An Intelligent Tool for Sketching USIXML User Interfaces

Adrien Coyette, Stéphane Faulkner, Manuel Kolp, Quentin Limbourg, Jean Vanderdonckt
Université Catholique de Louvain, School of Management (IAG)
Place des Doyens, 1 – B-1348 Louvain-la-Neuve (Belgium)
{coyette, faulkner, kolp, limbourg, vanderdonckt}@isys.ucl.ac.be – www.isys.ucl.ac.be/staff
Phone: +32-1047 {8379,8990, 8395, 8384, 8525} – Fax: +32-10478324

ABSTRACT
During these last years, many researchers have proposed new alternatives for early interface design based on hand-sketch. But these new alternatives seem to be dedicated to obsolescence as they only offer the possibility to generate user interfaces for a single platform in a unique language. Indeed, in a context where the number of computing-platforms and system environments is exploding, new alternatives should be considered. This paper presents an innovating alternative with SketchiXML, a multi-agent application able to handle several kinds of hand-drawn sources as input, and to provide the corresponding specification in USIXML (USer Interface eXtensible Markup Language), a platform-independent user interface description language.

ACM Classification Keywords
D.2.1 **[Software Engineering]:** Requirements/Specifications – *elicitation methods (e.g., rapid prototyping, interviews, JAD)*. D.2.2 **[Software Engineering]:** Design Tools and Techniques – *user interfaces***.** H.5.2 **[Information Interfaces and Presentation]:** User Interfaces – *Multi-agents, Prototyping, Graphical User Interfaces (GUI)*. I.3.6 [**Computer Graphics**]: Methodology and Techniques – *interaction techniques*.

General terms: Design, Languages, Human Factors.

Author Keywords
Development processes, multi-platform, multi-path development, user interface description language, multi-agent architecture, BDI, SKwyRL, interface sketching, user interface engineering.

INTRODUCTION
Most interfaces designers consider hand-sketch on paper as the most effective way to represent the first drafts of the future interfaces. Indeed, this kind of unconstrained approach is fast and easy and permits the designer to focus on basic structural issues instead of unimportant details. But computer assisted interfaces design also offer a range of advantages such as the possibility of easily erasing or moving components. This perspective was at the origin of huge efforts during the last decade, where numerous of computer design environment came on the scene, with famous software like Borland JBuilder, Microsoft Visual Basic and others. However, these elements-approach based software did not generate the saving of time expected during the early design; designers have reported that clients or even other designers tend to focus on details such as color, exact alignment or typography when using high fidelity mocks-up [11]. In response to the uncovered gap between these two approaches, many researches were carried out in order to propose alternatives based on a hybrid approach, taking the best of the hand-sketching and of computer assisted interfaces design. Two major orientations have appeared among all the computer-sketch tool considered, one orientation considers the design process as a creative process that should not be interrupted, and thus only offer to the user to sketch the interfaces and the scenarios [2,15]. The second orientation couples the design process with an interpretation of the interfaces sketched in a programming language [4,20]. The two approaches will be discussed in the next section, and on basis of the analysis of the different design tools, we will propose an extension to overcome some drawbacks of the second orientation.

This paper will present the agent-architecture used to design SketchiXML, a new kind of application for early interface design based on hand-sketch drawing. SketchiXML is different from others sketching applications as it provides more than user interfaces (UIs) in a specific programming language; it provides the specification of the interface in USIXML (www.usixml.org) [17, 19], a platform-independent UI Description Language (UIDL). Moreover, SketchiXML will also assist the developer during the design process in a flexible way. The designer will have the opportunity to define how the different experts composing the application must participate in the design process. As an example, the user may request that the interfaces critiquing experts provide real time advice on all the issues encountered, or just on the major issue.

These requirements fit very well the agent oriented paradigm. Indeed multi-agent architectures appear to be more flexible, modular and robust than traditional, including object-oriented ones. Multi-agent architectures represent dynamic and evolving structures and components which can change at run time to benefit from new knowledge or components [14].

The structure of the paper will be as follow: the two next sections establish the research context with an introduction to the related works of the different domain linked to the application, and with an illustrative scenario of SketchiXML. Section 3 proceeds to an introduction to the SKwyRL framework (Socio-Intentional ArChitecture for Knowledge Systems and Requirements Elicitation (http:// www.isys.ucl.ac.be/skwyrl)) [7], which is dedicated to the specification of BDI multi-agent systems. Section 4 introduces USIXML, a language allowing designers to apply a multi-path development of UIs. Section 5 presents the multi-agent architecture of SketchiXML and its formal specifications with the SKwyRL-framework. The last section concludes and proposes some ideas for future extensions.

SketchiXML will be open source, and will be available for download on the USIXML web site as soon as ready to be shared.

RELATED WORK

To uniformly present the solution usually considered for early interface design, this section gives an overview of the main alternatives currently used for prototyping.

The paper and pencil approach or the whiteboard/blackboard and post-its approach are often considered as the most effective way to prototype the future interfaces. The advantages of these approaches find roots in the fact that it is easy to have access to all the components, and that the designer mainly focus on the main issue of the design rather than on detail.

A second approach is based on the use of drafting tools such as Macromedia Director or Microsoft Visio. These tools allow the designer to build quick prototypes of the future interface using a graphical tool. The result of the process with this kind of tool is a medium-fidelity mock-up that cannot be directly used for the code generation. Moreover, the use of medium-fidelity prototype may cause the designer to spend too much time on superficial details while these details are not yet needed.

A third approach, closely related to the drafting tools are the graphical interfaces builders such as Visual Basic, Borland JBuilder, etc. These tools allow the designer to build graphically the final UI in a determined programming language. Obviously, this approach suffers from the same problem as the drafting tool in a stronger way, since this kind of tool produce high-fidelity mock-ups. But these kinds of tool are very useful for the interface implementation phase once the early design is completed.

Other tools, in the same line as the two preceding approaches, are the "what you see is what you get" (WYSIWYG) web authoring tools such as Microsoft FrontPage or Macromedia StudioMX. These tools offer the same functions than the graphical interface builders, but they are dedicated to people without specific knowledge of programming language. The underlying concept of WYSIWYG used by these kinds of applications, naturally lead the designer to spend more time on details than on the core issues.

As explained in the introduction, several alternatives were produced in response to the uncovered designer expectancies in the early UIs design domain. Two major trends appeared from these new alternatives, on one hand applications that just provide a framework for interface sketching, and on the other hand applications that couple the features of the first ones with shapes recognition and interpretation.

The major tools for interfaces prototyping based on hand-sketch without shapes recognition are DENIM [15] and DEMAIS [2]. DENIM is a sketch-based web site design application for early stage of design. It allows sketching the web pages, to create the links between the pages with the use of a storyboard, and to see the interaction in practice thanks to a run mode. DEMAIS is also a hand-sketch based web site design for early stage of design, and offers almost the same features. The major difference between these tools is graphical presentation of the dialogue. DENIM works on a single plane, while DEMAIS uses the concept of layers. A first layer contains all the widgets sketched, a second layer contains annotations, and a third layer contains a set of sketch describing the temporal and interactive behavior. As is the case with DENIM, the interaction can be visualized thanks to a run mode.

JavaSketchIt [4] and Freeform [20] are the two major applications for interface design based on hand-sketch recognition. JavaSketchIt proceeds in a slightly different way than Freeform, as it recognizes the shapes drawn by the user in real time, and generates a Java UI as output. Freeform only recognizes the shapes once the design of the whole interface is completed, and produces Visual Basic 6 UIs.

To identify differences between the tools evoked above, we present with Fig. 1 a summary as a cross table where all the applications are evaluated on basis of nine attributes. Some results in the table may appear surprising as the applications are only evaluated for the early design phase. The attributes considered are the following:

- The *Language neutrality* attributes represents to what extend the tool is associated with a specific language.
- The *Development time* represent the time needed to build a first draft of the interface with this tool.

- The *Precision* attribute represents the accuracy of the output produced by the considered tool.
- The *Pre-requisite knowledge* attribute depicts the expertise needed by the user of the tool.
- The *Scenario* attribute illustrates the fact that the tool can handle scenarios or storyboards.
- The *Presentation* attribute represents the graphical coverage of the tool in terms of numbers of widgets that can be represented.
- The *Dialogue* attribute represents the ability of the tools to describe the navigational concept.
- The *Representativeness* attribute represents the fact that the interface represented with the tool is close to its representation in a programming language
- The *Compatibility* attribute focuses on the naturalness of the interface construction with the tool.

	Language Neutrality	Development time	Precision	Pre-requisite knowledge	Scenario	Presentation	Dialogue	Representativeness	Compatibility
Paper & Pencils	++	+/-	+/-	++	+	+	+	-	+
MacroMedia Director	+	-	+	+/-	-	++	-	+	-
Microsoft Visio	+	-	+	+/-	-	++	-	+	-
Visual Basic	--	--	++	+/-	-	++	+/-	++	--
Borland JBuilder	--	--	++	+/-	-	++	-	++	--
Microsoft FrontPage	+/-	--	++	+	-	++	+/-	++	--
Macromedia StudioMX	+/-	--	++	+	-	++	+/-	++	--
DENIM	++	++	+/-	+	++	+	++	--	++
DEMAIS	++	++	+/-	+	++	+	++	--	++
JavaSketchIt	--	+	+/-	+	-	+/-	--	+/-	+/-
Freeform 2	--	+	+/-	+	-	+/-	--	+/-	+/-

Figure 1. Summary of the tools' characteristics.

The scope of SketchiXML will be, on one hand, to combine in a flexible way, the advantages of tools such as DENIM or DEMAIS with the advantages of tools such as JavaSketchIt [4]. On the other hand, SketchiXML will integrate new features such as interface critiquing, computer-aided generation of specifications, code generation for multiple computing platforms, multi-source of input.

Given that SketchiXML will assist the designer during the design process with usability advice, we will briefly introduce some relevant related work in the domain of interfaces critiquing tools. Ergoval [6] appears to be one of the most interesting works in that area. It allows to automatically evaluating the usability of any UI under the windows environment, regardless of the development tool used or the stage of development cycle.

A second interesting tool related to our application is SHERLOCK [18]. It is a set of tools aimed at checking the visual and textual consistency of Graphical User Interface (GUI). SHERLOCK provides terminology analysis tools including an Interface Concordance, an Interface Spellchecker, and Terminology Baskets to check for inconsistent use of familiar groups of terms.

The last part of this section will focus on the related work in the domain of multi-agent systems applied to shape recognition, and interface building. Even if the literature in that domain appears to be limited, we have found a very attractive approach in EsQUIsE [16]. Its scope is slightly different than in SketchiXML since EsQUIsE is aimed at supporting collaborative work for architectural representation of a building.

EsQUIsE captures and interprets an architectural sketch in real time, and chronologically constructs the architectural representation of the building. Thus, while the designer freely sketches his project on a digital tablet, EsQUIsE first captures and synthesizes each stroke. Even if the post-treatment will obviously be different, the main concept will be roughly the same as SketchiXML; a multi-agent system will extract character strings, words and some symbols recognition which are translated to captions and icons. Following the recognition process, the system will compose the closed graphic borders which will be associated and interpreted as functional spaces in the architectural representation. EsQUIsE can then give the geometrical model and the topologic diagram of the design, as needed by basic evaluators and classical tools of architectural production.

SCENARIO

In order to give a better understanding of the application, we will present SketchiXML with a small case study based on the design of a real estate web site. Once the future system functionalities are defined, the designer will proceed to the early prototyping of the future UIs with the customer. At this level, the designer is just willing to obtain a global view of the UIs and does not want to spend time on unimportant details.

In that situation, SketchiXML appears to be very appropriate as it permits to sketch the UIs as easily as on paper, but also offers the possibility to generate usability advices and interface specifications during or at the end of the process. So, the first step for the designer using USIXML will consist in providing all the parameters to be used by the application.

Fig. 2 depicts a screenshot of the settings interface where the designer chooses the level of system support for each agent, ranging from fully automated to fully manual, the middle being computer-aided. For instance, Fig. 2 depicts a situation where the designer does not want to be interrupted during the design phase. So recognition, usability advice and USIXML generation are all set on manual and output quality is set on the minimum. This type of configuration is thus appropriated when the designer wants to have a quick

result and does not want to waste time. The sketching phase in that situation will be very similar to the sketching process of application such as DENIM or DEMAIS. Of course, the designer is always allowed to enable a feature while the process is running, or to execute it manually. For instance, the designer starts to sketch the future "search properties" interface, with all the features disabled.

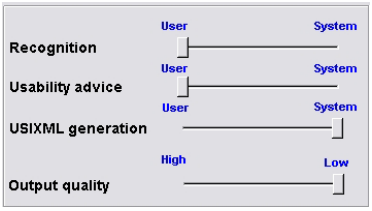


Figure 2. Settings interface.

As the process advances, the future interface becomes more complex, and the designer may decide to set the shapes recognition and usability advice on automatic mode. SketchiXML will then analyze the full interface, and provide real time recognition and interface critiquing. Fig. 3 gives an illustration of the early design of the "search properties" interface with the actual version of JavaSketchIt [4]. On basis of the shapes recognitions and interpretation, the interface critiquing expert will express usability advices if required. For instance, in Fig. 3, the user is advised to center the left button, and to group the widgets present in the interface in a container.

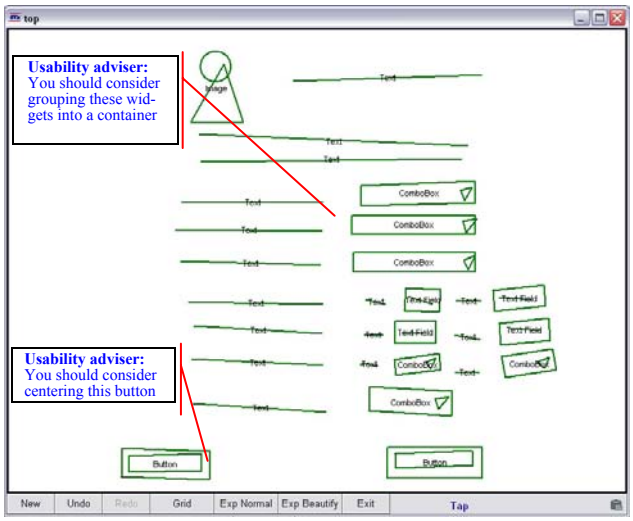


Figure 3. Sketch of the "search properties" interface with JavaSketchIt [4].

Then, once the designer considers that the interface prototype is good enough, the components layout can be converted in USIXML if no ambiguities are met. Otherwise, the system will consider the parameters entered for the process in order to evaluate how to solve the ambiguities. For instance, in Fig. 2 observe a situation where the designer just wants low fidelity specification of the interface. So, if the system faces ambiguities, it will just try to disambiguate itself with the help of its disambiguation algorithms. If the output quality value was set on high instead of low then the system would firstly try to disambiguate the situation. If it considers that the degree of certainty attached to the widgets was not sufficient, it would ask to the designer to solve the unsolved ambiguities, with the graphical editor. Fig. 4 gives the USIXML specifications corresponding to the interface prototyped on Fig. 3.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<cuiModel creationDate="2004-07-14T21:52:43.155-08:00"
name="immo " schemaVersion="1.4.3" id="immo__14"
xsi:schemaLocation="http://www.usixml.org/spec usiXML-cui.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.usiXML.org">
<version modifDate="2004-07-14T22:08:33.191-08:00"
xmlns="">1</version>
<authorName xmlns="">Adrien</authorName>
<comment xmlns="">Generated by SketchiXML </comment>
<window isResizable="false" windowTopMargin="0" windowLeft-
Margin="0" isAlwaysOnTop="false" height="588" width="713"
bgColor="#e0dfe3" isEnabled="true" isVisible="true"
fgColor="#000000" borderWidth="0" name="window_0"
id="window_0">
<box relativeMinWidth="0" relativeWidth="0" isFill="false" relative-
Height="0" isResizableHorizontal="false" type="horizontal" isScrol-
lable="false" isDetachable="false" isSplitable="false" isResiz-
ableVertical="false" relativeMinHeight="0" isBalanced="false" is-
Flow="false" height="588" width="713" isEnabled="true" isVisi-
ble="false" name="box_0" id="box_0">
<imageComponent isEnabled="true" isVisible="true"
name="image_0" id="image_0"/>
<textComponent textMargin="0" isItalic="false" isBold="true"
textFont="Dialog" textColor="#000000" visitedLinkColor="#000000"
isSuperscript="false" isSubscript="false" textSize="12" textVerti-
calAlign="middle" isPreformatted="false" isUnderline="false"
isStrikethrough="false" activeLinkColor="#000000" textHorizontalA-
lign="left" bgColor="#e0dfe3" isEnabled="true" isVisible="true"
fgColor="#000000" name="label_3" id="label_3"/>
<comboBox isDropDown="false" isEditable="false" bgColor="#ffffff"
isEnabled="true" isVisible="true" fgColor="#000000"
name="combobox_0" id="combobox_0"/>
        […]
<button bgColor="#e0dfe3" isEnabled="true" isVisible="true"
fgColor="#000000" name="button_0" id="button_0"/>
      </box>
   </window>
</cuiModel>
```

Figure 4. USIXML specifications of "search properties".

The designer will then have the possibility to import the USIXML specifications generated from the first draft in GrafiXML [19]. The main idea behind this progression is that a UI is rarely designed perfectly from the beginning. Rather, it progressively evolves from a rough general idea to a more precise layout as the development life cycle is evolving. GrafiXML is a

USIXML editor based on a classical elements-based approach. So, once the designer has completed the first phase of early design with the customer, he can thus directly import the specification and define all the detail that cannot be defined during this first phase. Fig. 4 gives an illustration of the "search properties" interface specification imported in GrafiXML. When the specifications obtained from SketchiXML are refined, the designer will have the option to generate graphical UI in several programming language. Several interpreters currently exist such as FlashiXML or Tcl-Tk USIXML, others are in ongoing development.

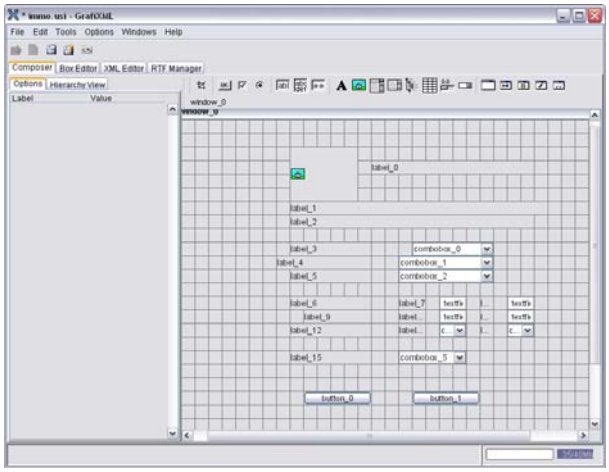


Figure 4. Import of the specification in GrafiXML.

WIDGETS CATALOGUE

As SketchiXML has to cope with widgets representations in different formats, it is important to define a clear mapping between these representations. The following section will briefly present the correspondence catalogue between the representations. Each widget is described with a literal description, a graphical representation, one or several sketching propositions and one or more examples of vectorial representation. The graphical representation was chosen on basis of the different representations present in the common elements-based approach environments. The choice of the widget sketching representation was made according to two major constraints; firstly the proposition had to be as natural as possible. For instance it seems obvious to us that a button with a cross on it, is less natural to depict a validate button than a button with a V, represented on Fig. 6. Secondly, the sketching alternatives had to use the minimum amount of different shapes, while keeping the sketching alternative sufficiently different, in order to avoid confusion.

Button	*Description*	This widget allows the user to trigger an action (of any kind).
	Graphical representation	Button

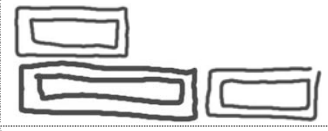
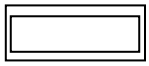
	Sketching proposition	
	Vectorial representation	

Figure 5. Representation of a button in different modes.

As some widgets or groups of widgets appear very frequently when designing GUIs, we have extended the catalogue of widgets to cover these frequent needs. For instance figure 5 depicts a simple button, but on basis of this button we have defined the widget of figure 6 that is a validate button. The main principles used in this case is to always proceed incrementally, such as if the validate button is not recognized, the simple button is likely to be.

Validate Button	*Description*	This widget is an incremented version of the button widget. It allows the user to validate a form.
	Graphical representation	Validate
	Sketching proposition	
	Vectorial representation	

Figure 6. Representation of a pushbutton in different modes.

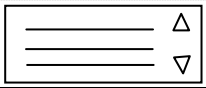
ListBox	*Description*	This widget allows the user to make a single or multiple selections within a list.
	Graphical representation	Item 1 Item 2 Item 3 Item 4
	Sketching proposition	
	Vectorial representation	

Figure 7. Representation of a list box in different modes.

THE SKWYRL-FRAMEWORK

We describe and specify the architecture of SketchiXML using the SKwyRL framework [7]. This framework is aimed to help to design BDI multi-agent system architectures. It is based on a specific agent Ar-

chitectural Description Language (ADL), called SKwyRL-ADL [8], and a catalogue of re-use organizational styles structuring the agent interactions [14]. The rest of this section introduces the key main concepts of multi-agent systems and presents the SKwyRL Framework.

Multi-Agent Systems and BDI Model

An *agent* defines a system entity, situated in some environment, that is capable of flexible and autonomous action in order to meet its design objective [14]. An agent can be useful as a stand-alone entity that delegates particular tasks on behalf of a user. However, in the overwhelming majority of cases, agents exist in an environment that contains other agents. Such environment is a *multi-agent system* that can be defined as a social organization composed of agents that interact with each other to achieve common or private goals [14]. In order to reason about themselves and act in an autonomous way, agents are usually built on rationale models and reasoning strategies that have roots in various disciplines including artificial intelligence, cognitive science, psychology or philosophy. An exhaustive evaluation of these models would be out of the scope of this paper or even this research work. A simple yet powerful and mature model coming from cognitive science and philosophy that has received a great deal of attention, notably in artificial intelligence, is the *Belief-Desire-Intention* (BDI) model [13]. This approach has been extensively used to study the design of rationale agents and is proposed as a keystone model in numerous agent-oriented development environments such as JACK [12] or JADEX [13]. The main concepts of the BDI agent model are (in addition to the notion of agent itself):

- *Beliefs* that represent the informational state of a BDI agent, i.e. what it knows about itself and the world;
- *Desires (or goals)* that are its motivational state, that is, what the agent is trying to achieve;
- *Intentions* that represent the deliberative state of the agent, that is, which plans the agent has chosen for possible execution.

SKwyRL-ADL

SKwyRL-ADL proposes a set of abstractions that are fundamental to the description and specification of agent architectures based on the BDI (Belief-Desire-Intention) model. An ADL provide a concrete syntax for specifying architectural abstractions in a descriptive notation [21]. SKwyRL-ADL is composed of two sub-models which operate at two different levels of abstraction: behavioral and structural. The *behavioral model* captures the informational and motivational state of the agent and its intentional behavior. The *structural model* captures the primitive entities that support the construction of configurations. That is, they represent the element that are "instantiated" to form an architecture. In the following of this section, we briefly present both models.

Behavioral Model

Fig. 8 illustrates the main entities and relationships of the behavioral model. The agent needs knowledge about the environment in order to reach decisions. Knowledge is contained in agents in the form of one of many *knowledge bases*. A knowledge base consists of a set of *beliefs* that the agent has about the environment and a set of *goals* that it pursues. A belief is a finite set of objects, things with individual *identities* and *properties that* represent a view of the current environment states of an agent. However, beliefs about the current state of the environment are not always enough to decide what to do.

In other words, as well as a current state description, the agent needs some goal information, which describes an environment state that is (not) desirable.

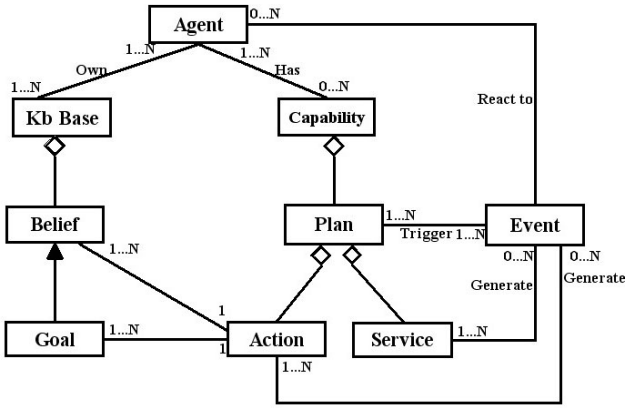


Figure 8. Conceptual representation of Behavioral Model.

The intentional behavior of an agent is represented by its *capabilities* to react to *events*. An event is generated either by an *action* that modifies beliefs or adds new goals, or by service*s* provided from another agent. Note that these services are represented in the global model because they involve interaction among agents that compose the agent system. An event may invoke (trigger) one or more *plans*; the agent commits to execute one of them, that is, it becomes intention. A plan defines the sequence of action to be chosen by the agent to accomplish a task or achieve a goal. An action can query or change the beliefs, generate new events or submit new goals.

Structural Model

Fig. 9 conceptualizes the structural model which describes the interaction among agents that compose the system. *Configurations* are the central concept of architectural design, consisting of an interconnected set of

agents. The topology of a configuration is defined by a set of bindings between provided and required services.

An agent interacts with its environment through an interface composed of *sensors* and *effectors*. An effector provides to the environment a set of services. Then, a sensor requires a set of services from the environment. A service is an action involving an interaction among agents.

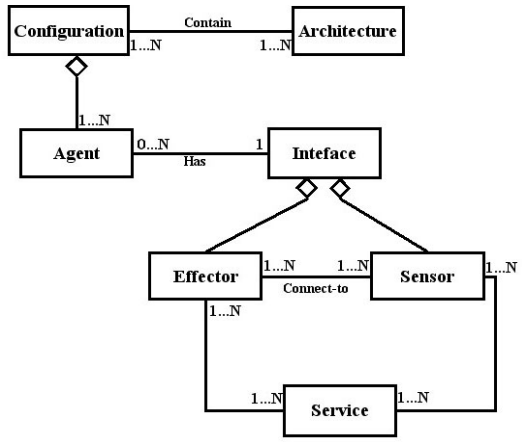


Figure 9. Conceptual Representation of the Global Model.

The whole agent system is specified with an *architecture* which contains a set of configurations. An architecture represents the whole system by one or more detailed configuration descriptions.

Organizational Styles

Architectural styles are intellectually manageable abstractions of system structure that describe how system components interact and work together. We have defined multi-agent systems as *social organizations* composed of autonomous and proactive agents that cooperate with each other to achieve common or private goals. A key aspect to conduct architectural design in SKwyRL is the specification and use of organizational styles (e.g., [7,14]). These are socially-based design alternatives inspired by models and concepts from organizational theories that analyze the structure and design of real-world human organizations.

For instance, the SketchiXML architecture has been designed following and adapting the joint-venture organizational style detailed in [7]. In a few words, the joint-venture organizational style is a meta-structure that defines an organizational system that involves agreement between two or more partners to obtain mutual advantages (greater scale, a partial investment and to lower maintenance costs…).

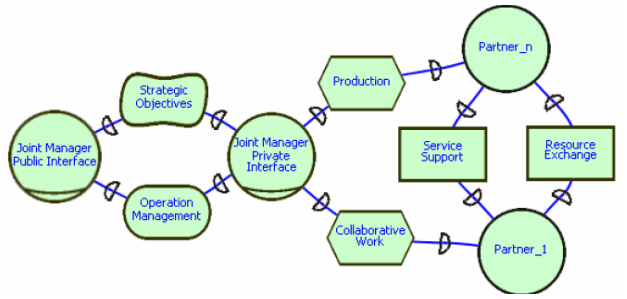


Figure 10: *i representation of the Joint Venture organizational style.**

Fig. 10 models the joint-venture organizational style using *i** [22]. *i** is a graph, where each node represents an *actor* (or system component) and each link between two actors indicates that one actor depends on the other for some goal to be attained. A dependency describes an "agreement" (called *dependum*) between two actors: the *depender* and the *dependee*. The *depender* is the depending actor, and the *dependee,* the actor who is depended upon. The type of the dependency describes the nature of the agreement. *Goal* dependencies represent delegation of responsibility for fulfilling a goal; *softgoal* dependencies are similar to goal dependencies, but their fulfillment cannot be defined precisely; task dependencies are used in situations where the dependee is required.

As shown in Fig. 10, actors are depicted as circles; dependums – goals, softgoals, tasks and resources – are respectively represented as ovals, clouds, hexagons and rectangles; dependencies have the form depender → dependum → dependee. From this, a common actor, the *joint manager*, assumes two roles: a *private interface role* to coordinate partners of the alliance, and a *public interface role* to take strategic decisions, define policy for the private interface, represent the interests of the whole partnership with respect to external stakeholders and ensure communication with the external actors. Each partner can control himself on a local dimension and interact directly with others to exchange resources, data and knowledge.

MULTI-PATH UI DEVELOPMENT: USIXML

USIXML is intended to cover the specification of multiple models involved in UI design such as: task, domain, presentation, dialog, and context of use, which is in turn decomposed into user, platform, and environment. These models are structured according to the four layers of the Cameleon framework depicted in Fig. 11: task & concepts (T&C), Abstract User Interface (AUI), Concrete User Interface (CUI), and Final User Interface (FUI).

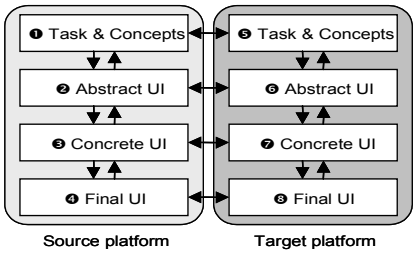


Figure 11. The Cameleon Reference Framework.

- At the FUI level, the rendering materializes how a particular UI coded in one language is rendered depending on the UI toolkit, the window manager and the presentation manager.
- The CUI level is assumed to abstract the FUI independently of any computing platform; this level can be further decomposed into two sub-levels: platform-independent CIO and CIO type. For example, a HTML push-button belongs to the type "Graphical 2D push button". Other members of this category include a Windows push button and XmButton, the OSF/Motif counterpart.
- Since the AUI level is assumed to abstract the CUI independently of any modality of interaction, this level can be further decomposed into two sub-levels: modality-independent AIO and AIO type. For example, a software control and a physical control (e.g., a physical button on a control panel or a function key) both belong to the category of control AIO.
- At the T&C level, a task of a certain type (here, download a file) is specified that naturally leads to AIO for controlling the downloading.

SketchiXML will first generate CUI specifications as this level represents a reasonable degree of expressiveness. Therefore, we will only describe this model into details in the next section. AUI specifications can come later on.

Concrete User Interface

A CUI is a UI model allowing a specification of an appearance and behavior of a UI with elements that can be perceived by users. A CUI consists of:

- *Modality dependent* i.e., an instance of a CUI addresses a single modality at a time. Two modalities fall in the intended scope of USIXML: graphical and auditory.
- *Platform independent* i.e., elements populating a CUI realize an abstraction of common languages used to develop UIs.
  - Concrete Interaction Objects (CIOs) realize an abstraction of widget sets found in popular graphical toolkits (Java AWT/Swing, HTML 4.O, Flash DRK6). A CIO is defined as an entity that users can perceive and/or manipulate (e.g., a push button, a list box, a check box). CIOs are divided into two types: graphical containers (e.g., window, panel, table, cell, dialog box) and graphical individual components (e.g., a button, a text component, a menu, a spin button).
  - The layout of the CUI is defined without any absolute coordinates. A box embedding mechanisms is used to specify a layout. Alignments between CIOs are defined with a special relationship called alignment.

Fig. 4 shows a declaration of a window containing a set of labels, buttons, text fields, combo boxes allowing the user to make a query. A CUI is also equipped with a mechanism, called dialog, allowing the specification of the dynamic behavior of a CUI. This mechanism covers a navigation definition language and a powerful event/action language.

SKETCHIXML: AN AGENT ARCHITECTURE FOR INTERFACES SKETCHING

In the previous sections, we have introduced the different feature to be included in SketchiXML. The application will have to, amongst all, make shapes recognition, provide spatial shapes interpretation, provide usability advices, solve ambiguities, and generate USIXML specifications. In addition, SketchiXML will also allow the user to define to what extend the application of these features must be automated. Indeed, the designer will be free to define the behavior of the whole application. For instance, designers may consider that they do not need usability advices, or that they just want to be advised on major issues. Some designers may also be willing to disable the shapes recognition during the design process as they do not want to be interrupted during the design process. Moreover, even if not depicted in the previous sections, SketchiXML will also have to be open and modular, as new feature are likely to be added later.

On basis of these requirements, we have considered that a BDI agent-oriented architecture were particularly judicious. Indeed, such architectures permit to build robust and flexible applications by distributing the responsibilities among autonomous and cooperating agents. In that situation all the agents are in charge of a specific part of the process, and cooperate together in order to provide the service required according to the designer preferences. This kind of approach appears to be more flexible, modular and robust than traditional including object-oriented ones.

The following section presents how we have applied the joint-venture organizational style to design the architecture of SketchiXML and how we have used SKwyRL-ADL to formally specify each architectural aspect (belief, goal, plan, action, interface, configuration, service) of the application. The joint-venture architectural style was chosen on basis of non-functional

requirement depicted in [7]. Among all organizational styles defined in the SKwyRL framework, the joint venture fits to SketchiXML as it is the most open and distributed organizational style.

SketchiXML Architecture

Throughout the section 2, we have presented the working principles of the application with a small scenario. On basis of that scenario, this section presents the multi-agent architecture of SketchiXML depicted on figure 13, and the distribution of the competencies among the agent participating in the system.

Fig. 13 shows that the Coordinator plays the role of the joint manager private interface and that the Broker plays the role of the joint manager public interface. Other joint venture partners are the Parser, the Shapes Recognizer, the Data Editor, the Ambiguity Solver, the Usability Adviser, the XML Specificator and the Graphical Editor.

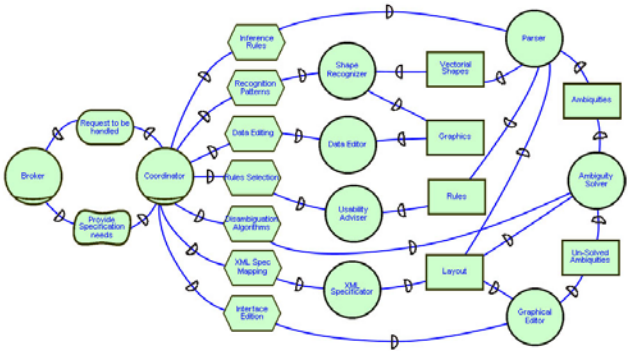


Figure 13. The SkechiXML Architecture in joint-venture.

Thus, when a user wishes to create a specification, it contacts the Broker agent, which serves as an intermediary between the external actor and the organizational system. The Broker will query the user for all relevant information needed for the process, such as depicted on Fig. 2. According to the criteria entered, the coordinator will choose the most suitable handling and coordinates all the agents participating in the process in order to meet the objectives determined by the user. The coordinator also plays the role of transmitting the results back to the Broker, once the specification process is completed.

Once the user has provided all the information needed for the process, the coordinator is informed and chooses the most suitable handling according to the request; in this case, it contacts the Data Editor agent. Following that, this agent displays a white board allowing the user to draw its hand-sketch interface. All the strokes are collected and then transmitted to the Shapes Recognizer for identification. The recognition engine of this agent is based on JavaSketchIt [4] and the CALI library [14], which appears to be one of the more powerful application in that domain. Indeed, this application is not only able to identify shapes of different sizes, rotated at arbitrary angles, drawn with dashed, continuous strokes or overlapping lines, but also use fuzzy logic to associate degrees of certainty to recognized shapes to overcome uncertainty and imprecision in shape sketches. Thus, the Shapes Recognizer provide to the parser all the shapes recognized with all the relevant information such as location, dimension or degree of certainty. On basis of these Shape set, the parser will attempt to create a components layout.

The technique used for the creation of this layout is the same than the one used by JavaSketchIt, which is based on a set of fuzzy spatial relations allowing us to deal with imprecise spatial combinations of geometric shapes. In addition to widget recognition, the parser agent will have to integrate a set of usability rules provided by the usability adviser. The usability adviser will also assist the designer for the conception of the UIs, if required. Indeed, the designer may require real-time assistance for the design process. In this case, on basis of all the widgets recognized, the agent will proceed to the interface critique, and utter advice on usability matters. Eventually, if the Parser fails to identify all the components or to apply all the usability rules, then the ambiguity solver agent may be invoked. This agent will choose how to optimally solve the problem according to the initial parameters entered by the user. The agent can either attempt to solve the ambiguity itself using its set of disambiguation algorithms, or to invoke a third agent, the graphical editor agent. The graphical editor displays all the widget recognized at this point, as a classical element-approach software, and highlights all the components with low degree of certainty for confirmation. Once one the last three agents evoked considers the degree on certainty for all the widgets to be sufficient, the components layout is transmitted to the XML Specificator, for conversion to USIXML.

SketchiXML Formal Specification

The architecture described in Figure 13 gives an organizational representation of the system-to-be including relevant actors and their respective goals, tasks and resource inter-dependencies. This model can serve as a basis to understand and discuss the assignment of system functionalities but it is not adequate to provide a precise specification of the system details. Thus, to complete the organizational representation of the SketchiXML architecture, we propose to use SKwyRL-ADL. SKwyRL-ADL provides a finite set of formal agent-oriented constructors that allow detailing in a formal and consistent way the software architecture as well as its agent components and their behaviors.

```
Agent:{ Shapes Recognizer
Interface:
  Sensor[require(Graphics)]
  Effector[provide(Shape_Recognized)]
  Effector[provide(Recognition_Pattern)]
KnowledgeBase:
  System_KB
  Data_KB
  Shapes_Patterns_KB
Capabilities:
  Shapes_Recognition_CP
  Handle_Dotsset_CP
       }
```

Figure 14. Agent description of the Shapes Recognizer.

Fig. 14 and 15 show a formal description of the *Shapes Recognizer* and of the *Parser* agents. Three aspects of this agent component are of concern here: the *interface* representing the interactions in which the agent will participate, the *knowledge base* defining the agent knowledge capacity and the *capabilities* defining agent behaviors.

```
Agent:{ Parser
Interface:
  Sensor[require(Usability_Rules)]
  Sensor[require(Shape_Recognized)]
  Effector[provide(Layout)]
  Effector[provide(Inference_Rule)]
  Effector[provide(Ambiquities_Set)]
KnowledgeBase:
  Selected_Usability_Rules_KB
  VectorialLayout_KB
  ComponentLayout_KB
  System_KB
  Widgets_Patterns_KB
Capabilities:
  Convert_Vectorial_Layout_CP
  Create_Components_Layout_CP
  Implement_usability_rules_CP
  Disambiguation_CP
       }
```

Figure 15. Agent structure description of the parser.

SkwyRL-ADL allows to work at different levels of architectural abstractions (i.e., different views of the system architecture) to encapsulate different components of the system in independent hierarchical descriptions. For instance, in Fig. 15 the *parser* agent has six knowledge bases (KB) and four capabilities (CP) and in Fig. 14 the *shapes recognizer* has four knowledge bases and two capabilities; but the description level chosen here does not specify the details of the beliefs composing the KB or the plans and events composing each capability.

The rest of the section focuses on the *Shapes Recognizer* agent to give an example of a refinement specification with SKwyRL-ADL for each of the three aspects of the agent: interface, KB and capabilities.

Interface. The agent *interface* consists of a number of effectors and sensors for the agent. Each of them represents an action in which the agent will participate. Each effector provides a service that is available to other agents, and each sensor requires a service provided by another agent. The correspondence between a required and a provided service defines an interaction. For example, the Parser requires the *ShapesRecognized* service that the *Shapes Recognizer* provides.

Such interface definition points two aspects of an agent. Firstly, it indicates the expectations the agent has about the agents with which it interacts. Secondly, it reveals that the interaction relationships are a central issue of the architectural description. Such relationships are not only part of the specification of the agent behavior but reflect the potential patterns of communication that characterize the ways the system reason about itself.

The required query translation service is described in greater detail in Fig. 16. We can see that the *Shapes Recognizer* (sender) initiates the service by asking the *Parser* (receiver) to convert the vectorial layout. To this end, the *Shapes Recognizer* provides to the *Parser* a set of parameters allowing to build the corresponding components layout. Such *Shapes Recognizer* transaction is specified as belief with the predicate *shapeset* and the following terms:

```
shapeset(Parent_Id, Shape_Id, ShapeType, De-
      gree_Certainty, coordinate (+))
```

Each term represents, respectively, the Id of the source from which the shape is extracted, the individual Id of the shape, the type of the shape, the degree of certainty associated to the shape, and finally the co-ordinates of the shape.

```
service:{Tell(Shapes_Recognized)
    sender: Shapes_Recognizer
    parameters: Pid:Parent_Id ∧ Sid: Shape_ID ∧
      dc: degree_of_certainty ∧ st:ShapeType
      ∧  co: coordinates (+)
    receiver: Parser
effect:  Add(Vectorial_Shapes  _KB,  shapeset(Pid,
        Id, st, dc,  co(+))}
```

Figure 16. A Service Specification.

The service effect indicates that a new shape is added to the shape set belief in the Vectorial_Shapes _KB of the parser.

Knowledge Bases. A *knowledge base* (KB) is specified with a name, a body and a type. The name identifies the KB whenever an agent wants to query or modify them (add or remove a belief). The body represents a set of beliefs in the manner of a relational database schema. It describes the beliefs the agent may have in terms of fields. When the agent acquires a new belief,

values for each of its fields are specified and the belief is added to the appropriate KB as a new tuple. The *KB type* describes the kind of formal knowledge used by the agent.

```
KnowledgeBase: {Vectorial_Shapes_KB
  KB_body:
    shapeset(Parent_Id, Shape_Id, ShapeType, De-
    gree_Certainty, coordinate (+))
  KB_type: closed_world
                }

KnowledgeBase: { Widgets_Patterns_KB
  Kb_body:
    Widgets(Id, type, description, Relation.id(+))
    relation(Id, description, fuzzy_rules(+))
  Kb_type: closed_world
                }
```

Figure 17. Parser knowledge bases specification.

A *Closed world* assumes that the agent is operating in a world where every tuple it can express is included in a KB at all times as being true or false. Inversely, in an *open world* KB, any tuple not included as true or false is assumed to be unknown. The '+' symbol means that the attribute is multi-valued.

Capabilities formalize the behavioral elements of an agent. It is composed of plans and events that together define the agent's abilities. It can also be composed of sub-capabilities that can be combined to provide complex behavior. Fig. 18 shows the `Sketch_Recognition_CP` capability of the *Shapes Recognizer* agent. The capability is made up of three plans and four events.

```
Capability: { Sketch_Recognition_CP
  CP_body:
    Plan Graphics_conversion
    Plan On_demand_Graphics_conversion
    Plan Display_conversion
    Plan Force_conversion
    SendEvent Vectorial_Shape
    SendEvent EndOfConversion
    SendEvent BeginningOfConversion
    PostEvent ReadyToCreateNext
            }
```

Figure 18. A Capability Specification.

A plan defines the sequence of actions and/or services (i.e., actions that involve interaction with other agents) the agent selects to accomplish a task or achieve a goal. A plan consists of:

- An *invocation condition* detailing the circumstances, in terms of beliefs or goals, that cause the plan to be triggered.
- An *optional context* that defines the preconditions of the plan, i.e., what must be believed by the agent for a plan to be selected for execution.
- The *plan body* specifies either the sequence of formulae that the agent needs to perform, a formula being either an action or a service to be executed.
- An *end state* that defines the post-conditions under which the plan succeeds.
- And optionally a set of services or actions that specify what happens when a *plan fails* or *succeeds*.

Both plans `On_demand_Graphics_conversion` and `Graphics_conversion` are triggered each time a new `Source` belief is added to the `Data_KB`. The argument values of the Source belief are required by the Ask (graphics) service that the *Shapes Recogniser* initiates. However, the plan is only executed if the value of the type attributes assigned to the Id is "*manual*". That means that the designer has manually asked the system to do the shape recognition. This second condition is expressed by the plan context. The aim of the context is to help to the selection of the most appropriate plan in a given situation. Nevertheless, the presence of such a context is not mandatory, if there is no context, then the plan is selected only on the basis of the invocation condition.

```
Plan:{ On_demand_Graphics_conversion
invoc:

   Add(Data_KB,Source(Id, DS(+))
    // with Id:SourceIdentifier From ShapesRecog-
    nizer.Ask(Graphics).reply_with//

    // with DS: DotsSets From ShapesRecog-
    nizer.Ask(Graphics).reply_with//

context:
     handling_mode(Id, type = "manual")

body:
    ∀ DS: DotsSets ∈ Source(Id, type, DS(+), Pc(+))
     DO
    action Identify_Shape(Shapes(Id, type, descrip-
      tion, fuzzy rules))
    as st: ShapeType ∧ dc: degree_of_certainty
    service:{Tell(Shapes_Recognized)
      sender: Shapes_Recognizer
      parameters: Pid:Parent_Id ∧ Sid: Shape_ID ∧
        dc: degree_of_certainty ∧ st:ShapeType ∧
        co: coordinates (+)
      receiver: Parser
    effect: Add(Vectorial_Shapes_KB, shapeset (Id,
            st(+), co, d)
     End-DO
endstate:
     ∀ DS: Data ∈ Source(Id, type, DS(+), Pc (+))
      Add(Vectorial_Shapes_KB, shapeset (Pid, Id,
        st, dc, co)
suceed:
    effect:
      service:{Tell(Recognition_Completed)
      sender: Shapes_Recognizer
      parameters: Id:Data_Id
      receiver: Parser
      effect: Add(System_KB, ToDo(Id, "completed")
fail:

        Plan: Force_conversion}
```

Figure 19. Specification of the Convert_sktech plan.

As soon as the invocation condition and the context are true, the sequence of actions or services specified in the plan body can be executed. The plan body of the `On_demand_Graphics_conversion` plan is composed by the sequence of an action and a service. The *Shapes*

Recogniser identifies all the shapes drawn by the user, and transmits the information to the *Parser*.

The plan succeeds if the statement described by the endstate is successful. The succeed specification of the `On_demand_Graphics_conversion` tell the Parser that all the shapes were recognized, while the fail specification of the `On_demand_Graphics_conversion` plan returns to the execution of the `Force_conversion` plan.

Configuration. To describe the complete topology of the system architecture, the agents of an architectural description are combined into a SKwyRL *configuration*.

```
Configuration SketchiXML
  Agent  Public interface
  Agent  Coordinator
  Agent  Shape_Recognitor
  Agent  Parser
  Agent  Data_Editor
  Agent  Ambiguity_Solver
  Agent  Usability_Adviser
  Agent  XML_Specificator
  Agent  Graphical_Editor
  Service Tell(Shape_Recognized)
  Service Ask(Shape_Recognized)
  Service Achieve(Provide_Specs)
  Service Do(Provide_Specs)
  Service Import(Usability_Rules)
  Service Export(Usability_Rules)
  ….
  Instances
  PI:  Public interface
  CO:  Coordinator
  SR:  Shape_Recognitor
  PA:  Parser
  DE:  Data_Editor
  AS:  Ambiguity_Solver
  UA:  Usability_Adviser
  XS:  XML_Specificator
  GE:  Graphical_Editor
  TellReco: Tell(Shape_Recognized)
  AskReco: Ask(Shape_Recognized)
  AchSpecs: Achieve(Provide_Specs)
  DoSpecs: Do(Provide_Specs)
  ImportErgo: Import(Usability_Rules)
  ExportErgo: Export(Usability_Rules)
  ….
   Collaborations
  CO. AchSpecs  --- DoSpecs. XS;
  SR. TellReco --- AskReco. PA;
  PA. ImportErgo --- ExportErgo. ER;
  ….
End SketchiXML
```

Figure 20. The SketchiXML Configuration .

Instances of each agent or service that appear in the configuration must be identified with an explicit and unique name. The configuration also describes the collaborations (i.e., which agent participates in which interaction) through a one-to-many mapping between provided and required service instances.

Such a configuration allows for dynamic reconfiguration and architecture resolvability at run-time. Configurations separate the description of composite structures from the description of the elements that form those compositions. This permits reasoning about the composition as a whole and to reconfigure it without having to examine each component of the system.

CONCLUSIONS AND FUTURE WORK

Several researchers have proposed alternatives for code generation from hand-sketch interface design. But, in a context where the number of computing-platform and system environments is exploding, the possibility offered by all the current application to generate UIs for a single platform in a unique language, seems to be insufficient. With SketchiXML we have introduced a new innovative concept. Firstly, the application will provide USIXML file as output, and thus overcome the language neutrality weakness of the current approaches. Secondly, the application will be based on a set of experts collaborating together in a flexible way. Indeed, on basis of the criteria provided by the designer, the experts will have to adapt their roles and collaborations. From these requirements, we have developed trough this paper a formal specification of the BDI multi-agent architecture of SketchiXML with the SkwyRL-framework. Each expert depicted in the requirements is then represented by an autonomous and collaborative agent part of an organizational system.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of the *Request* research project under the umbrella of the *WIST* (Wallonie Information Société Technologies) programme under convention n°031/5592 RW REQUEST). We also warmly thank Joaquim A. Jorge and Anabela Caetano for allowing us to use JavaSketchIt for our research.

REFERENCES

1. Anderson, R.E. Social impacts of computing: Codes of professional ethics. *Social Science Computing Review* 10, 2 (1992), 453-469.
2. Bailey, B.P. and Konstan, J.A. Are Informal Tools Better? Comparing DEMAIS, Pencil and Paper, and Authorware for Early Multimedia Design. *Proc. of the ACM Conference on Human Factors in Computing Systems CHI'2003* (Fort Lauderdale, April 2003). ACM Press, New York, 2003, pp. 313-320.
3. Bratman, M.E., *Intention, Plans and Practical Reason*. Harvard University Press, 1987.
4. Caetano, A., Goulart, N., Fonseca, M. and Jorge, J. JavaSketchIt: Issues in Sketching the Look of User Interfaces. *Proc. of the 2002 AAAI Spring Symposium - Sketch Understanding* (Palo Alto, March 2002), pp. 9-14.
5. Clements, P.C. A Survey of Architecture Description Languages. *Proc. of the 8th Int. Workshop on Software Specification and Design* (Paderborn, March 1996).
6. Farenc, Ch., Liberati, V. and Barthet, M.F. Automatic Evaluation: What are the Limits? *Proc. of 2nd Int. Workshop on Computer-Aided Design of User Interfaces CADUI'96* (Namur, 5-7 June 1996), J. Vanderdonckt

(ed.).

7. Faulkner, S. and Kolp, M. Towards an Agent Architectural Description Language for Information Systems. *Proc. of the 5th Int. Conf. on Enterprise Information Systems ICEIS 03* (Angers, April 2003).
8. Faulkner, S., *An Architectural Framework for Describing BDI Multi-Agent Information Systems*. Ph.D. Thesis, Université Catholique de Louvain, Institut d'Administration et de Gestion (IAG), Louvain-la-Neuve, Belgium, May 2004.
9. Fonseca, M.J. and Jorge, J.A. Using Fuzzy Logic to Recognize Geometric Shapes Interactively. *Proc. of the 9th Int. Conf. on Fuzzy Systems FUZZ-IEEE'00* (Antonio, May 2000), IEEE Computer Society Press, 2000, pp. 191-196.
10. Fonseca, M.J., Pimentel, C. and Jorge, J.A. CALI: An Online Scribble Recognizer for Calligraphic Interfaces. *Proc. of the 2002 AAAI Spring Symposium - Sketch Understanding* (Palo Alto, March 2002), pp. 51-58.
11. Hong, J.I., Li, F.C., Lin, J., and Landay, J.A. End-User Perceptions of Formal and Informal Representations of Web Sites, *Extended Abstracts of Proc. of ACM Conf. on Human Factors in Computing Systems CHI 2001* (Seattle, March 31-April 5, 2001). ACM Press, New York, 2001.
12. JACK Intelligent Agents. http://www.agent-software.com/.
13. Jadex BDI Agent Systems http://vsis-www.informatik.uni-hamburg.de/projects/jadex/.
14. Kolp, M., Giorgini, P. and Mylopoulos, J. An Organizational Perspective on Multi-agent Architectures. *Proc. of the 8th Int. Workshop on Agent Theories, architectures, and languages ATAL'01* (Seattle, August 2001).
15. Landay, J.A. *Interactive Sketching for the Early Stages of User Interface Design*. Ph.D. thesis, report #CMU-CS-96-201. Computer Science Department, Carnegie Mellon University, Pittsburgh, December 1996.
16. Leclercq, P. and Juchmes, R. The Absent Interface in Design Engineering, *Proc. of Artificial Intelligence for Engineering Design, Analysis and Manufacturing AIEDAM* Special Issue: Human-computer Interaction in Engineering Contexts, Cambridge University Press, Cambridge.
17. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and Lopez-Jaquero, V. USIXML: a Language Supporting Multi-Path Development of User Interfaces. *Proc. of 9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems EHCI-DSVIS'2004* (Hamburg, July 11-13, 2004).
18. Mahajan, R. and Shneiderman, B., Visual and Textual Consistency Checking Tools for Graphical User Interfaces. *IEEE Trans. Software Engineering* 23, 11 (1997) 722-735.
19. Michotte, B., Limbourg, Q., and Vanderdonckt, J. GrafiXML, A User Interface Builder Based on USIXML, IAG, Louvain-la-Neuve, July 2004.
20. Plimmer, B. and Apperley, M. Interacting with Sketched Interface Designs: An Evaluation Study. *Proc. of ACM Conf. on Human Factors in Computing Systems CHI'04* (Vienna, April 2004). ACM Press, New York, 2004.
21. Wooldridge, M. and Jennings, N.R. (eds.). Special Issue on Intelligent Agents and Multi-Agent Systems. *Applied Artificial Intelligence Journal* 9, 4 (1996).
22. Yu, E. *Modeling Strategic Relationships for Process Reengineering*. Ph.D. thesis, Department of Computer Science, University of Toronto, Toronto, 1995.

APPENDIX

This appendix lists all widgets to be recognized for web-based applications.

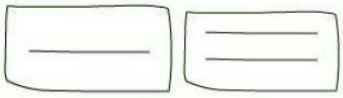
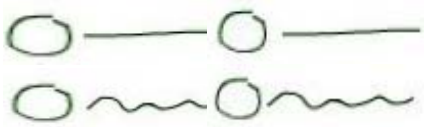
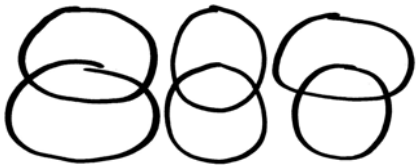
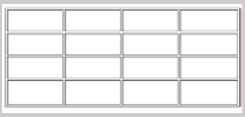
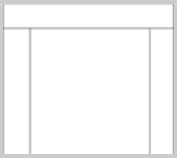
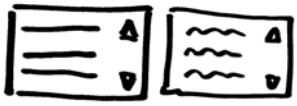
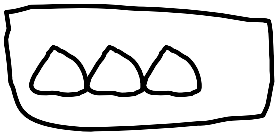
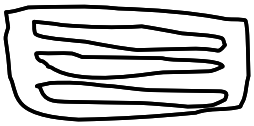
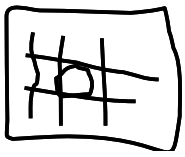
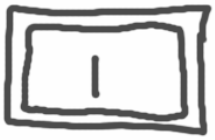
Widget Type	Graphical Representation	Description	Sketching proposition
Text	This is text	This widget is aimed at displaying non – editable text on the web page.	
TextField		This widget allows to the user to enter a single-lined text value.	
TextArea		This widget offers the same function than the Textfield, but adding the opportunity to enter several lines of text.	
Button	Button	This widget allows the user to trigger an action (of any kind).	
Search Field	submit	This widget is composed of a text field and a button. It allows the users to submit a search.	
Login	User name Password Log in	This widget is composed of two text fields, a button, and text. It allows the users to log in on the website with a login and a password.	
Log out	Log out	This widget is an incremented version of the button widget. It allows the user to log out of the web site.	
Reset Form	Reset	This widget is an incremented version of the button widget. It allows the user to reset a form.	
Validate	Validate	This widget is an incremented version of the button widget. It allows the user to validate a form.	
RadioButton	Radio Button	This widget (usually set of widgets) allows the users to make a single selection between several choices.	
CheckBox	Check Box	This widget offers almost the same property than the radio button, except that multiple selection is allowed	
Combobox		This widget allows the user to make a single selection within a list.	

Image		This widget is aimed at displaying a picture on a web page. This picture can be “clickable” or not. The arrow means that the picture will “clickable”	
Multi Media Area		This widget is aimed at displaying a multi-media object (video, flash, ad banner…) on the web page. As the image widget, the arrow indicates that the component must be “clickable”	
Layer		This widget represents a multi layered area.	We do not have any sketching representation for this widget. The idea is to work on only one layer at a time, and to have the opportunity to switch from one layer to another with some tabs. Ideally, we should have the opportunity to work on a layer, and see the others layers by transparency with different colors.
Group Box		This widget is a container.	
Table		This widget represents a table	
Separator		This widget is a horizontal or vertical separator on the web page.	
Frames		These widgets represent frames. They make it possible to display several web pages on a common frameset.	We do nt have any sketching representation for this widget. We would represent the frame by straight lines, with a condition. It the line touches a border or another frame, then the line defines a frame.
Hyperlink	Hyperlink	This widget is an hyperlink, when clicking on it, the user is redirected on another location	
Anchor		Puts a hyperlink at a particular place on a Web page	
ListBox		This widget allows the user to make a single or multiple selections within a list.	

TabDia-logBox		This widget allows the user to switch from one pane to another thanks to the tab.	
Menu		This widget is a generic menu.	
Color Picker		This widget allows the user to pick a color from a palette of available colors.	
File Picker		This widget allows the user to pick a file present on the computer or eventually on a remote computer.	
Date Picker		This widget allows the user to pick a date on an agenda.	
Hour Picker		The user has to choose an hour, but he might also specify minutes and seconds.	
Toggle Button		As the classical button does, the toggle button allows the user to trigger an action (of any kind). The main difference lies in the fact that this button has two different states. The first click makes the button to become true, while the second makes the button to become false again.	
Slider		This widget allows the user to choose a value from a values ladder by moving a cursor	
Progress Bar		This widget displays the state of execution of a process in real time	
Spinner		This widget allows increasing or decreasing a numerical value in a bounded interval.