Interacting with Computers Advance Access published March 30, 2014

doi:10.1093/iwc/iwu003

Graphical Toolkits for Rapid Prototyping of Multimodal Systems: A Survey

FREDY CUENCA*, KARIN CONINX, DAVY VANACKEN AND KRIS LUYTEN

Hasselt University – tUL – iMinds, Expertise Centre for Digital Media, Diepenbeek, Belgium
**Corresponding author: fredy.cuencalucero@uhasselt.be*

The creation of prototypes and their iterative adaptation are important stages in the time-consuming development process of a multimodal system. This reality has brought about the appearance of many specialized toolkits intended for facilitating the prototyping of multimodal systems. Using these toolkits, some functionalities of the intended prototype can be specified by means of a visual language instead of programming code. This article reports on a comparative study of a representative set of graphical toolkits for rapid prototyping of multimodal systems. It presents a set of criteria to identify the scope of a toolkit. According to their scopes, toolkits can be clustered into three classes called flow-based, state-based and token-based. Toolkits within each class do not only offer similar benefits to their users, but also exhibit resemblance in their underlying visual languages.

RESEARCH HIGHLIGHTS

- We define an indicator, herein called scope, which enables us to gauge the support provided by a toolkit during the creation of a multimodal prototype.
- The scope of a toolkit is defined as a nominal variable so that it can yield unambiguous measurements and allow comparisons.
- We compare several toolkits as a function of their scopes.
- We identify three distinct classes of toolkits: flow-based, state-based and token-based and show that visual models created with the toolkits within each class are similar. They resemble block diagrams, state diagrams or Petri net graphs.

Keywords: user Interface toolkits; interface design prototyping; systems and tools for interaction design

Editorial Board Member: Jean Vanderdonckt

Received 11 March 2013; Revised 15 October 2013; Accepted 13 January 2014

1. INTRODUCTION

A multimodal system is a computer system capable of collecting the information provided by a user through multiple input modes, integrating these inputs in order to interpret the user's intent and responding to him/her via multiple outputs. Examples of input modes that can be used to enter information into a multimodal system are speech, touch, hand gestures, handwriting or sketching. Some output modes used to respond to the user may include images, audio, synthesized voice, video or haptics.

Multimodal systems can expand computing to accommodate to a broader spectrum of people and more adverse usage conditions than in the past (Oviatt, 2003). They can combine complementary information conveyed through different input modes. This capacity can be exploited to perform disambiguation (Oviatt, 1999), and thus to reduce the misinterpretations that usually arise because of the use of error-prone input modes. For instance, speech recognition can be improved when supported by lip movements recognition (Gibbon *et al.*, 2000). Furthermore, multimodal systems have the potential to provide equivalent input modes for issuing a command so that the users can choose the most convenient mode to utilize according to the context. For example, in noisy environments, a person may prefer to issue touch commands rather than voice commands to his/her multimodal mobile phone. Such flexibility enables human–machine interaction under unfavorable circumstances.

The development of multimodal systems is time-consuming (De Boeck *et al.*, 2009; Dumas *et al.*, 2013; Flippo *et al.*, 2003) and therefore expensive (De Boeck *et al.*, 2007). This is partly because of the complexity of multimodal interaction (Ait-Ameur and Kamel, 2004; Dargie *et al.*, 2007), the absence of standardized methodology (Dargie *et al.*, 2007) and the mastering of different state of the art technologies required for their construction (Dumas *et al.*, 2013). One important phase of the development of multimodal applications is prototyping (Lawson *et al.*, 2009). It involves the creation and iterative adaptation of prototypes (Vanacken *et al.*, 2006), i.e. incomplete versions of the system being developed. Therefore, the development phase of a multimodal system can be shortened by facilitating the creation and modification of prototypes, which is precisely the purpose of the toolkits under study.

A graphical toolkit for rapid prototyping of multimodal systems includes a framework and a graphical editor. It aims to enhance an external application, herein called client application, with multimodal capabilities. On one hand, the client application has to be developed by means of a textual programming language and with no support from the toolkit. On the other hand, the graphical editor allows the depiction of visual models that will be interpreted and executed by the framework. The visual models specify the tasks the prototype must perform during its interaction with the end-user. Some of these tasks are present in a wide variety of multimodal systems (e.g. speech recognition, tracking of the system's state, simultaneous activation of rendering devices) and can be performed by the framework. Other tasks are application-specific and have to be carried out by the subroutines of the client application.

For illustrative purposes, consider a multimodal system whose users are allowed to utter a voice command 'zoom here' while touching a specific point on the screen to indicate the region to zoom in (left side of Fig. 1). Prototyping such a system with the support of a toolkit entails the implementation of the graphical user interface (GUI) the end user will interact with and the subroutine(s) required to scale up a specific region of this GUI. Both the particular behavior of the GUI and the specific scaling algorithms must be implemented as part of a client application. This client application does not need to detect voice commands or touchscreen events. Neither does it have to verify the temporal co-occurrence of the speech input 'here' and the touch on the screen, required to zoom in a region of the GUI. Both functionalities can be delegated to the framework through a visual model like the one shown in the right-hand side of Fig. 1. This model specifies that the detection of the speech input 'zoom' followed by the simultaneous detection of the speech input 'here' and a touch on the screen will cause the execution of the subroutine *ZoomAt*, implemented in the client application.

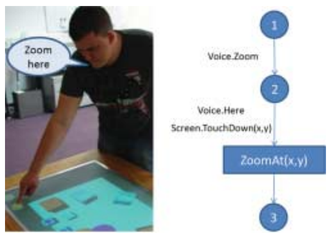


FIGURE 1. *Left:* End-user interacting with a multimodal system. *Right:* Visual model used for specifying human–machine interaction.

By using both the client application and the visual models as inputs, the framework of the toolkit can load a working system that is used to explore, clarify and refine user requirements. After this prototype has served its purpose of elaborating the user's requirements, one typically progresses toward the implementation of the final system. Aspects like efficiency, portability, error handling, ambiguity resolution and contradiction detection have to be considered for the final implementation. The functionality described by the visual models has to be implemented too, since the toolkits cannot convert it into programming code. This inability to translate visual models into programming code confines the studied toolkits to the role of executors able to launch *rapid prototypes* (Beaudouin and Mackay, 2003).

The visual model shown in Fig. 1 was not created with any of the toolkits presented in this work. However, as with the models supported by many of the studied toolkits, it resembles a state diagram (Wagner *et al.*, 2006). For other toolkits, their editors allow the creation of visual models that resemble block diagrams (Nise, 2011) or Petri net graphs (Jensen *et al.*, 2007; Murata, 1989).

The differences among the existing graphical toolkits are not restricted to their underlying visual languages. These toolkits also provide different features, target different domains, use different programming paradigms and/or expect different skills from their users. This overwhelming diversity obstructs the comparative study of these toolkits. It is difficult to find criteria that are applicable to every graphical toolkit.

The present work provides a comparative study of graphical toolkits for rapid prototyping of multimodal systems. The variable chosen as the main comparison criterion was the scope of a toolkit—an indicator that measures the amount of programming workload that can be avoided through the use of a toolkit. This criterion was chosen not only because it abstracts away the abundant technical differences among existing toolkits. But it is also of interest for a user wondering how much support he/she can receive from a toolkit, i.e. how much programming work he/she can avoid. Additionally, other aspects of the toolkits have also been contrasted throughout the paper. The formal definition and the way to measure the scope of a toolkit will be shown in future sections.

An important finding that resulted from this comparative study is that the evaluated toolkits can be clustered into three groups such that every toolkit within a group has the same scope. These three groups are flow-based, state-based and token-based toolkits. Furthermore, toolkits within a class also exhibit resemblance in their underlying visual languages. Flow-based toolkits do not allow depicting symbols for representing system states. State-based toolkits allow using one symbol, usually a circle, to represent each state of a system. And token-based toolkits allow one symbol to represent the state variables of a system and another one, usually a token, to represent the values assigned to a state variable.

This article is organized as follows: Section 2 describes the architecture of a multimodal system. Section 3 presents the definition of the scope of a toolkit and shows that this yields unambiguous measurements. Section 4 describes how to measure the scope of an arbitrary toolkit. Section 5 outlines the results obtained after measuring the scopes of several toolkits. Section 6 describes a representative set of flow-based toolkits. For each toolkit, a strong emphasis is placed on explaining the semantics of its underlying visual language through a running example. This emphasis is justified because the visual language is the means through which users exploit the functionalities incorporated in the toolkit framework. At the end of this section, a comparison between the scopes of flow-based toolkits is made. Sections 7 and 8 have the same structure as Section 6 but for the case of state-based and token-based toolkits, respectively. The article is concluded by a discussion and conclusions in Sections 9 and 10, respectively.

2. ARCHITECTURE OF A MULTIMODAL SYSTEM

The architecture of a software system is an abstraction of that system (Len *et al.*, 2000). This abstraction is defined by describing the set of structures needed to reason about the system, and the relations among those structures (Clements *et al.*, 2010).

The study of several multimodal systems (Bui, 2008; Johnston *et al.*, 2002; Flippo *et al.*, 2003; Oviatt, 2003; Neal *et al.*, 1989; Vo and Wood, 1996; Wahlster *et al.*, 2001) reveals that, despite their differences, it is still possible to distinguish functionalities that are common to most of them. The components that implement these common functionalities will be referred to as recognizers, fusion engine, dialog manager, fission component, synthesizers and knowledge sources. The relation among these components is depicted in Fig. 2 and their functionalities are described below.

While a user is interacting with a multimodal system, inputs are recognized by a group of specialized components called recognizers. Each recognizer is continuously sensing the external environment: it captures natural input from the user and transforms it into an appropriate format. Some examples of these components are handwriting, voice and facial expression recognizers whose implementation usually involves stochastic models such as hidden Markov chains, Kalman filters or particle filters (Mitra and Acharya, 2007; Monwar and Gavrilova, 2011; Rabiner, 1989; Sushmita, 2007; Trung, 2008). This probabilistic treatment is usually intended to cope with the uncertainty caused by the noise of the environment.

Whenever a recognizer has interpreted a stream of user inputs, it informs the fusion engine (Lalanne *et al.*, 2009). This engine is in charge of merging the information provided by all the recognizers in order to interpret the user's request. Since recognizers do not necessarily complete their work at the same time, a fusion engine must be able to synchronize several recognizers' outputs so that the fusion can be performed only after all the relevant data are available. But synchronization is not the only matter to be considered when implementing a fusion engine; the information provided by the recognizers may be ambiguous or contradictory, and therefore the fusion engine must have disambiguation and error recovery mechanisms.

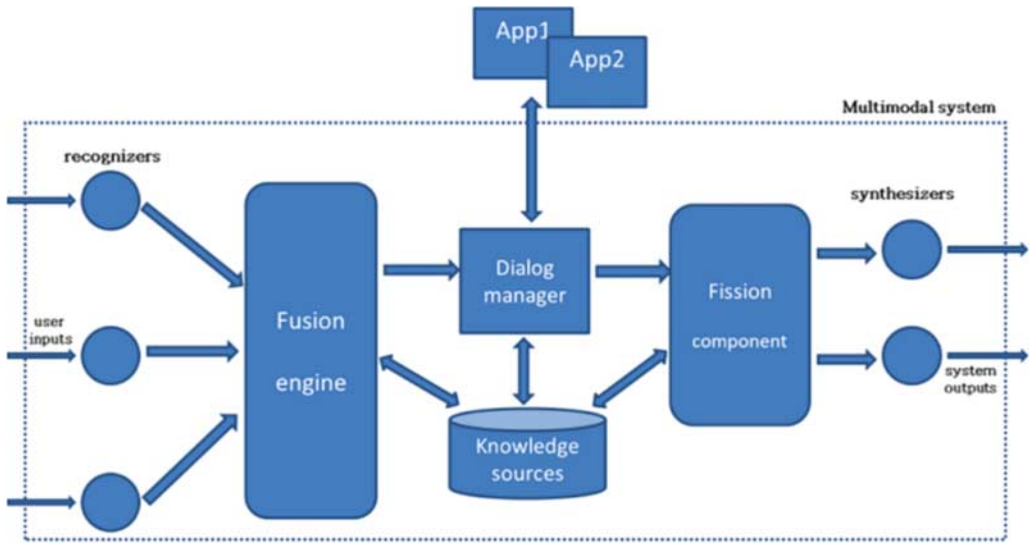


FIGURE 2. Architecture of a multimodal system performing fusion at the semantic level (Sharma *et al.* 1998). These systems are characterized by the use of independently developed recognizers whose outputs are not merged before reaching the fusion engine. These are the types of systems targeted by the toolkits under study.

Since the same user's request may result in different system reactions depending on the context, the dialog manager must track the status of the human–machine dialog so that user requests can be addressed correctly. For instance, the voice command 'yes' may have distinct meanings at different stages of a human–machine dialog. Beyond this, the dialog manager can also request services from external applications, e.g. when a system for cinema seat reservation recognizes the intended seat's positions of the user and notifies a reservation agent through an external service (Wahlster *et al.*, 2001).

After the dialog manager has decided on the response to be sent, it delegates this task to the fission component, which must then choose and activate the synthesizers—computer programs that control rendering devices—that are best suited for the situation. Whereas the fusion engine merges the information coming from different input modes, the fission component dissociates the returning message through several output modes. The generation and coordination of multiple outputs through the selected modalities is the task of the fission component, e.g. it can control a graphics display and a voice synthesizer in order to highlight the object that is being mentioned in the simultaneous natural language output (Neal *et al.*, 1989).

Finally, the interpretation of the fusion engine and the decisions made by the dialog manager or the fission component may not depend only on the user's actions. Personal information about his/her age, gender, language or disabilities may affect the interpretation and/or response of the system. The system may also need information that is particular to its application domain, e.g. when a multimodal restaurant recommender system (Rajman *et al.*, 2004) consults a database to retrieve information of the names, addresses and telephone numbers of a series of restaurants. All relevant information needed for decision-making is contained in data storages called knowledge sources.

Since the creation of prototypes has to be efficient, the toolkits do not expect their users to implement software for recognition or synthesis of modalities. Rather, they already incorporate a wide assortment of recognizers and synthesizers; which can be exploited by the client application. With regard to the knowledge sources, these are expected to be created with a database management system and without toolkit intervention. For these reasons, we are only interested in determining the support provided by a toolkit to the implementation of the fusion engine, dialog manager and fission component.

3. SCOPE OF A GRAPHICAL TOOLKIT FOR RAPID PROTOTYPING OF MULTIMODAL SYSTEMS

Several researchers have presented study cases showing that their proposed toolkits do support the implementation of multimodal prototypes (Bourguet, 2003; De Boeck *et al.*, 2007; Dragicevic and Fekete, 2004; Dumas *et al.*, 2010; Lawson *et al.*, 2009; Navarre *et al.*, 2006, 2009; Werner *et al.*, 2010). For instance, Vanacken (2009) shows how CoGenIVE can be used to create a multimodal prototype that allows its end-users to manipulate objects from a virtual world by using a tracking glove and speech inputs. (Dragicevic and Fekete, 2004) show that Input Configurator (ICon) can support the development of a graphical application that can be controlled with a wide assortment of hardware devices like palmtops or trackballs. Navarre *et al.* (2009) show that PetShop can support the creation of a multimodal prototype intended to control the orientation of a 3D virtual satellite. The complexity and sophistication of the aforementioned prototypes may be misleading. Readers might associate the complexity of these prototypes with the goodness of the toolkits used to create them. However, such association can be mistaken because significant parts of a complex system could have been built with fine-grained programming code at the client side and with no participation of the toolkit.

The scope of a graphical toolkit for rapid prototyping of multimodal systems is a concept we are proposing to refer to its capability for facilitating the implementation of the fusion engine, dialog manager or fission component of a multimodal prototype. When properly measured, this concept will make us aware of how much functionality of an intended multimodal prototype can be delegated to the framework of a toolkit (by means of visual models) and how much has to be implemented in a client application. The more functions that can be delegated to the framework of a toolkit, the more programming work its users can avoid. Therefore, the scope of a toolkit would be an important indicator to estimate how well a toolkit accomplishes its ultimate goal of shortening the prototyping phase. This concept must be defined with enough precision so that it can be measurable, thus comparable.

Definition 1. *Let T be the set of all graphical toolkits for rapid prototyping of multimodal systems and $C = \{\text{fusion engine}, \text{dialog manager}, \text{fission component}\}$. The scope of a graphical toolkit for rapid prototyping of multimodal systems is a mapping function S such that*

$$\begin{aligned} S : T &\longrightarrow 2^C, \\ t &\mapsto \{x \in C : \textit{Some functionalities of the component} \\ &\qquad \textit{`}x\textit{' can be delegated to the framework of `}t\textit{'}\}, \end{aligned}$$

where 2^C is the power set of C, i.e. the set containing all subsets of C.

The set 2^C will be the scale to be used for measuring the scope of a toolkit—hereafter referred to as the scale of measurement (Stevens, 1946) of the scope. This definition satisfies three properties that allow us to claim that we have defined the scope of a toolkit as a nominal variable (Engel and Schutt, 2013).

First, the values of the scale of measurement have no mathematical meaning. For instance, let 'r' and 't' be two toolkits such that their scopes are $S(r) = \{\text{fusion engine, dialog manager}\}$ and $S(t) = \{\text{dialog manager, fission component}\}$, respectively.

This means that the utilization of toolkit 'r' can reduce the programming effort involved in the implementation of the fusion engine and dialog manager. And the same is true for the toolkit 't' with regard to the dialog manager and fission component. However, no quantitative information can be obtained about toolkits 'r' and 't'. Moreover, we cannot add, subtract, multiply or divide $S(r)$ and $S(t)$. These observations disclose the qualitative character of $S(r)$ and $S(t)$. Secondly, the scale of measurement has collectively exhaustive values. This means that the scope of any arbitrary toolkit can take on at least one value of the scale of measurement. The support for this claim comes from the fact that, by definition, the power set 2^C contains all the possible combinations of components included in C. Thirdly, the scale of measurement has mutually exclusive values. This means that the scope of any arbitrary toolkit can take on only one value, which is directly verifiable from the correspondence rule used in Definition 1.

To clarify matters, the previous definition does not indicate how to measure the scope of a toolkit. Rather, it indicates how to record a measurement value: it will be recorded as a set, not as a number. As to the collective exhaustiveness and mutual exclusivity of the scale of measurement, they warrant that there will never be situations where there are no values or many values that can be correctly used to represent the scope of a toolkit.

Treating the scope of a toolkit as a nominal variable not only leads to unambiguous measurements. Additionally, nominal variables admit the equivalence operation (Engel and Schutt, 2013), which implies that the previous definition allows us to measure and determine whether the scopes of two arbitrary toolkits are equal or not.

The contents of this section ensure that the scope of a toolkit can be unambiguosly measured. The following is an explanation of how to measure it.

4. METHODOLOGY

The formalism shown in the previous section only guarantees that once the scope of a toolkit is determined, one and only one value of the scale of measurement can be correctly used to record this result. However, it does not mention how to measure the scope of a toolkit.

The study of several toolkits, along with their visual modeling languages, made us devise rules for determining the functionalities that are pre-programmed in their frameworks.

(1) *The toolkits that facilitate the implementation of the fusion engine are those that allow for composite events in their visual models.*

A composite event is a group of events, along with the temporal constraints among them; its occurrence reveals that a multimodal command has been issued. Consider, for instance, a speech input 'zoom' followed by the simultaneous occurrence of a speech input 'here' and a finger touch (Fig. 1). All these events in the specific order previously mentioned define a composite event whose occurrence discloses the user's intention to zoom in a region of the GUI. The toolkits that do not allow composite event definitions are confined to notify the client application of each event that occurs in the outside world. The client application has to verify whether these streams of events are reflecting the user's intention to issue a multimodal command or not. Such verification is implemented by time-stamping the events and checking whether their order of receipt matches with some meaningful pattern. In contrast, toolkits allowing the definition of composite events permit their users to specify event patterns that are of interest for the client application and omit the need to process each event separately. The possibility to include composite events in a visual model enables us to delegate their detection to the (framework of the) toolkit, thus reducing the programming workload at the client side. The detection of composite events discloses the user's intent, this being a function of the fusion engine.

(2) *The toolkits that facilitate the implementation of the dialog manager are those allowing the depiction of the system's states.*

The responses provided by a multimodal system may depend on its current state, which is continuously changing during the human–machine dialog (Luyten *et al.*, 2003). For example, a command to zoom in a region of a GUI may lead to its enlargement or to an error message if the target area has previously been enlarged.

When a toolkit does not allow depicting the states the intended prototype may ever be in, its client application has to be overloaded because it now has to be constantly determining the prototype's current state. This task entails maintaining global variables across different event handlers, and evaluating them at different branching points (if-else conditions) (Samek, 2009). In contrast, by using an appropriate toolkit, users can release their client application from this tracking operation. Toolkits allowing one to depict the potential states of the intended prototype guarantee that its current state will always be correctly identified after any arbitrary sequence of events. This identification is pivotal to undergo context-dependent human–machine dialogs, this being the goal of the dialog manager. For some toolkits, the state of the system is depicted by using one designated symbol; for others, it is required to depict the system's state variables and enter their initial values.

(3) *The toolkits that facilitate the implementation of the fission component are those including language constructs for concurrency and synchronization.*

The concurrent activation of synthesizers allows conveying the returning message through multiple modalities; the synchronization of their outputs makes the returning message intelligible to the end-user. For instance, a system implementing an animated avatar must be concurrently activating a display manager and a voice synthesizer in order to portray the avatar's face and generate its voice, respectively. Additionally, both outputs have to be constantly synchronized so that the avatar's face can always be in accordance with its speech. Petri net graphs, which can be interpreted by some toolkits, are well-known visual models that can represent the synchronization of concurrent processes. The inability to express concurrency and synchronization in a visual model will lead to the implementation of a multithreaded client application. However, manual implementation of multiple threads is error-prone and cumbersome.

The toolkits whose study allowed us to devise these rules were ICon (Dragicevic and Fekete, 2002), OpenInterface (Lawson *et al.*, 2008), Squidy (Werner *et al.*, 2010), MEngine (Bourguet, 2002), CoGenIVE (De Boeck *et al.*, 2007), HephaisTK (Dumas *et al.*, 2009) and PetShop (Navarre *et al.*, 2009). For all these toolkits, the semantics of their visual languages were revealed from the analyzes of the numerous running examples discussed in the papers, from the study of the theses that refer to these toolkits, from the observations of publicly available demos and/or from e-mail correspondence with some members of their development teams.

Because the visual languages of many toolkits are variations of finite state machines (Wagner *et al.*, 2006) or object Petri nets (OPNs) (Lakos, 1991; Sy *et al.*, 1999), the mathematical apparatuses behind these formal models were an additional source of information we used to decipher the semantics of the said toolkits.

As regards ICon, Squidy and PetShop, they were publicly available and could be successfully installed and evaluated. CoGenIVE was also available for evaluation since it was developed in our research lab. The evaluation of these toolkits consisted of exploring corner cases that were not commented upon in the papers, thus increasing our understanding of their underlying semantics.

The proposed rules allowed us to easily evaluate the scopes of several toolkits.

5. EVALUATION

The heuristic rules exposed in the preceding section were used to evaluate the scopes of several toolkits. The measurements obtained unveiled that the studied toolkits can be classified into three groups (Table 1).

For the first group, their toolkits work as servers for event recognition but they do not facilitate the fusion or fission of modalities, or the dialog management. Their scopes are the empty set {}. These toolkits were called *flow-based* toolkits (Section 6) because their visual models are intended to specify how the data flows from the input devices to the client application.

A second group of toolkits support the implementation of the fusion engine and dialog manager, i.e. their scopes are {fusion, engine, dialog manager}. These were called *state-based* toolkits (Section 7) because of the resemblance of their visual models with state diagrams.

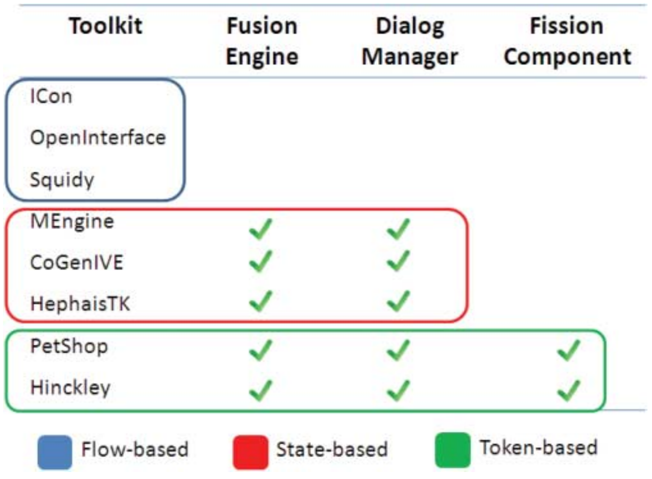
Finally, the third class of toolkits facilitates the implementation of the fusion engine, dialog manager, and fission component, i.e. their scopes are the set C defined above. These were called *token-based* toolkits (Section 8) because the models designed with their editors use a symbol called token to allow for modeling parallelism.

The next section presents a representative set of flow-based toolkits.

6. FLOW-BASED TOOLKITS

A toolkit is called flow-based when its graphical editor does not allow depicting the state of a system. In the remainder of this article, any diagram that can be depicted with the editor of these toolkits will be called a flow-based model. These are directed

TABLE 1. Checkmarks indicate the components whose implementation is facilitated through the use of a toolkit. Toolkits can be clustered into three groups.



Toolkit	Fusion Engine	Dialog Manager	Fission Component
ICon			
OpenInterface			
Squidy			
MEngine	✓	✓	
CoGenIVE	✓	✓	
HephaisTK	✓	✓	
PetShop	✓	✓	✓
Hinckley	✓	✓	✓

🟦 Flow-based 🟥 State-based 🟩 Token-based

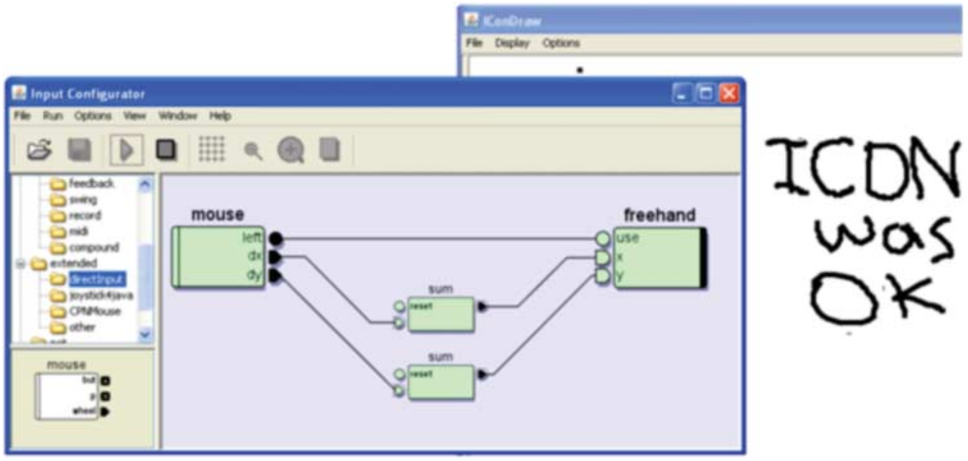


FIGURE 3. ICon's diagrams specify the transformations applied to the data that flow from input devices towards a client application.

graphs whose links allow data to flow in the direction of their arrowheads and whose nodes perform operations with the data that flow through them. In this section, we describe our study of ICon, OpenInterface and Squidy.

6.1. Input Configurator

ICon (Dragicevic and Fekete, 2002, 2004) is a toolkit that allows a client application (implemented in Java) to support a large set of heterogeneous hardware devices without the need of low-level configuration programming code. Therefore, ICon enables users with no programming skills to test their applications with a wide variety of input devices. The possibility to easily define and redefine the input devices supported by a client application encourages innovation (Dragicevic and Fekete, 2004).

Diagrams made in ICon are composed of nodes, called devices, connected by links. Figure 3 shows a diagram used to control some functionality of an application called IConDraw. In this diagram, the device mouse is representing the hardware to be used as input source. When the left button of the mouse is down, its corresponding virtual device is continuously sending signals through its output port left. Similarly, delta values are sent through ports dx and dy whenever the mouse is moved. As to the devices *sum*, they are in charge of transforming the delta values (dx, dy) into cursor locations (x, y). In summary, this diagram establishes that every time the user is pressing the mouse, the free-style mode of IConDraw will be activated and thus the user will be able to paint free-form shapes by moving the mouse over the IConDraw's canvas.

Each ICon device belongs to one of three categories: system, library or application devices. System devices are virtual representations of the hardware used to interact with the system, e.g. mouse, trackball, palmtop. Library devices are intended to process the signals generated by system devices before finally passing them to the application devices. In Fig. 3, both nodes sum are library devices and freehand is an application device. Whereas system and library devices are predefined, application devices belong to external applications and are thus extensible. In order to define an ICon-recognizable application device, the client application must implement classes that extend from AbstractDevice, found in the libraries of ICon.

With respect to the architecture shown in Fig. 2, ICon serves as a library of ready-to-use input recognizers but does not support fusion of modalities (Dragicevic and Fekete, 2004). ICon permits its users to acquire information from the external environment and apply simple transformations to this data before sending it to a client application, which must implement the functionality of a fusion engine, dialog manager and fission component.

6.2. OpenInterface

OpenInterface (Lawson *et al.*, 2008, 2009; Pineux, 2012) is an open-source software solution designed to support rapid prototyping of interactive multimodal systems. SKEMMI, its graphical editor, allows specifying multimodal systems by means of visual models that will then be executed on the OpenInterface runtime platform. Models created with SKEMMI are directed graphs composed of two symbols: nodes and links. The nodes represent components that are autonomous pieces of software that implement the functionalities of a multimodal system. These components can be developed in different programming languages, but their interfaces have to be described in an XML-based language called Component Interface Description Language. On the other hand, the links are intended to connect the input ports of one component to the output ports of another, thus indicating the path in which the data can flow. Once a group of components have been built, a multimodal system can be specified by plugging and wiring these components. SKEMMI also offers

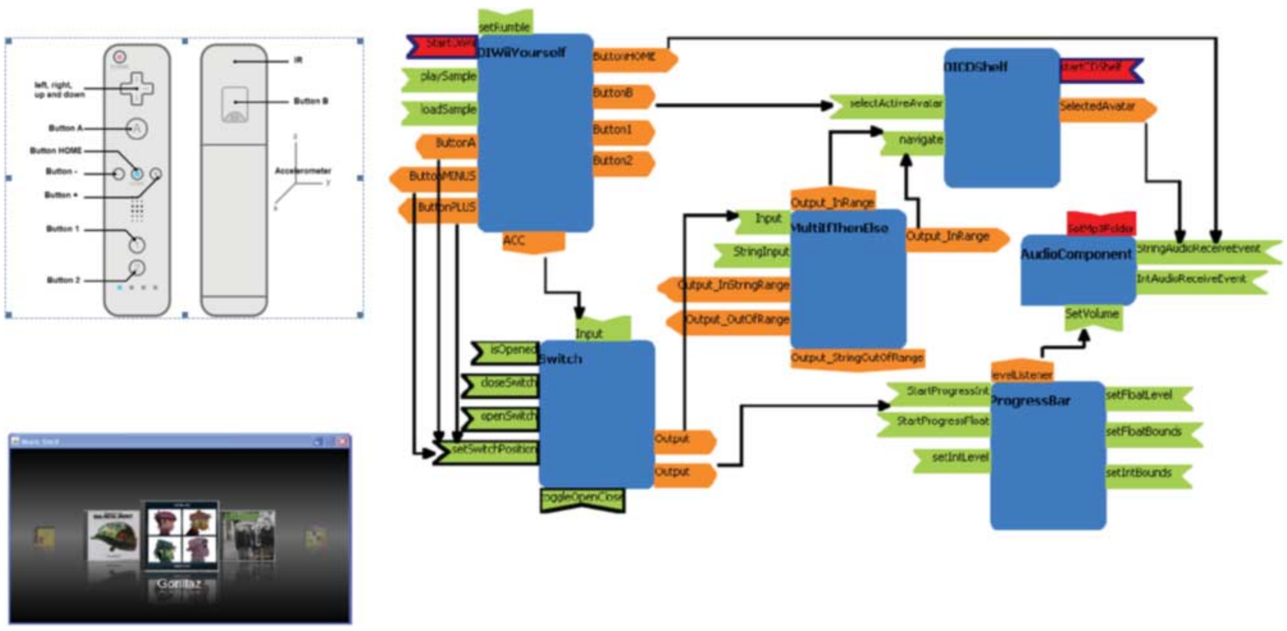


FIGURE 4. *Upper left:* Wiimote control. *Lower left:* OICDShell is a component whose graphical interface shows the album covers (pictures) of a series of available albums. *Right:* SKEMMI model of an MP3 player. [*Images from* (Pineux, 2012), *reproduced with permission.*]

predefined components, called OI Adapters, which provide common functionality such as filtering, buffering and data transformation.

As an example, consider an MP3 player that can be handled by a Wiimote control. Specifically, button A will set the system in album navigation mode, whereas buttons + or − will change it into volume mode. Moreover, signals sent by the Wiimote's accelerometer will adjust the volume of the music or allow navigating through a list of albums depending on whether the system is in the volume or navigation mode. The specification of the described system is shown in the right-hand side of Fig. 4. Components OIWiiYourself, OICDShell and AudioComponent, referred to in Fig. 4, have been developed by the user. OIWiiYourself recognizes the events generated by the Wiimote control, OICDShell is the GUI that shows the list of available albums and AudioComponent allows playing/stopping a song and increasing/decreasing the volume of the MP3 player. The three additional components seen in the right side of Fig. 4, MultiIfThenElse, ProgressBar and Switch, are called OI Adapters and are ready-to-use predefined SKEMMI components. The adapter MultiIfThenElse is used to transform the value received through its input port into -1 or $+1$ to indicate whether the previous or the next album cover must be displayed by OICDShell. The ProgressBar adapter shows a progression bar that enables the user to set the volume between 0 and 1 as required by AudioComponent. Finally, the adapter Switch must route the signals generated by the accelerometer of the Wiimote –to the OICDShell or to AudioComponent depending on whether the system is in navigation mode or volume mode.

It is important to highlight that the identification of the context (MP3 player's operation mode) is the responsibility of the component Switch and not of the OpenInterface platform. In other words, the identification of the context has to be written in the source code of a component and cannot be delegated to the framework by means of graphical specifications (as is the case of state-based models we will elaborate upon later). Thus, it cannot be said that the studied toolkit facilitates the prototyping of a dialog manager.

Unlike ICon, OpenInterface routes the data generated by input devices to several software components (possibly implemented in different programming languages), whereas ICon routes it to a single client application (developed in Java).

6.3. Squidy

Squidy (Werner *et al.*, 2010) is another toolkit that provides a visual language based on dataflow programming. Its graphical editor allows users to select nodes from a predefined collection, called knowledge base, and to connect them through wires in order to create a directed graph, called pipeline (Fig. 5), which aims to specify an interaction technique. Some nodes are virtual representations of hardware devices—like a mouse, laser pointer, mobile phone or a printer—whereas others are filters—like gesture recognizers or Kalman filter—to process the information flowing through the pipelines.

Squidy's graphical editor allows users to configure devices or filters by double-clicking on their corresponding nodes and setting a list of relevant parameters. The capability to enable

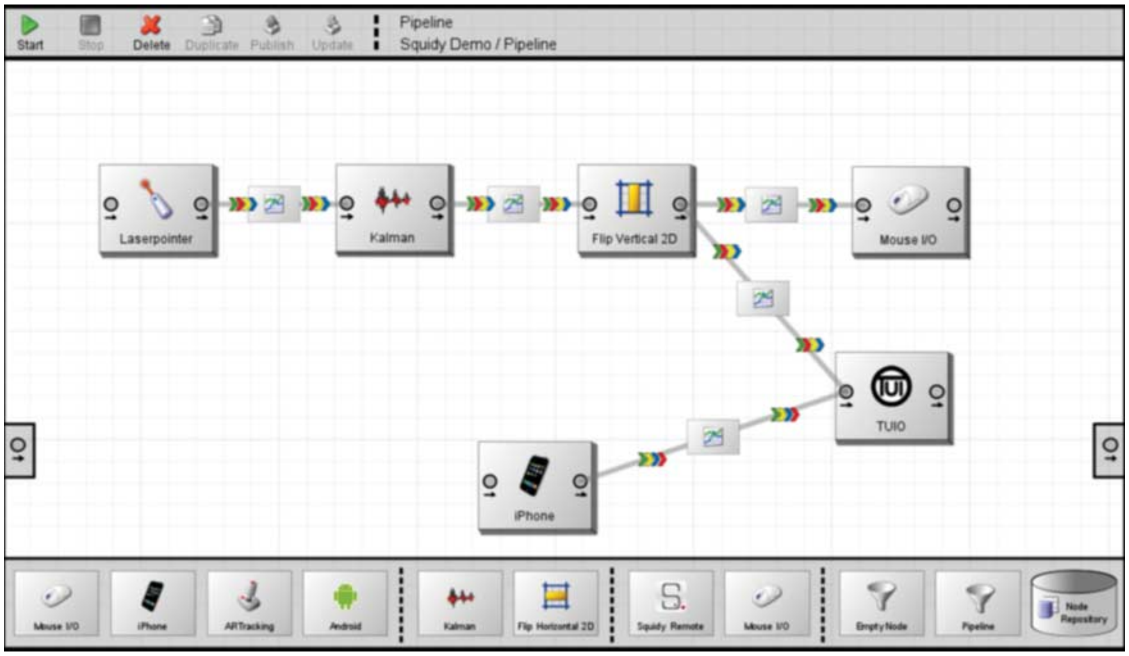


FIGURE 5. Pipeline specifying that the data generated by a laser pointer and a mobile device must be directed to a TUIO client application. Additionally, the mouse cursor position will be controlled with the laser pointer.

access to more advanced functionality on demand is called semantic zooming. Furthermore, users can also monitor the signals flowing through the pipeline by double-clicking the wires. Signal analysis can be useful to verify whether some data processing, e.g. filtering, is needed. For every signal generated by an input device, a message is sent through the outgoing arcs of its corresponding node. These messages will continue flowing downstream. Every message that flows through the pipelines is encapsulated in an object that implements a generic interface, IData. Therefore, unlike ICon and SKEMMI, there is only one link between two connected nodes. The use of generic data types rather than low-level ones not only prevents graphs from abundant wiring, but also releases users from recalling the primitive data types of the flowing information.

Another important feature of Squidy is the possibility to add empty nodes (from the knowledge base) in a diagram and to embed programming code into them. In this way, new functionalities can be added to a pipeline, thus increasing the range of interaction techniques that can be modeled. Squidy's programming editor can be opened by double-clicking on an empty node.

Like ICon and OpenInterface, prototypes specified in Squidy can route the data generated by input devices to existing applications. The only requirement for these applications is to use the TUIO protocol (Kaltenbrunner *et al.*, 2005). Figure 5 shows how to route the data generated by a laser pointer and a mobile device to an external application represented as a node labeled TUIO. By double-clicking on a TUIO node, users can set the address and listening port of the external application. In this model, the data generated by the laser pointer is Kalman-filtered to reduce the imprecision produced by the natural hand tremor.

In contrast to ICon and OpenInterface, Squidy also allows routing input data directly to an output device (without needing an external application). This possibility enables users to define interaction techniques like controlling the mouse cursor position with a laser pointer, digital pen, palmtop or other hardware device as seen in the upper branch of Fig. 5.

In order to wrap up this section, the support provided by the studied toolkits to the development of multimodal systems will be commented upon.

6.4. Comparing the scopes of flow-based toolkits

Based on the toolkits presented in this section, the following observations have been made:

- Prototyping a fusion engine by using ICon or Squidy is not possible. The fusion requires collecting data coming from multiple recognizers in order to interpret user requests. Unfortunately, the semantics of these languages do not allow their nodes to postpone their execution until the reception of a meaningful set of data. Rather, these nodes are continuously transforming and reinjecting the data they receive as soon as they arrives. With regard to OpenInterface, (Lawson *et al.*, 2009), claim that it supports fusion of modalities. Unfortunately, we could not verify it during this research.

TABLE 2. Components whose implementation can be facilitated through the use of a flow-based toolkit.

	Fusion engine	Dialog manager	Fission component
Icon	×	×	×
OpenInterface	×	×	×
Squidy	×	×	×

- With regard to the dialog manager, it must track the state of its conversation with the user in order to allow context-dependent dialogs. Unfortunately, flow-based models are stateless, i.e. the state of a system cannot be depicted. Consequently, the task of tracking the state of a system cannot be delegated to the framework, but has to be programmed at the client side.
- Flow-based toolkits do not simplify the prototyping of a fission component either. They are limited to route the data obtained from user inputs to a client application that must be responsible for rendering multimedia output. That is, the client application must include the logic required for selection and coordination of synthesizers.

These observations are summarized in Table 2.

For flow-based models, the nodes that represent input devices (like mouse in Fig. 3, OIWiiYourself in Fig. 4, or Laserpointer in Fig. 5) are representing the recognizers associated to those devices. Then, the support provided by a flow-based toolkit during the implementation of a multimodal system is mainly limited to the specification and configuration of these recognizers. Put in another way, the users of flow-based toolkits can release their applications from including programming code for input recognition, but the fusion and fission of data, as well as the dialog management have to be implemented in the client application with no support from these toolkits.

7. STATE-BASED TOOLKITS

A state-based toolkit is supported by a graphical editor that offers exactly one symbol to depict each possible state of a system. In the remainder of this article, any diagram that can be edited with these toolkits will be called a state-based model. State-based models resemble widely known state diagrams that, in their canonical form, fail to express synchronization and concurrency. However, they are well suited for modeling the states of a human–machine dialog from which relevant aspects of its history can be inferred. In this section, we are going to study the state-based toolkits supported by the frameworks MEngine, CoGenIVE and HephaisTK.

7.1. MEngine

For a traditional WIMP system, the detection of a single user event is enough to identify the user's intent, e.g. a click on a button *Accept* or *Cancel* of a GUI is enough to realize whether a user wants to process or close a form, respectively. In contrast, a multimodal system allows its users to dissociate a command so that it can be conveyed through multiple modalities. Therefore, the detection of a single user-event is not enough to understand his/her request. Rather, it is required to wait for a series of user-events that together enclose the user's intent. These meaningful sets of events will be called composite events.

In order to allow a client application to detect composite events, a toolkit consisting of two components, IMBuilder and MEngine, was proposed (Bourguet, 2002, 2003). Briefly speaking, IMBuilder allows specifying composite events as state diagrams, whereas MEngine is in charge of detecting those composite events and notifying about the client application their occurrence. The nodes of the said state diagrams represent the potential states of the system, and the arcs, its state transitions. Every arc of a state diagram can expose two annotations: one to indicate the event that makes a system change its state, and other to specify the subroutine that must be executed during this transition.

For illustrative purposes, assume that, for an existing client application, it would be desirable to detect when a user is trying to move an object from its original position to a different one. In order to perform this action, the user clicks on an object, utters the voice command 'move' and clicks on the new position of the object. Alternatively, he/she can also click on an object and then on its new position after the word 'move' is uttered. Both sequences of events are specified in IMBuilder as shown in Fig. 6. The implementation of the client application would be facilitated if this could be directly notified about the occurrence of the composite event 'move-object' and not of the multiple and meaningless atomic events that compound it. It is precisely MEngine, the framework, that is in charge of notifying the client application about the occurrence of composite events. To accomplish its work, MEngine is continuously tracking the state of the system by checking one or several state diagrams. The client application must redirect the atomic events it detects to MEngine.

With regard to Fig. 6, when MEngine is loaded, it moves to state 1. Then, every time MEngine is informed by the client application of the occurrence of some event, the transition whose uppermost label (blue) refers to this event will cause MEngine to change its state, and the client application to execute the subroutine indicated in the red label. Reaching the final state, labeled as 5, implies that the composite event 'move-object' has already been detected, and the subroutine *moveObject* has been launched at the client side.

The advantage of this approach is that it releases programmers from writing code to detect the occurrence of composite events, which is usually performed by updating a set of global variables across different event handlers. On the other hand, the creation of a multimodal system that enables its end-users to request a service by performing actions in his/her preferred order will lead to redundant specifications, e.g. in Fig. 6, arcs 2–4 and 1–3,

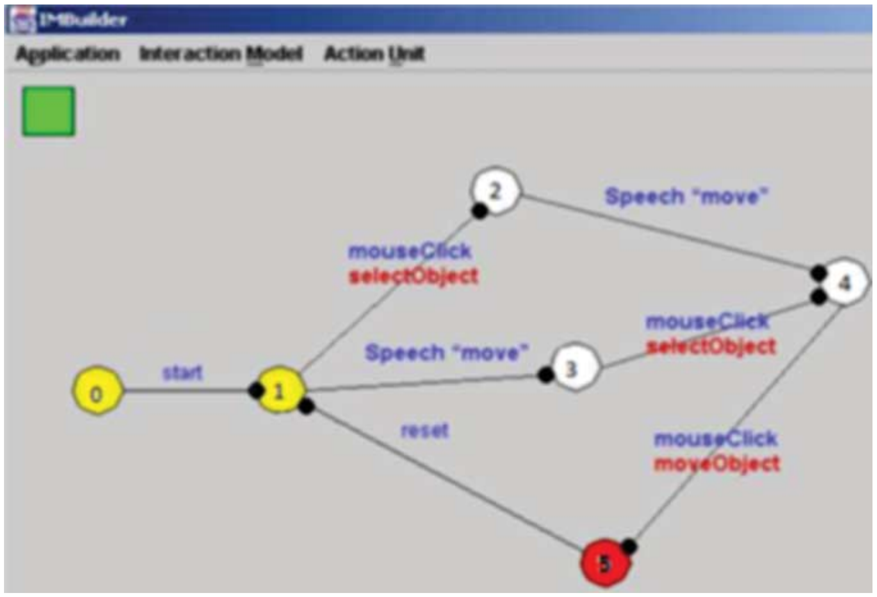


FIGURE 6. Composite event 'move-object' specified in IMBuilder. [*Images from* (Bourguet 2002), *reproduced with permission.*]

and 1–2 and 4–5 are redundant. This may cause an exponential blow-up in the size of the state diagrams.

To sum up, MEngine helps a client application to decode the user's intent. Nevertheless, the selection and/or coordination of synthesizers in charge of responding the user cannot be delegated to MEngine; rather, it has to be programmed in the client application.

7.2. CoGenIVE

NiMMiT (De Boeck *et al.*, 2007, 2009; Vanacken *et al.*, 2006) is a visual language intended to model the ways in which an end-user can interact with a virtual world. CoGenIVE, its supporting framework, can be conceived as a black box that receives a set of interaction techniques, specified in NiMMiT, and a virtual world, specified in X3D, as inputs; and outputs the programming code of an interactive virtual environment where user behavior is regulated according to the specification of NiMMiT models. In a NiMMiT model, the circles represent the states of the system; the arcs, user events; the rectangles, the tasks the system executes; and the labels, the information the system stores/retrieves during its execution. Tasks can be grouped into containers called task chains that are executed during the transitions. NiMMiT offers a library of predefined tasks like selecting, moving or deleting a virtual object. Users are also supported to create their own tasks by means of the scripting language Lua. An important feature provided by NiMMiT is the possibility to group events in order to create composite events. A NiMMiT composite event consists of a set of atomic ones and is triggered whenever all its constituent events are triggered simultaneously. For illustrative purposes, the well-known put-that-there interaction technique (Bolt, 1980) is specified as shown in Fig. 7. Shortly speaking, the put-that-there interaction technique specified in the right side of Fig. 7 permits creating a system that allows moving an object by using speech and mouse clicking. The end-user must utter the sentence 'put that there' to move an object from its original position to a new one. In order for the system to interpret the meanings of the utterances 'that' and 'there', the user must click on an object and on any arbitrary position while pronouncing these words, respectively.

Figure 7 contains one circle to represent every possible stage of the interaction. Initially, when the system is loaded, it will be in the state Start. The recognition of the utterance 'put' will change the system state to a new one labeled as Put. Then, in order to indicate the object to be moved, a voice command 'that' and a click on that object are both required. This multimodal request can be easily specified by creating a composite event that groups the events Voice.That and Mouse.ButtonPressed. When CoGenIVE detects the occurrence of this composite event, it will notify a client application to select the target object. After this selection, the system will pass to a new state label Put-That. Similarly, the simultaneous detection of a click and the utterance 'there' will make CoGenIVE require the client application to move the selected object to the current mouse position and the system will move to a final state indicating that the interaction technique has been performed.

Models depicted with NiMMiT resemble state diagrams where the actions to be executed during a transition are explicitly unfolded; being abundant diagrams, the price to pay for this explicitness. Another disadvantage of NiMMiT is that the execution of the tasks contained in a task chain

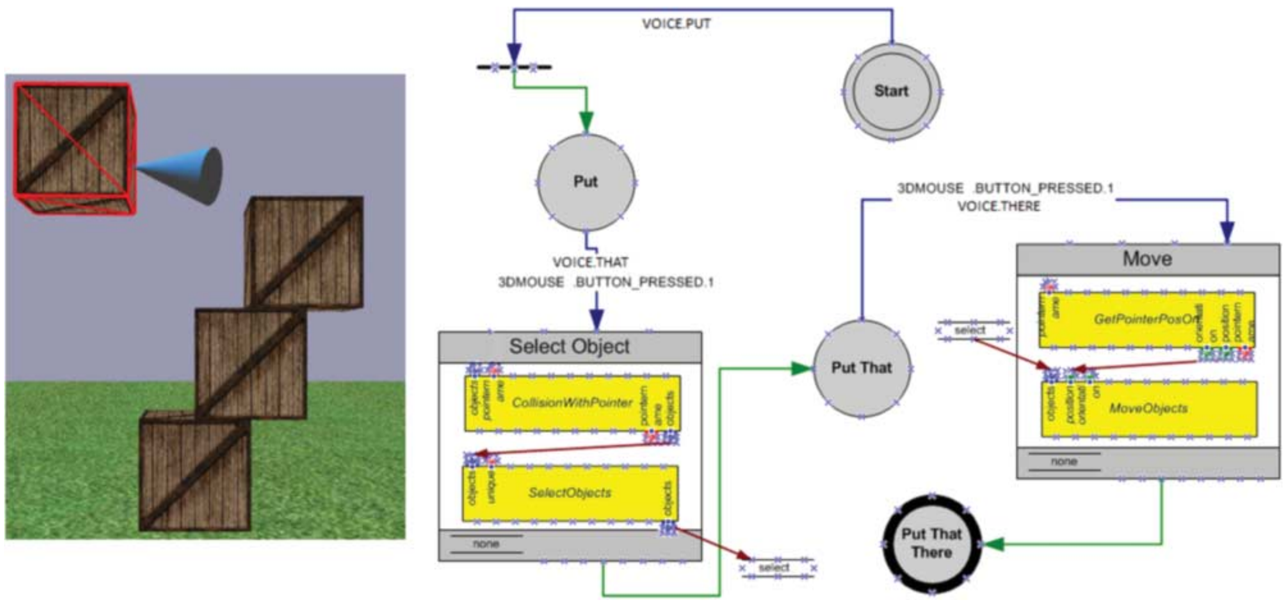


FIGURE 7. *Left:* The virtual world the end-user will interact with. *Right:* Specification of the put-that-there interaction technique.

is always sequential. The initial version of NiMMiT did not allow modeling conditional and iterative execution of tasks, thus restricting the computational capacity of task chains. In order to solve this problem, its notation was extended with new symbols, e.g. pass-through states (Vanacken, 2012), at the expense of obstructing its semantics.

7.3. HephaisTK

HephaisTK (Dumas *et al.*, 2009; Dumas, 2010) is a toolkit designed to plug itself in a client application that wishes to handle multimodal requests. The specification of the interaction techniques expected for the client application has to be written in a markup language, called Synchronized Multimodal User Interfaces Modeling Language (SMUIML) (Dumas *et al.*, 2010, 2013), for its subsequent interpretation by the HephaisTK's framework. SMUIML files are also used to specify the set of recognizers the client application wants to support and other parameters that regulate HephaisTK's behavior. Since writing XML-based files can be tedious and error-prone, SMUIML has an alternative graphical representation.

The graphical editor of SMUIML allows the specification of multimodal interaction techniques by drawing graphs that resemble state diagrams. The ellipses and directed arcs represent the states of the system and the state transitions, respectively. The triggering events that cause a state transition and the actions to be executed during such transition by a client application are annotated in the arcs.

This language allows associating arcs not only to atomic but also to composite events. The composite events that can be defined in SMUIML are more complex than the ones that

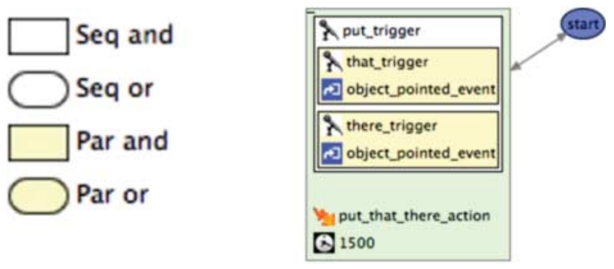


FIGURE 8. *Left:* Notation used to specify composite events. *Right:* Specification of the put-that-there interaction technique. [*Images from* (Dumas *et al.*, 2013), *reproduced with permission.*]

can be defined in NiMMiT. SMUIML's composite events are easily and compactly described by grouping atomic events into nested containers that can belong to four types: containers Seq-And indicate that their events are complementary and must be triggered in consecutive order; Par-And, that their events are complementary and must be triggered in any order; Seq-Or, that their events are equivalent and thus any of them must be triggered; and Par-Or, that a group of equivalent events must be triggered simultaneously. For instance, the put-that-there interaction technique (Bolt, 1980) can be concisely modeled as shown in Fig. 8.

Figure 8 specifies that the framework (HephaisTK) will begin in the start state and after detecting that the user is trying to move an object, it will indicate the client application to execute the *put_that_there_action* subroutine. Then, the framework will go back to its initial state and wait for new user requests. The composite event put-that-there is specified

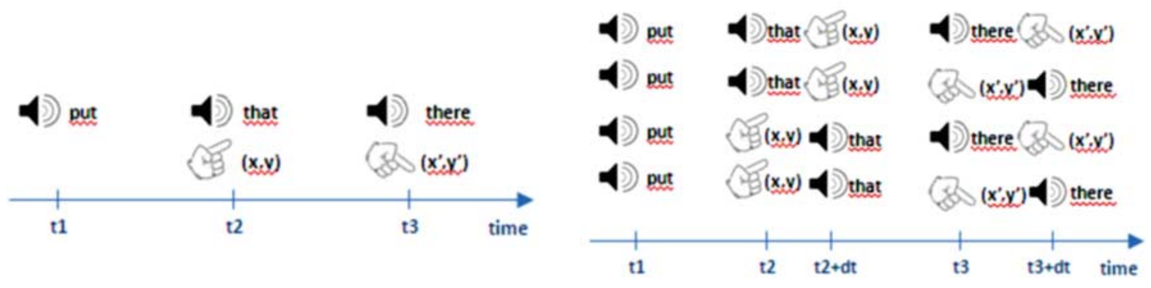


FIGURE 9. *Left:* When implementing the put-that-there interaction technique with CoGenIVE (Fig. 7), it will notify the client application at three moments: $t1$, $t2$ and $t3$. With HephaisTK (Fig. 8), it will only notify once, at $t3$, after all the events between $t1$ and $t3$ have been recognized and interpreted as a request for moving the object on (x, y) to (x', y'). *Right:* Considering that user-events are detected one by one, the put-that-there interaction technique can be specified, in IMBuilder, by depicting all the possible permutations of the quasi-simultaneous events—received at around $t2$ and $t3$—in one state diagram.

with three containers. One of them, the outermost one (white), indicates the sequential triggering of three events: command voice put, the simultaneous detection of voice command 'that' and a mouse click, and the simultaneous detection of voice command 'there' and a mouse click. These last two couples of simultaneous events are encapsulated into two containers (yellow) to indicate their parallelism.

SMUIML's graphical notation facilitates the prototyping of a fusion engine. By grouping events in appropriate containers and nesting them appropriately, the framework is told what sequences of events will be triggered because of a user's request. That information enables the framework to release a client application from handling series of atomic events that are meaningless by themselves. Rather, the client application will be only notified after the identification of the user's request, thus facilitating the development at the client side. HephaisTK also supports the prototyping of the dialog manager. The user of this toolkit only has to depict the states and state transitions of a multimodal system and then the current system state will be automatically identified, by HephaisTK, after any arbitrary sequence of events.

Finally, note that the same interaction technique that was described with four states in NiMMiT can be described with only one state in SMUIML. This influences the frequency of communication between their frameworks and their associated client applications (Fig. 9). On the other hand, one commonality of state-based models is that, unlike flow-based ones, they do not include symbols to represent the recognizers of the system. Rather, the events that can be triggered by these recognizers are included in the arc annotations.

In order to wrap up this section, the support provided by the studied toolkits to the development of multimodal systems will be commented.

7.4. Comparing the scopes of state-based toolkits

Based on the toolkits presented in this section, the following observations have been made:

TABLE 3. Components whose implementation can be facilitated through the use of a state-based toolkit.

	Fusion engine	Dialog manager	Fission component
Mengine	√	√	×
CoGenIVE	√	√	×
HephaisTK	√	√	×

- The fusion engine must identify user requests, which are expressed across a series of user actions. Each of these actions will cause a recognizer to trigger an event. Consequently, the presence of composite events in a model is indirectly indicating to the framework those sets of user actions that must be interpreted as a single request. By including composite events in a NiMMiT or SMUIML diagram, the responsibility of detecting user requests is delegated to their supporting frameworks. Similarly, every model depicted in IMBuilder is the specification of a composite event that has to be detected by MEngine. The detection of composite events unveils the user's intent.
- Tracking the state of a system is the responsibility of the dialog manager. The implementation of this functionality usually requires maintaining a set of flags and global variables (Samek, 2009). However, when using a state-based toolkit, its users only have to specify the states and state transitions of the system in a visual model. This model will be enough to permit the framework to identify the current state of the system after any arbitrary sequence of events.
- In order to render multimodal output, a fission component may require the simultaneous execution of several subroutines; each one controlling one synthesizer. Unfortunately, state-based toolkits only allow modeling systems that can execute one subroutine at a time (because state-diagrams only experience one transition at a time). Of course, one subroutine can be programmed so that it

can handle concurrent computation but this would put the burden of flow control on the programmer instead of on the framework. In short, state-based toolkits do not reduce the programming effort involved in the implementation of a fission component.

These observations are summarized in Table 3.

8. TOKEN-BASED TOOLKITS

A token-based toolkit is one whose underlying notation offers one symbol to depict the state variables that describe a system and another symbol to denote the values assigned to each state variable. In the remainder of this article, any diagram that can be depicted with these toolkits will be called a token-based model. By representing the state variables rather than the states of a system, more concise models can be obtained. In this section, we study PetShop and a language proposed in Hinckley *et al.* (1998).

8.1. PetShop

The Interactive Cooperative Objects formalism, a.k.a. ICO (Navarre *et al.*, 2006; Navarre *et al.*, 2009; Palanque and Schyn, 2003) is an approach that conceives the operation of an interactive system as a group of objects communicating among them by requesting and/or responding to the services they offer. The behavior of these cooperative objects is not specified by algorithms, but by OPNs that can be edited in a toolkit called PetShop. Petshop supports the specification, prototyping and validation of interactive software (Navarre *et al.*, 2009).

An OPN is a mathematical modeling tool whose graphical counterpart consists of a digraph composed of elliptical nodes, called places, connected to rectangular ones, called transitions. There are also dots, called tokens, which can flow through the places by obeying a so-called transition rule (Fig. 10). When used for modeling multimodal systems, the tokens represent the relevant objects of the system to be modeled, and the transitions represent events the system can detect. The distribution of tokens among the places indicates the state of the system, and their flow through the net reflects the dynamics of the interaction. When a transition is fired, functions that use the data embedded in the tokens can be called. For instance, the main functionality of the put-that-there interaction technique (Bolt, 1980) can be modeled with the OPN shown in Fig. 10 (an alternative model that uses two interrelated diagrams can be seen in Palanque and Schyn, 2003). In this net, transition put is enabled, which means that when the user utters the word put, this transition will fire, thus placing one token into $p1$ and $p2$ in order to indicate that the system is now awaiting the recognition of a click and the utterance that. After both events have been triggered (in any order), places $p3$ and $p4$ will contain tokens, thus causing the immediate firing of a transition bypass that will redirect tokens from $p3$ and $p4$ to $p5$ and $p6$, respectively, indicating that the system is now awaiting a click and utterance of ‘there’. When the user utters the command ‘there’ and clicks the mouse (in any order), the transition named notification is fired and the method ‘putThatThere’ will move the object over from (x, y) to (nx, ny). The arrangement of symbols that goes from transition put to places $p7$ and $p8$ can be seen as the definition of a composite event put-that-there. For the sake of readability, the wiring required to avoid the potential inconsistencies caused by unexpected sequences of events has been hidden (Fig. 11). In OPNs, the state of a dialog can be tracked, but not with a dedicated symbol like in the case of state-based languages. For instance, in the put-that-there example, the presence of tokens in $p1$ and $p2$ indicate that the command put has been uttered. Similarly, the presence of tokens in $p5$ and $p6$ indicates that the user has already uttered ‘put-that’ and clicked on an object. This decentralized representation of system states prevents Petri nets from the explosion of states. Unfortunately, this also makes Petri net models more difficult to create and read than state-based models.

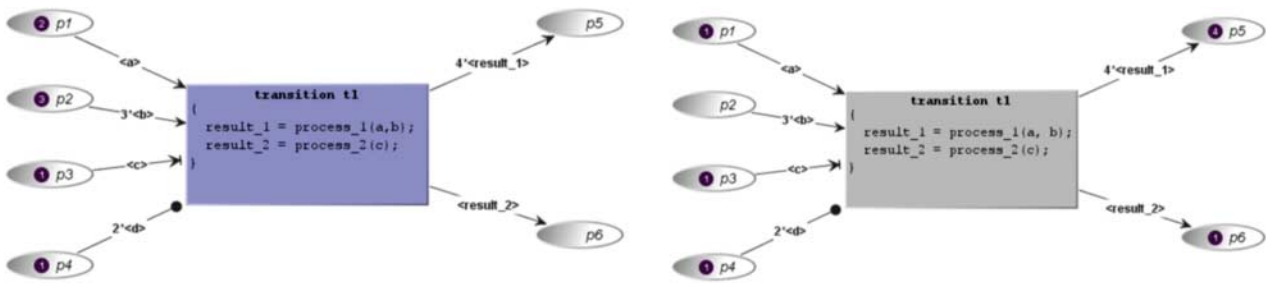

```
transition t1
{
result_1 = process_1(a,b);
result_2 = process_2(c);
}
```

FIGURE 10. *Left:* Outgoing arcs from $p1$, $p2$ are called regular arcs; those from $p3$ and $p4$ are called test and inhibitor arcs, respectively. The transition rule for OPNs (Sy *et al.*, 1999) takes into account the type of the arcs. For $t1$ to be enabled for firing, there must be one, three and one token in places $p1$, $p2$ and $p3$, respectively, and less than two tokens in $p4$. *Right:* After the firing of $t1$, one and three tokens are consumed from $p1$ and $p2$, respectively, and no tokens from $p3$ or $p4$ (test and inhibitor arcs do not consume tokens). Also, four and one token are placed in $p5$ and $p6$, respectively. During the firing, the number of tokens that are taken out from the input places and put into the output places is specified in the arc annotations.

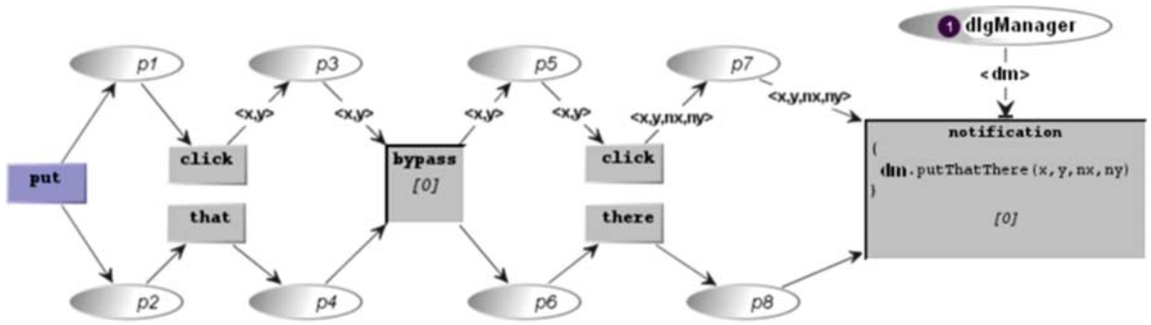
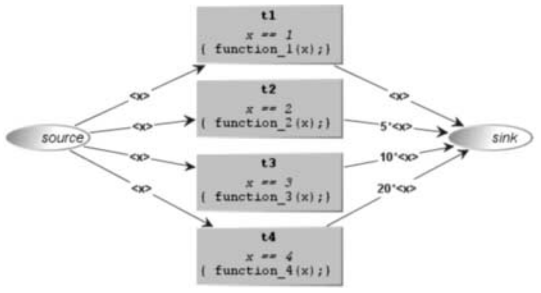


FIGURE 11. Event click and voice commands 'put', 'that' and 'there' are triggered by the user. For the first mouse click, the position of the mouse is stored in the variables x and y that will flow downstream and be attached to the variables nx and ny that are assigned during the second mouse click. These values are needed to address the put-that-there command. Label [0] in transitions bypass and notification indicates that these transitions will fire immediately after being enabled.


source
sink
t1 x == 1 { function_1(x); }
t2 x == 2 { function_2(x); }
t3 x == 3 { function_3(x); }
t4 x == 4 { function_4(x); }
<x>
5*<x>
10*<x>
20*<x>

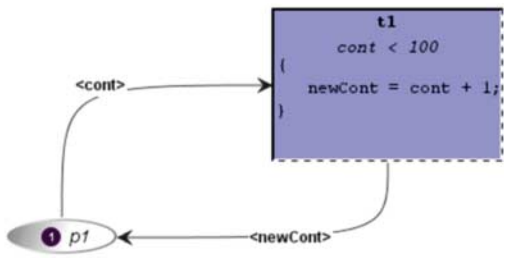
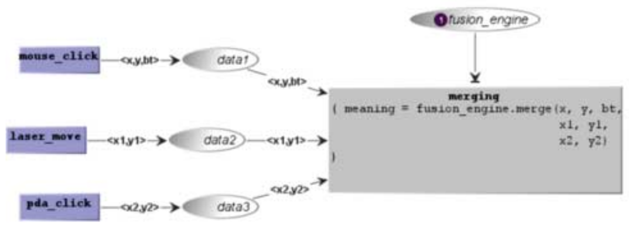


FIGURE 12. *Left:* Transition preconditions allow OPNs to express conditional flow. In the figure, for every token placed in the source, its data will be assigned to a variable x whose value will determine the transition to be executed and, consequently, the function that will transform the data. *Right:* The data embedded in the token placed in $p1$ will be iteratively transformed until a stop condition is met.



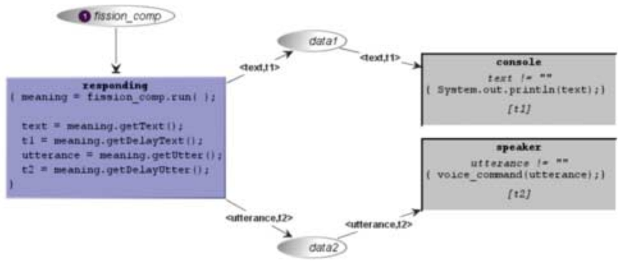

fission_comp
responding { meaning = fission_comp.run( ); text = meaning.getText(); t1 = meaning.getDelayText(); utterance = meaning.getUtter(); t2 = meaning.getDelayUtter(); }
data1
data2
<text,t1>
<utterance,t2>
console text != "" { System.out.println(text); } [t1]
speaker utterance != "" { voice_command(utterance); } [t2]

FIGURE 13. *Left:* After detecting the events provided by three different devices, the fusion engine will be called, decipher user's request and encapsulating it in an object meaning that will keep flowing towards other sections of the net. *Right:* The system can print a text in the console and/or utter a voice command. The output channel(s) chosen will depend on the context and user profile that can be determined inside the method run of the object $fission\_comp$. Note that PetShop allows specifying delay time before firing a transition.

Furthermore, the method putThatThere, executed during the firing of transition notification, is identifying the selected object and updating its position. This method has to be implemented in a textual language.

Unlike flow-based toolkits, the data flow of a PetShop's model can be conditioned by boolean expressions set by the user. This is achieved by using the transitions' prerequisites as shown in Fig. 12. Finally, the fusion and fission of modalities can be naturally modeled by taking advantage of the fact that OPNs offer constructs for synchronization and concurrency. At a high level of abstraction, the fusion and fission of data would have the patterns shown in Fig. 13. The subroutines to be executed at the client side are specified in the annotations within the transitions. Unlike state-based models, the annotations, associated with a transition, can contain programming instructions as seen in Figs. 12 and 13.

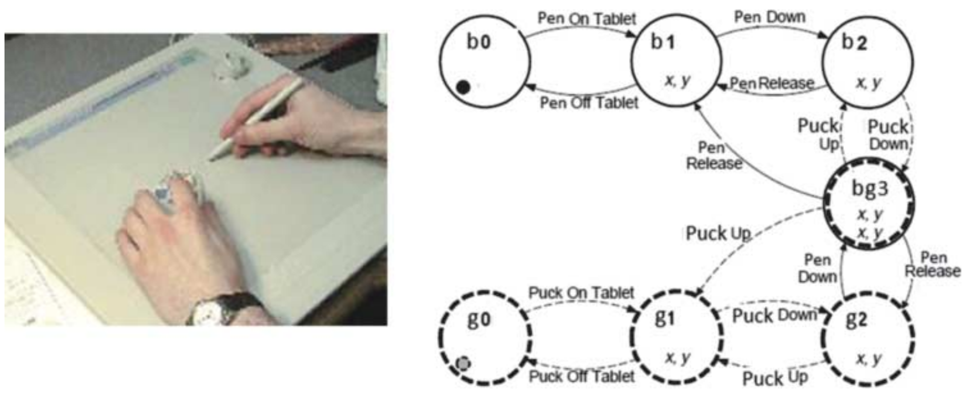


FIGURE 14. *Left:* Two-handed interaction technique on a tablet. *Right:* Solid and dashed arcs represent the events generated by the pencil and puck, respectively. The black token can only be in solid states ($b0$, $b1$, $b2$ or $bg3$) and the gray one, in dashed states ($g0$, $g1$, $g2$ or $bg3$). States $g0$, $g1$ and $g2$ indicate the puck's possible situations: off the tablet, in contact with the tablet or being clicked, respectively. States $b0$, $b1$ and $b2$ have an analogous interpretation for the pen. [*Images from* (Hinckley *et al.*, 1998), *reproduced with permission.*]

8.2. Hinckley's language

A graphical language for modeling systems was proposed by Hinckley *et al.* (1998). Hinckley's language does not have a supporting toolkit because it was specifically created to specify bimanual interaction techniques for whiteboard sessions. Despite this, the existence of a framework that can interpret the semantics of this language will be assumed. In the end, the capabilities of a toolkit depend on (and can be inferred from) the expressiveness of its underlying language, i.e. a framework can only perform those tasks that can be specified in its visual language.

Models created with Hinckley's language are directed graphs composed of circular nodes, tokens and directed arcs. Circular nodes can be solid or dashed. These two types are provided to distinguish the states of each hand device. With regard to arcs, they may also be solid or dashed, thus allowing a visual way to represent the hand that raised an event. Finally, models always contain exactly two tokens -gray and black- whose flow among the graph enables tracking the state of each device.

In order to illustrate the semantics of this language, we will study the specification of an interaction technique described in (Hinckley *et al.*, 1998). The interaction technique makes use of a tablet that is navigated upon with a puck and a pen stylus. The navigation on the tablet is reflected on a map whose sectors can be panned or zoomed depending on the actions performed on the tablet. More precisely, clicking and dragging with the puck only allows panning the map. Pressing down and dragging only the pen draws a free-form shape in the map. Finally, clicking the puck while pressing the pen on the tablet allows zooming the map. The interaction technique is modeled as shown in Fig. 14. Once the pen and the puck come into contact with the tablet, the tokens initially placed in $b0$ and $g0$ will move to $b1$ and $g1$, respectively. In that situation, a click on the puck will move the gray token from its new state $g1$ to $g2$ as indicated by transition Puck Down. Then, if the pen is pressed, the event Pen Down will be triggered causing two changes. First, the gray token will move from state $g2$ to $bg3$. Secondly, the black token will move from state $b1$ to $b2$, and then immediately to state $bg3$ (because the puck's button is still down). Two tokens in state $bg3$ indicate that both devices are being pressed and, consequently, the map can now be zoomed by moving both hands. A framework capable of interpreting the semantics of this diagram could automatically recognize when the user is requesting a zoom on the map, thus releasing programmers from hard-coding the identification of user's request, which is a function of the fusion engine.

It is important to note that the presence of tokens allows a natural and economical representation of the intrinsic parallelism involved in two-handed interaction. Furthermore, the fact that one event may cause two simultaneous transitions (e.g. if the system is in $bg3$, the occurrence of Pen Release will cause two transitions: from $bg3$ to $g2$ and from $bg3$ to $b1$) suggests that it would be possible for a framework to execute simultaneous subroutines if the model would associate one action to each arc as in state-based or Petri net models.

In order to wrap up this section, the scope of the studied toolkits for prototyping a multimodal system will be mentioned.

8.3. Comparing the scopes of token-based toolkits

Since some authors consider two-handed interaction techniques as a special case of multimodal interaction (Roope, 1999), the language proposed by (Hinckley *et al.*, 1998) was included in this comparative study. The main observations regarding the toolkits supporting these languages are: First, the semantics of their underlying languages allow a natural representation of parallel interactions. Secondly, both languages can represent synchronization and concurrency, which are indispensable to implement the fusion and fission components, respectively. Thirdly, the set of possible states and the potential transitions

TABLE 4. Components whose implementation can be facilitated through the use of a token-based toolkit.

	Fusion engine	Dialog manager	Fission component
PetShop	√	√	√
Hinckley	√	√	√

a system may experience are visually identifiable, which facilitates the development of a dialog manager (Table 4).

9. DISCUSSION

Developing a multimodal system is time-consuming and therefore expensive. In order to alleviate this cumbersome situation, many toolkits intended for rapid prototyping of multimodal systems have been created. These toolkits include a framework and a graphical editor that together are able to equip a client application with a multimodal interface. Since the functionalities provided by a framework can only be exploited by means of its visual modeling language, a strong emphasis was placed on explaining the semantics of these languages through detailed descriptions of running examples.

Estimating the reduction of programming workload that can be achieved through the use of a toolkit is not an easy task since toolkits are exclusively described in terms of their low-level technicalities, e.g. implementation details and fancy features of their user interfaces, in the existing literature. The present article defined the scope of a toolkit (Section 3) and its measurement procedure (Section 4) in order to address this challenge.

By measuring and comparing the scope of several graphical toolkits, we observed that they can be clustered into three groups that are called flow-based, state-based and token-based toolkits (Section 5). All these toolkits release their users from incorporating and configuring recognizers in their client applications. Besides this, state-based toolkits also free their users from interpreting user requests and detecting the current state of their prototypes. Finally, token-based toolkits provide the additional gain of preventing their users from programming multiple threads in the client application. As seen in Section 6–8, the visual models of flow-based, state-based and token-based toolkits resemble block diagrams, state diagrams and Petri net graphs, respectively.

The fact that visual languages based on Petri nets exhibited the highest expressiveness among all the studied languages does not mean that the inclusion of Petri nets editors has to be mandatory when designing a new toolkit with such high expressiveness. Any new visual language including constructs for describing composite events, system states and synchronization of concurrent processes may have the same power as Petri nets when it comes to modeling multimodal human–machine interaction.

Table 1 has to be interpreted carefully. The fact that token-based toolkits can facilitate the implementation of a higher number of components does not necessarily mean that they are always the best choice. There is a trade-off between the scope of a toolkit and the semantic simplicity of its visual language; and this brings about that each class of toolkits may be better suited to a specific domain. If the intended multimodal system is not required to convey context-dependent responses, the use of flow-based toolkits may be enough to model its simple behavior. Transformational systems (Harel and Pnueli, 1985) are examples of such systems where the context is irrelevant; they always produce the same output for a given set of inputs. Using state-based or token-based toolkits for specifying these systems will entail the depiction of more complex visual models in return for an unproductive benefit: tracking the state of a stateless system. For prototyping multimodal dialogue systems that convey unimodal output, the use of state-based toolkits may be the most convenient option. GUIs supporting multimodal input are examples of these systems. They almost always respond the user through the visual modality, e.g. by presenting summary reports or pop-up message boxes. The concurrent computation performed on the framework side of a token-based toolkit is not necessary to render unimodal output. Finally, a system supporting not only multimodal input and context-dependent dialogs but also intensive multimodal output (e.g. through avatar animations) can benefit more from token-based toolkits. These can generate and synchronize the multiple output streams involved in the multimodal feedback provided by these systems.

The aforementioned correlation between the class of a toolkit and its application domain must be taken into account before choosing/designing a toolkit. In order to avoid choosing a toolkit that is unnecessarily complicated, users must consider the characteristics of the systems they want to prototype. Similarly, software developers must take into account the needs of their target users before creating a new toolkit.

This work has identified the extent to which toolkits can reduce the amount of programming code involved in the creation of multimodal prototypes (Table 1), but the ease of use of the toolkits has not been extensively commented. Supplementary studies may explore in this direction by evaluating additional features of the toolkits such as the configuration effort required to start a new project (e.g. need for speech or gesture grammars), the extent of user assistance (e.g. software wizards, contextual help), the necessary steps to extend their predefined set of recognizers (e.g. configuration file), the variety of their debugging tools (e.g. log files, system dump, interactive debugging), the type of exceptions they can cope with (e.g. compilation, runtime errors), their range of detectable events (e.g. input hardware events, client application custom events), the learning curve of their visual languages, the number of applications they can manage (e.g. one or multiple, distributed client applications), the type of client application they target (e.g. stand-alone, web, mobile), etcetera.

For over ten years researchers have been proposing new toolkits and suggesting that these newer ones are more effective because they exhibit some features that the older ones do not have. We think these innuendos are misleading and obstruct the way towards better toolkits: the more effective toolkit is the one that minimizes the programming effort at the client side because this will lead to faster creation of prototypes. This is the ultimate goal of a toolkit.

Similarly to (Dumas *et al.*, 2013), we consider that determining the extent to which the combination of a framework and a visual language can represent multimodal dialogues is a significant issue. We strongly believe that this paper has been a major step in tackling this issue.

10. CONCLUSIONS

This article has surveyed and compared a series of graphical toolkits for rapid prototyping of multimodal systems. The novelty of this work is the proposal of an indicator, herein called scope, which helps one to estimate how well an arbitrary toolkit accomplishes its goal of shortening the prototyping phase. Since multimodal systems share common functionalities, we measured the scope of a toolkit by determining which of these functionalities are incorporated in its framework so that they can be invoked through the depiction of visual models and without the need of programming code.

When observing their technical aspects, the studied toolkits seem rather different from one another. However, when comparing their scope, they can only be classified into three groups, called flow-based, state-based or token-based toolkits. The frameworks of the toolkits within each group can perform similar tasks, which do not have to be programmed by their users anymore.

FUNDING

This research was supported by the Special Research Fund (BOF) of Hasselt University.

REFERENCES

Ait-Ameur, Y. and Kamel, N. (2004) A Generic Formal Specification of Fusion of Modalities in a Multimodal HCI, pp. 415–420. IFIP World Computer Science, Kluwer Academic Publishers.

Beaudouin-Lafon, M. and Mackay, W. (2003) Prototyping Tools and Techniques. The Human–Computer Interaction Handbook, pp. 1006–1031. ACM.

Bolt, R. (1980) Put-That-There: Voice and Gesture at the Graphics Interface. Proc. 7th Annual Conference on Computer Graphics and Interactive Techniques, pp. 262–270. ACM.

Bourguet, M. (2002) A Toolkit for Creating and Testing Multimodal Interface Designs. Posters and Demos from the 15th Annual ACM Symposium on User Interface Software and Technology, pp. 29–30. ACM.

Bourguet, M. (2003) Designing and Prototyping Multimodal Commands. Human–Computer Interaction INTERACT'03, pp. 717–720. IOS Press.

Bui, T.H. (2008) Multimodal Dialogue Management—State of the Art. CTIT Technical Report Series, No. 06-01, University of Twente (UT), Enschede, The Netherlands.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R. and Stafford, J. Documenting Software Architectures: Views and Beyond (2nd edn). Addison–Wesley.

Dargie, W., Strunck, A., Winkler, M., Mrhos, B., Thakar, S. and Enkelmann, W. (2007) A Model-Based Approach for Developing Adaptive Multimodal Interactive Systems. 2nd International Conference on Software and Data Technologies. ICSOFT.

De Boeck, J., Vanacken, D., Raymaekers, C. and Coninx, K. (2007) High level modeling of multimodal interaction techniques using NiMMiT. J. Virtual Real. Broadcast., 4.

De Boeck, J., Raymaekers, C. and Coninx, K. (2009) CoGenIVE: Building 3D Virtual Environments Using a Model Based User Interface Design Approach. Computer Vision and Computer Graphics: Theory and Applications, Springer, ISBN 978-3-642-10225-7, ISSN 1865-0929.

Dragicevic, P. and Fekete, J. (2002) ICON: Input Device Selection and Interaction Configuration, Demonstration. UIST 2002 Companion, pp. 47–48.

Dragicevic, P. and Fekete, J. (2004) Support for Input Adaptability in the ICON Toolkit. Proc. 6th Int. Conf. on Multimodal Interfaces (ICMI04), State College, PA, USA, 212–219.

Dumas, B., Lalanne, D. and Ingold, R. (2009) HephaisTK: A Toolkit for Rapid Prototyping of Multimodal Interfaces. Proc. Int. Conf. on Multimodal Interfaces and Workshop on Machine Learning for Multi-modal Interaction (ICMI-MLMI 2009), pp. 231–232. ACM.

Dumas, B. (2010) Frameworks, description languages and fusion engines for multimodal interactive systems. PhD Thesis, University of Fribourg.

Dumas, B., Lalanne, D. and Ingold, R. (2010) Description languages for multimodal interaction: a set of guidelines and its illustration with SMUIML. J. Multimodal User Interfaces 3, 237–247.

Dumas, B., Signer, B. and Lalanne, D. (2013) A Graphical UIDL Editor for Multimodal Interaction Design Based on SMUIML. Science of Computer Programming.

Engel, R. and Schutt, R. (2013) The Practice of Research in Social Work (3rd edn), Chapter 4. SAGE Publications, Inc.

Flippo, F., Krebs, A. and Marsic, I. (2003) A Framework for Rapid Development of Multimodal Interfaces. Proc.f ICMI 2003, Vancouver, BC, November 5–7, pp. 109–116.

Gibbon, D., Mertins, I. and Moore, R. (2000), Handbook of Multimodal and Spoken Dialogue Systems. The Springer International Series in Engineering and Computer Science.

Harel, D. and Pnueli, A. (1985) On the Development of Reactive Systems. In Apt, K.R. (ed.) Logics and Models of Concurrent Systems, NATO ASI Series, Vol. F-13, pp. 477–498. Springer, New York.

Hinckley, K., Czerwinsky, M. and Sinclair, M. (1998) Interaction and Modeling Techniques for Desktop Two–Handed Input. Proc. ACM UIST98, pp. 49–58. ACM.

Jensen, K., Kristensen, L. and Wells, L. (2007) Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf., 9, 213–254.

Johnston, M., Bangalore, S., Vasireddy, G., Stent, A, Ehlen, P., Walker, M., Whittaker, S. and Maloor, P. (2002) MATCH: An Architecture for Multimodal Dialogue Systems. Proc. 40th Annual Meeting of the Association for Computational Linguistics, pp. 376–383. Association for Computational Linguistics.

Kaltenbrunner, M., Bovermann, T. Bencina, R. and Constanza, E. (2005) TUIO—a Protocol for Table-Top Tangible User Interfaces. Proc. 6th Int. Workshop on Gesture in Human–Computer Interaction and Simulation. Springer.

Lakos, C. (1991) Language for Object-Oriented Petri Nets. Department of Computer Science, University of Tasmania.

Lalanne, D., Nigay, L., Palanque, P., Robinson, P., Vanderdonckt, J. and Ladry, J. (2009) Fusion Engines for Multimodal Input: A Survey, ICMI–MLNI09. Proc. 2009 Int. Conf. on Multimodal Interfaces, pp. 153–160. ACM.

Lawson, J.-Y., Vanderdonckt, J. and Macq, B. (2008) Rapid Prototyping of Multimodal Interactive Applications Based on Off-The-Shelf Heterogeneous Components. Adjunct Proc. 21st Annual ACM Symposium on User Interface Software and Technology, UIST08, pp. 41–42. ACM, New York, NY.

Lawson, L., Al–Akkad, A., Vanderdonckt, J. and Macq, B (2009) An Open Source Workbench for Prototyping Multimodal Interactions Based on Off-the-Shelf Heterogeneous Components. Proc. 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS09. ACM.

Len, B., Clements, P. and Kazman, R. Software Architecture in Practice (3rd edn). Addison-Wesley.

Luyten, K., Clerckx, T., Coninx, K. and Vanderdonckt, J. Derivation of a Dialog Model from a Task Model by Activity Chain Extraction. Proc. of DSV-IS 2003, Lecture Notes in Computer Science, Vol. 2844, pp. 191–205. Springer.

Mitra, S. and Acharya, T. (2007) Gesture recognition: a survey. IEEE Trans. Syst. Man Cybern.-Part C: Appl. Rev., 37.

Monwar, M. and Gavrilova, M. (2011) Markov Chain Model for Multimodal Biometric Rank Fusion. Signal, Image and Video Processing. Springer.

Murata, T. (1989) Petri Nets: Properties, Analysis And Applications. Proceedings of the IEEE, Vol. 77.

Navarre, D., Palanque, P., Dragicevic, P. and Bastide, R. (2006) An approach integrating two complementary model-based environments for the construction of multimodal interactive applications. J. Interact. Comput., 18, 910–941.

Navarre, D., Palanque, P., Ladry, J. and Barboni, E. (2009) ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. ACM Trans. Comput.–Hum. Interact., 16, 1–56.

Neal, J., Thielman, C., Dobes, Z., Haller, S. and Shapiro, S. (1989) Natural Language with Integrated Deictic and Graphical Gestures. Proc. Workshop on Speech and Natural Language, pp. 410–423. Association for Computational Linguistics.

Nise, N. (2011) Control Systems Engineering (6th edn). Wiley E-text, ISBN 978-1-1180-0618-4.

Oviatt, S. (1999). Ten Myths of Multimodal Interaction. Commun. ACM, 42. ACM Press, New York, pp. 74–81.

Oviatt, S. (2003). Multimodal Interfaces. The Human–Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications. Lawrence Erlbaum Assoc., Mahwah, NJ.

Palanque, P. and Schyn, A. (2003) A Model-Based Approach for Engineering Multimodal Interactive Systems. INTERACT 2003.

Pineux, A. (2012) Runtime models extension for SKEMMI, a component–based graphical authoring tool for multimodal interactions. MSc. Thesis, Ecole Polytechnique de Louvain.

Rabiner, L. (1989) A tutorial on hidden Markov models and selected applications in speech recognition. Proc. IEEE, 77, 257–285.

Rajman, M., Bui, T.H., Rajman, A., Seydoux, F. and Trutnev, A. (2004) Assessing the Usability of a Dialog Management System Designed in the Framework of a Rapid Dialogue Prototyping Methodology. Acta acustica united with acustica, 90, 1096–1111.

Roope, R. (1999) Multimodal Human–Computer Interaction: a constructive and empirical study. PhD Thesis, University of Tampere, Finland.

Samek, M. (2009) A crash course in UML state machines. Embedded.com.

Sharma, R., Pavlovic, V. and Huang, T. (1998) Toward multimodal human–computer interface. Proc. IEEE, 86, 853–860.

Stevens, S. (1946) On the theory of scales of measurement. Science 7, 103, 677–680.

Sushmita, M. (2007) Gesture Recognition: A Survey. IEEE Trans. Syst. Man Cybern.-Part C: Appl. Rev., 37.

Sy, O., Navarre, D., Le, D., Palanque, P. and Bastide, R. (1999) Formal definition of Interactive Cooperative Objects. ESPRIT Reactive LTR 24963 Project, L.I.H.S., University of Toulouse.

Trung, B. (2008) Toward affective dialogue management using partially observable Markov decision processes. PhD Thesis, University of Twente.

Vanacken, L. (2009) Multimodal Selection in virtual environments: enhancing the user experience and facilitating development. PhD Thesis, UHasselt Diepenbeek.

Vanacken, D. (2012) Touch-based interaction and collaboration in walk-up-and-use and multi–user environments, Universiteit Hasselt, PhD thesis, UHasselt Diepenbeek.

Vanacken, D., De Boeck, J., Raymaekers, C. and Coninx, K. (2006) NiMMiT: A Notation for Modeling Multimodal Interaction Techniques. Proc. Int. Conf. on Computer Graphics Theory and Application (GRAPP06).

Vo, M. and Wood, C. (1996) Building an Application Framework for Speech and Pen Input Integration in Multimodal Learning Interfaces. Int. Conf. on Acoustics, Speech and Signal Processing.

Wahlster, W., Reithinger, N. and Blocher, A. (2001) SmartKom: Multimodal Communication with a Life–Like Character. 7th European Conf. on Speech Communication and Technology. ISCA.

Wagner, F., Schmuki, R., Wagner, T. and Wolstenholme, P. (2006) Modeling Software with Finite State Machines. A Practical Approach, Auerbach Publications.

Werner, K., Raedle, R. and Harald, R. (2010) Interactive design of multimodal user interfaces—reducing technical and visual complexity. J. Multimodal User Interfaces, 3, 197–213.