & speech.objects \end{aligned}$$

The operator '|'allows end users the possibility to pronounce the number of objects to be created (e.g. by saying *'three'*), or to write it down with a mouse gesture (e.g. by drawing the symbol '3'). In any case the result will be the same: the execution of $createNObjects(N)$ with $N = 3$. Thus, the disjunctive operator allows the definition of robust multimodal commands.

The matching templates of the digits one to nine are stored in xml files in a specific directory. For commands involving mouse gestures, the toolkit can be conveniently configured to show the mouse trail; this is possible through the controls shown in Figure 3b, FB.

Arbitrarily long events

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 technique was implemented by coding the method $removeThisAndThis(int\ xs[\,], int\ ys[\,])$ -with the obvious functionality- and defining it as the handler of the composite event:

$$\begin{aligned} removeMany \;=\; & speech.remove; \\ & speech.this + mouse.click\langle x[\,], y[\,]\rangle; \\ & (speech.and; \\ & speech.this + mouse.click\langle x[\,], y[\,]\rangle)^*; \\ & speech.now \end{aligned}$$

The toolkit 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.

Timeout events

The toolkit can be configured to detect single, double, and triple clicks that can lead to a different computation in the back-end application. To this end, the following composite event must be defined:

$$\begin{aligned} manyClicks \;=\; & mouse.click\langle xs[\,], ys[\,]\rangle; \\ & (mouse.click\langle xs[\,], ys[\,]\rangle; \\ & \;mouse.click\langle xs[\,], ys[\,]\rangle \mid delay\text{-}250 \\ & \;) \mid delay\text{-}250 \end{aligned}$$

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.

A subroutine $nClicks(int\ xs[\,], int\ ys[\,])$ implements different responses to the single, double, and triple click detection. The number of clicks is disclosed from the size of the arrays $xs[\,]$ and $ys[\,]$ passed as parameters.

TOOLKIT IMPLEMENTATION

This section will describe the main libraries and algorithms used to implement the proposed toolkit. This was developed in C# the same as the back-end application described in the previous sections.

Third-party software components

The toolkit text editor is the control ScintillaNET [22]. Its API makes it simple to benefit from advanced text editing and syntax highlighting. Both the grammar specification of our language and the parsing of its utterances were implemented by invoking the Irony library [14]. The state diagrams used during the simulations are controlled through MSAGL [10]. Reflection libraries are used to inspect the methods of the back-end application and to invoke them upon the detection of composite events.

Voice recognition is implemented through the System.Speech namespaces in the Microsoft .NET Framework; mouse gestures are identified by the 1$ recognizer [25]; and mouse actions and keystrokes are intercepted through hook procedures. As to the synthesizers, speech generation is implemented by the System.Speech namespaces; and the visibility of the mouse trails is controlled through the Windows API.

Verifying the validity of composite events

The grammar defining the proposed composite event definition language can be seen in (2).

The nonterminal symbols, $\langle compEvt\rangle$ and $\langle atomEvt\rangle$, refer to composite and atomic events respectively; $\langle pName\rangle$ defines the syntax of the event parameter names and is expressed as a regular expression. As to the terminal symbols, some of them, e.g. $mouse.click$, are predefined by the toolkit; others, e.g. $speech.put$, are incorporated at startup while the toolkit reads the speech recognition grammar file.

$$
\begin{array}{lcl}
\langle compEvt \rangle & \rightarrow & \langle atomEvt \rangle \\
\langle compEvt \rangle & \rightarrow & \langle compEvt \rangle \; * \\
\langle compEvt \rangle & \rightarrow & \langle compEvt \rangle \; ; \; \langle compEvt \rangle \\
\langle compEvt \rangle & \rightarrow & \langle compEvt \rangle \; | \; \langle compEvt \rangle \\
\langle compEvt \rangle & \rightarrow & \langle compEvt \rangle \; + \; \langle compEvt \rangle \\
\langle atomEvt \rangle & \rightarrow & \; mouse.click \\
 & & | \; mouse.click\langle pName, pName \rangle \\
 & & | \; mouse.move\langle pName, pName \rangle \\
 & & | \; speech.any\langle pName \rangle \\
 & & | \; speech.put \\
 & & | \quad \ldots \\
 & & | \; (\; \langle compEvt \rangle \;) \\
\langle pName \rangle & \rightarrow & [a\text{-}zA\text{-}Z][a\text{-}zA\text{-}Z0\text{-}9]^*
\end{array}
\qquad (2)
$$

The correct syntax of each string specifying a composite event is verified against this grammar. Such syntactic analysis may have two results: the string does not belong to the language in which case an error message is thrown, or it is well formed in which case a parse tree is returned.

Transforming composite events into finite state machines

The parse tree of a well-defined composite event will be transformed into an state diagram by the function $createStateDiagram$ (Algorithm 2). This transforms the tree whose root node is passed as an argument into an state diagram, which is then returned as output. Hence, the state diagram representing a composite event E can be obtained by invoking $createStateDiagram(rt)$ -where rt is the root node of the tree obtained from parsing E.

The base case of the recursive Algorithm 2 occurs when its argument is a node with a single child. Such nodes represent atomic events and have trivial transformations, e.g. the smallest graph of Figure 4a represents the atomic event $e3$. The recursive cases involve intermixing small state diagrams obtained from transforming fragments of the parse tree. Each operator defines a different way to intermix state diagrams.

When two composite events are linked by a 'FOLLOWED BY' operator, the toolkit concatenates its reciprocal state diagrams (Figure 4b). When two composite events are connected by the disjunctive operator 'OR', the toolkit creates a new state diagram by overlaying the initial and final states of its graphical counterparts (Figure 4c). Two composite events connected by the conjunctive operator 'AND' causes the creation of a state diagram whose paths between its initial and final nodes are the permutations of all the events contained in their reciprocal state diagrams (Figure 4d). Finally, a single composite event followed by the 'ITERATION' operator causes the alteration of its parallel state diagram: the ingoing arcs of its final state will be redirected to its initial state (Figure 4e).

Events consumption policy

When entering simulation mode, the input recognizers are activated so that the toolkit can embark on event tracking. Every time an event occurs, the toolkit checks whether it was expected, i.e. whether its name is annotated in some outgoing arc of the node representing the current toolkit state. In the affirmative case, the toolkit moves to another state and sets the event parameters values associated with this transition. Otherwise, the toolkit state remains the same. In both cases, the triggering event will be consumed and no longer available for processing.

End user testing adds a layer of complexity. Here, the toolkit must handle several composite events occurring in arbitrary order. However, since our approach requires multimodal commands to be executed one at a time, identifying the multimodal command under execution will reduce the problem to the simulation case. This identification is achieved from the first event detected after the reset action.

Handling parallel inputs

When a composite event is transformed into its reciprocal state diagram, its nodes are automatically classified into: stable/unstable. The toolkit uses these unstable nodes to handle parallel input. Visits to unstable nodes cannot last longer than a threshold time set in the toolkit's configuration file.

For instance, during the detection of the *put-that-there* command, the toolkit entrance to states 3 or 4 (Figure 3b) will cause the activation of a timer ensuring fast transitions to state 5. If the timer expires the toolkit will go back to state 2, the last visited stable state. This prevents long delays between speech inputs and mouse clicks, i.e. end users are enforced to issue them simultaneously.

Unstable states appear when events are connected by the 'AND' operator. All the intermediate nodes of the graph returned by PERMUTE($sd1$, $sd2$) will be classified as unstable states, e.g. all nodes except for 1 and B in Figure 4d. When the toolkit steps back from an unstable to the last visited stable state, the variables set during this interim are rolled back.

RELATED WORK

There are currently toolkits that allow specifying the composite events characterizing multimodal human-machine interaction. These can be classified into two groups:

Toolkits providing domain-specific visual languages

MEngine [3] offers a graphical editor that allows users to depict composite events as state diagrams. It can combine gestures and speech inputs but its models grow too quickly when dealing with simultaneity. To correctly model simultaneous events, many possibilities must be considered, e.g. deictic terms can precede pointing or vice versa during speak-and-point selection. Our toolkit protects its users from this state explosion through the automatic generation of state diagrams.

NiMMiT [6] was a visual language used to specify multimodal interactions in the context of virtual environments. Its notation allows grouping sets of simultaneous events but not successions of related events. Another NiMMiT issue is that the parameter values of a triggering event, e.g. mouse cursor position, have to be captured by the back-end system. Thus, sequentally multimodal interactions lead to bulky diagrams. In our language, large series of sequential events can be regarded as a single composite event through the FOLLOWED BY operator. Moreover, the parameter values are captured and stored by our toolkit until the back-end system is invoked.

ICO [17] allows formal descriptions of multimodal interactive systems. It has been successfully applied in the field

Algorithm 2 Transforms (a fragment of) a parse tree into a state diagram

procedure CREATESTATEDIAGRAM($node$) ▷ $node$ is the root of the (sub)tree to be transformed

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

else if $isComposite(node.children[1])$ & $node.children[2] =$ '$*$' **then**
  $sd1 \leftarrow createStateDiagram(node.children[1])$
  return LOOP(sd1)

else if $isComposite(node.children[1])$ & $node.children[2] =$ ';' & $isComposite(node.children[3])$ **then**
  $sd1 \leftarrow createStateDiagram(node.children[1])$
  $sd2 \leftarrow createStateDiagram(node.children[3])$
  return CONCATENATE(sd1, sd2)

else if $isComposite(node.children[1])$ & $node.children[2] =$ '|' & $isComposite(node.children[3])$ **then**
  $sd1 \leftarrow createStateDiagram(node.children[1])$
  $sd2 \leftarrow createStateDiagram(node.children[3])$
  return OVERLAY(sd1, sd2)

else if $isComposite(node.children[1])$ & $node.children[2] =$ '$+$' & $isComposite(node.children[3])$ **then**
  $sd1 \leftarrow createStateDiagram(node.children[1])$
  $sd2 \leftarrow createStateDiagram(node.children[3])$
  return PERMUTE(sd1, sd2)

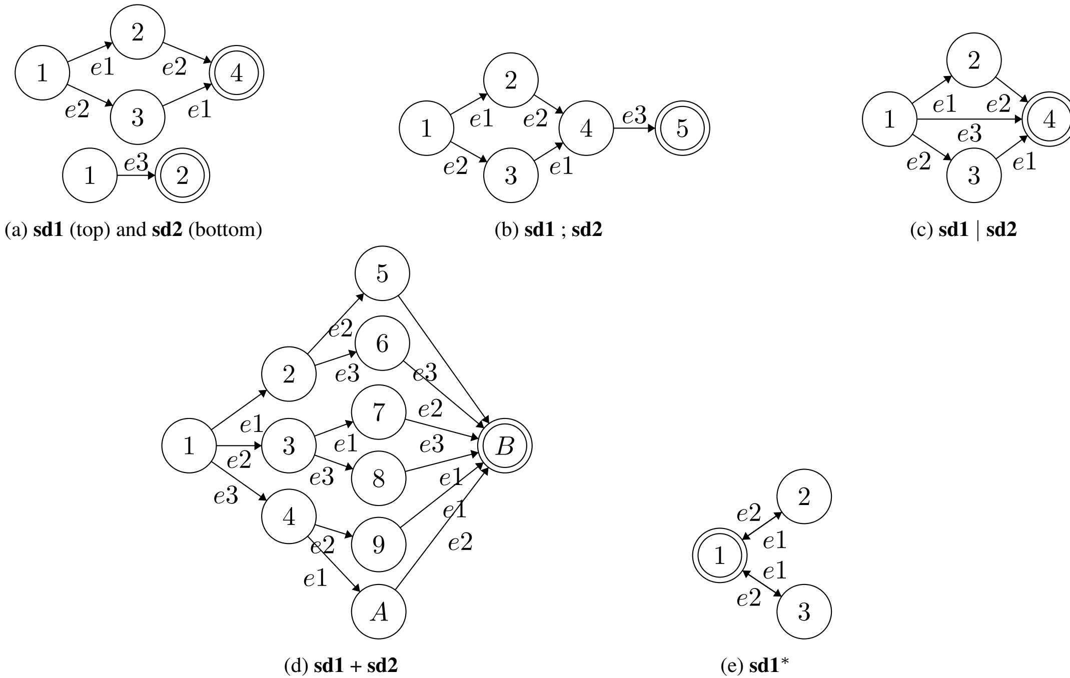


Figure 4: The effect of the operators as applied on two state machines specified in (a): (b) **CONCATENATE(sd1, sd2)**. (c) **OVERLAY(sd1, sd2)**. (d) **PERMUTE(sd1, sd2)** and (e) **LOOP(sd1)**.

of safety-critical systems. Both simultaneous and sequential user actions can be specified through the depiction of Petri nets-based models. But the correct understanding of these models is not a simple task: it requires a solid command of the transition rule of Petri nets. In contrast, our notation maps directly to the problem domain without need of specialized knowledge beyond multimodal interaction.

The visual models of HephaisTK [8] include symbols to group simultaneous and successive events. However, it can only group fixed-length sequences of events. Besides that, event parameters have to be specified in a separate file. In our language instead, an arbitrarily long sequence of events can be regarded as one composite event through the ITERATION operator. Our specifications also include event parameters.

Additional graphical toolkits were described in [4].

Toolkits exploiting ready-made textual languages

Mudra [13] interprets CLIPS-based specifications. Composite events are defined by declaring a set of rules that will be verified against a fact base. Facts are inserted as user events are detected. The satisfaction of a rule indicates the occurrence of a composite event. Unlike our toolkit, Mudra offers multi-user support.

MIML [1], XISL [15], and UsiXML [24] are XML-based languages aimed at describing multimodal interfaces. MIML describes interfaces in three layers: interaction, tasks and platform. XISL allows describing the user operations and system actions separated from XML contents. It offers constructs to describe sequential, parallel, alternative, and coordinated use of modalities. UsiXML can shape the user interface of any interactive application including multimodal ones. With UsiXML, the user interfaces can be described in a way that preserves the design independently from peculiar characteristics of physical computing platform. The proponents of these languages also offer toolkits that transform their specifications into concrete interfaces.

For each of these toolkits, its users must pay for the verboseness of its underlying language. Both CLIPS and XML programs contain a high number of pair-delimiter symbols. The commands, facts, and arguments, which are essential elements in CLIPS programs, must be enclosed between parentheses. This leads to expressions with excessive pairings as the comparison between our specification, shown in (1), and the one in [13, p. 5] attests. Likewise, the editing of XML documents is riddled with tag pairs. The use of pair-delimiter symbols is not only tedious, but also a potent source of slips since it is not uncommon for the pairing to go wrong [11].

DISCUSSION

As mentioned above, previous research has focused on two classes of executable multimodal human-machine interaction models. In this paper, we elaborate on a new approach to multimodal interaction modeling through the use of a domain-specific textual language. We decided to implement an specialized language because domain-specificity can lead to substantial gains in expressiveness and ease of use [16]. With the textual notation, we looked for compact specifications.

Our language certainly leads to compact specifications as the comparison of our *put-that-there* definition (1) against the six models shown in [5, p. 5], [13, p. 5], [3, p. 2], and [20, p .4] suggests. More precisely, the physical space occupied by the enclosing boxes of the aforementioned models ranges from 22.8 (HephaisTK) to $60.8\ cm^2$ (NiMMiT) whereas our specification only requires $9.0\ cm^2$. If we consider that the visual specification of HephaisTK is incomplete (parameters are set in another file), the second briefer specification would be the one of Mudra ($25.4\ cm^2$). In this comparison, all models were printed to have similar readability. Although these results cannot be generalized, they serve as an indicator of the conciseness of our language. The benefits of a concise notation should not be underestimated: 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 [11].

The expected gains in expressiveness were also observed. When counting the mininal number of subroutines required to implement the examples previously described, we found that this can be reduced by a range of 33% to 60% when using the toolkit. For instance, without using the toolkit, we needed five procedures to create a standalone C# *put-that-there* application: the four shown in Algorithm 1 plus the $InitializeComponent$ method required when building windows forms. When using the toolkit, we only needed a back-end C# system with two methods: PUTTHATTHERE and $InitializeComponent$.

It must be emphasized that our approach does not rely on the low frequency at which the speech inputs or mouse clicks, used in the examples, are normally generated. The toolkit can also handle high throughput of events. For instance, mouse movements, which may be raised around 100 times per second, can be processed by using the $mouse.move\langle x[\,], y[\,]\rangle^*$ pattern in unimodal commands such as the *drag and drop*. Then, extending the toolkit so that it can support other devices generating high throughput data streams, e.g. laser pointers or accelerometers, would not challenge our approach.

FUTURE WORK

We will investigate the usability of our toolkit via user studies. This includes measuring the time and programming workload required to define multimodal interfaces in both our toolkit and a commercial framework.

A manageable issue that we will address in future work is that the toolkit only invokes the external back-end application upon the full detection of a composite event. This can be a problem when specifying interactions that require application-dependent intermediate feedback, e.g. when the selected object in the *put-that-there* has to be highlighted. To redress this problem, we plan to modify the system structure so that the handling subroutines can be bound not to the state diagrams but to their nodes.

At this point, our toolkit can only handle one multimodal command execution at a time, which precludes multi-user applications. We will evaluate the feasibility of using UML state machines [23, Chapter 15]. In theory, treating com-

posite events as the orthogonal regions of a UML state machine would permit modeling simultaneous execution of multimodal commands.

CONCLUSION

The detection of multimodal commands requires a supervisory mechanism for detecting event patterns. In order to free programmers from implementing such mechanism, the HCI community has proposed many toolkits seeking to ease this problem. Some toolkits provide domain-specific visual languages in which the composite events to be detected must be specified. Other toolkits interpret specifications performed in a ready-made textual language like XML or CLIPS.

We have proposed a composite event definition language that combines the distinguishing features of these dominant approaches: the domain-specificity and the textual notation. The former offers gains in expressiveness and ease of use; the latter leads to compact specifications. The language allows defining composite events for their subsequent detection by a toolkit. This transforms user-defined composite events into finite state machines enhanced with parameterized events and unstable nodes. The partial detection of a composite event can be acknowledged through multimodal output; its full detection causes the invocation of handles in a back-end system.

REFERENCES

1. Araki, M., and Tachibana, K. Multimodal dialog description language for rapid system development. In *Proceedings of the 7th SIGdial Workshop on Discourse and Dialogue* (2006), 109–116.
2. Bolt, R. Put-that-there: Voice and gesture at the graphics interface. In *Proceedings of the 7th annual conference on computer graphics and interactive techniques (SIGGRAPH' 80)*, ACM (1980).
3. Bourguet, M. A toolkit for creating and testing multimodal interface designs. In *Proceedings of UIST'02* (2002), 29–30.
4. Cuenca, F., Coninx, K., Luyten, K., and Vanacken, D. Graphical Toolkits for Rapid Prototyping of Multimodal Systems: A Survey. *Interacting with Computers* (2014).
5. Cuenca, F., Vanacken, D., Coninx, K., and Luyten, K. 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)*, ACM (2013), 307–312.
6. De Boeck, J., Vanacken, D., Raymaekers, C., and Coninx, K. High level modeling of multimodal interaction techniques using NiMMiT. *Journal of Virtual Reality and Broadcasting 4*, 2 (2007).
7. Dragicevic, P., and Fekete, J. Support for input adaptability in the icon toolkit. In *Proceedings of the 6th International Conference on Multimodal Interfaces (ICMI'04)*, ACM (2004), 212–219.
8. Dumas, B., Lalanne, D., and Ingold, R. Description Languages for Multimodal Interaction: A Set of Guidelines and its Illustration with SMUIML. *Journal of multimodal user interfaces 3*, 3 (2010), 237–247.
9. Google Glass. `http://www.google.com/glass/start/`.
10. Microsoft Glee. `http://research.microsoft.com/en-us/projects/msagl/`.
11. Green, T., and Petre, M. Usability analysis of visual programming environments: a cognitive dimensions framework. *Journal of Visual Languages & Computing 7*, 2 (1996), 131–174.
12. Green, T. R., and Petre, M. When visual programs are harder to read than textual programs. In *In Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics)*, ACM (1992).
13. Hoste, L., Dumas, B., and Signer, B. Mudra: a unified multimodal interaction framework. In *Proceedings of the 13th international conference on multimodal interfaces (ICMI'11)*, ACM (2011), 97–104.
14. Irony. `http://irony.codeplex.com/`.
15. Katsurada, K., Nakamura, Y., Yamada, H., and Nitta, T. Xisl: A language for describing multimodal interaction scenarios. In *Proceedings of the 5th International Conference on Multimodal Interfaces (ICMI 2003)* (2003), 281–284.
16. Mernik, M., Heering, J., and Sloane, A. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR) 37*, 4 (2005), 316–344.
17. Navarre, D., Palanque, P., Ladry, J., and Barboni, E. ICOs: A Model-Based User Interface Description Technique dedicated to Interactive Systems Addressing Usability, Reliability and Scalability. *ACM Transactions on Computer-Human Interaction 16*, 4 (2009).
18. Oviatt, S. Ten myths of multimodal interaction. *Communications of the ACM 42*, 11 (1999), 74–81.
19. Oviatt, S. Multimodal interfaces. In *The Human Computer Interaction Handbook: Fundamentals, Evolving technologies and Emerging Applications* (2003).
20. Palanque, P., and Schyn, A. A model-based approach for engineering multimodal interactive systems. In *INTERACT* (2003).
21. Paton, N., and Daz, O. Active database systems. *ACM Computing Surveys 31*, 1 (1999), 63–103.
22. ScintillaNET. `http://scintillanet.codeplex.com/`.
23. UML State Machines. `http://www.omg.org/spec/UML/2.4.1/Superstructure/`.
24. UsiXML. `http://www.usixml.org/`.
25. Wobbrock, J., Wilson, A., and Li, Y. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *Proceedings of the 20th annual ACM symposium on User interface software and technology (UIST'07)*, ACM (2007), 159–168.