SketchiXML: An Informal Design Tool for User Interface Early Prototyping

Adrien Coyette, Jean Vanderdonckt, and Quentin Limbourg

Belgian Lab. of Computer-Human Interaction (BCHI), Information Systems Unit (ISYS)
Louvain School of Management, Université catholique de Louvain,
Place des Doyens 1, B−1348 Louvain-la-Neuve (Belgium)
{coyette, vanderdonckt, limbourg}@isys.ucl.ac.be – http://www.isys.ucl.ac.be/bchi

Abstract. Sketching consists of a widely practised activity during early design phases of product in general and for user interface development in particular in order to convey informal specifications of the interface before actually implementing it. It is quite interesting to observe that designers as well as end users have abilities to sketch parts or whole of the final user interface they want, while discussing the advantages and shortcomings. SketchiXML consists of a multi-platform multi-agent interactive application that enables designers, developers, or even end users to sketch user interfaces with different levels of details and support for different contexts of use. The results of the sketching are then analyzed to produce interface specifications independently of any context, including user and platform. These specifications are exploited to progressively produce one or several interfaces, for one or many users, platforms, and environments.

1 Introduction

Designing the right User Interface (UI) the first time is very unlikely to occur. Instead, UI design is recognized as a process that is intrinsically open (new considerations may appear at any time), iterative (several cycles are needed to reach an acceptable result), and incomplete (not all required considerations are available at design time). Consequently, means to support early UI design has been extensively researched [12] to identify appropriate techniques such as paper sketching, prototypes, mock-ups, diagrams, etc. Most designers consider hand sketches on paper as one of the most effective ways to represent the first drafts of a future UI [4,7,12, 15,16,17]. Indeed, this kind of unconstrained approach presents many advantages: sketches can be drawn during any design stage, it is fast to learn and quick to produce, it lets the sketcher focus on basic structural issues instead of unimportant details (e.g., exact alignment, typography, and colors), it is very appropriate to convey ongoing, unfinished designs, and it encourages creativity, sketches can be performed collaboratively between designers and end-users. Furthermore, the end user may herself produce some sketches to initiate the development process and when the sketch is close enough to the expected UI, an agreement can be signed between the designer and the end user, thus facilitating the contract and validation. Van Duyne et al. [20] reported that creating a low-fidelity UI prototype (such as UI sketches) is at least 10 to 20 times easier and faster than its equivalent with a high-fidelity prototype (such as produced in UI builders). The idea of developing a computer-based tool for sketching UIs naturally emerged from these observations [12,17]. Such tools would extend the advantages provided by sketching techniques by: easily creating, deleting, updating or moving UI elements, thus encouraging typical activities in the design process [3] such as model-checking and revision. Some research was carried out in order to propose a hybrid approach, combining the best of the hand-sketching and computer assisted interface design, but this marriage highlights five shortcomings:

1. Some tools only support sketching activities, without producing any output: when the designer and the end user agreed upon a sketch, a contract can be signed between them and the development phase can start from the early design phase, but when the sketch is not transformed, the effort is lost.
2. Sketching tools that recognize the drawing do produce some output, but not in a reusable format: the design output is not necessarily in a format that is directly reusable as development input, thus preventing reusability.
3. Sketching tools are bound to a particular programming language, a particular UI type, a particular computing platform or operating system: when an output is produced, it is usually bound to one particular environment, therefore preventing developers from re-using sketches in new contexts, such as for various platforms.
4. Sketching tools do not take into account the sketcher's preferences: as they impose the same sketching scheme, the same gestures for all types of sketchers, a learning curve may prevent these users from learning the tool and efficiently using it.
5. Sketching tools do not allow a lot of flexibility in the sketch recognition: the user cannot choose when recognition will occur, degrading openness and when this occurs, it is difficult to return to a previous state.

In the remainder of this paper, Section 2 demonstrates that state-of-the-art UI sketching tools all suffer from some of the above shortcomings. Section 3 provides an overview of the Concrete User Interface (CUI) used in the sketching process, which results from widget abstraction. In Section 4, these widgets are recognized on demand. The multi-agent architecture of SketchiXML is outlined to support various scenarios in different contexts of use with examples. Section 5 concludes the paper.

2 Related Work

UI prototypes usually fall into three categories depending on their degree of fidelity, that is the precision to which they reproduce the reality of the desired UI.

The *high-fidelity* (Hi-Fi) prototyping tools support building a UI that looks complete, and might be usable. Moreover, this kind of software is equipped with a wide range of editing functions for all UI widgets: erase, undo, move, specify physical attributes, etc… This software lets designers build a complete GUI, from which is produced an accurate image (e.g., Adobe Photoshop, PowerPoint) or code in a determined programming language (e.g., Visual Basic, DreamWeaver). Even if the final result is not executable, it can still be considered as a high fidelity tool given that the result provided looks complete.

The *medium-fidelity* (Me-Fi) approach builds UI mock-ups giving importance to content, but keeping secondary all information regarding typography, color scheme or others minor details. A typical example is Microsoft Visio, where only the type, the size and the contents of UI widgets can be specified graphically.

Low-fidelity (Lo-Fi) drafting tools are used to capture the general information needed to obtain a global comprehension of what is desired, keeping all the unnecessary details out of the process. The most standard approaches for Lo-Fi prototyping are the “paper and pencil technique”, the “whiteboard/blackboard and post-it approach” [16]. Such approaches provide access to all the components, and prevent the designer from being distracted from the primary task of design. Research shows that designers who work out conceptual ideas on paper tend to iterate more and explore the design space more broadly, whereas designers using computer-based tools tend to take only one idea and work it out in detail [20]. Many designers have reported that the quality of the discussion when people are presented with a Hi-Fi prototype was different than when they are presented with a Lo-Fi mock up. When using Lo-Fi prototyping, the users tend to focus on the interaction or on the overall site structure rather than on the color scheme or others details irrelevant at this level.

Consequently, Lo-Fi prototyping offers a clear set of advantages compared to the Hi-Fi perspective, but at the same time suffers from a lack of assistance. For instance, if several screens have a lot in common, it could be profitable to use copy and paste instead of rewriting the whole screen each time. A combination of these approaches appears to make sense, as long as the Lo-Fi advantages are maintained. This consideration results two families of software tools which support UI sketching and representing the scenarios between them, one with and one without code generation.

DENIM [15] helps web site designers during early design by sketching information at different refinement levels, such as site map, story board and individual page, and unifies the levels through zooming views. DENIM uses pen input as a natural way to sketch on screen, but do not produce any final code or other output.

In contrast, SILK [12], JavaSketchIt [4] and Freeform [16,17] are major applications for pen-input based interface design supporting code generation. SILK uses pen input to draw GUIs and produce code for OpenLook operating system. JavaSketchIt proceeds in a slightly different way than Freeform, as it displays the shapes recognized in real time, and generates Java UI code. JavaSketchIt uses the CALI library [4] for the shape recognition, and widgets are formed on basis of a combination of vectorial shapes. The recognition rate of the CALI library is very high and thus makes JavaSketchIt easy to use, even for a novice user. Freeform only displays the shapes recognized once the design of the whole interface is completed, and produces Visual Basic 6 code. The technique used to identify the widgets is the same than JavaSketchIt, but with a slightly lower recognition rate. Freeform also supports scenario management thanks to a basic storyboard view similar to that provided in DENIM.

To enable sketching of widgets which are traditionally found in window managers, there is a need to have an internal representation of the UI being built, in terms of those widgets. Therefore, the next section introduces a means for specifying such a UI in terms of concrete interaction objects, instead of widgets.

3 The Concrete User Interface in UsiXML

The need for abstracting widgets existing in various toolkits, window managers, or Integrated Development Environments (IDEs) has appeared since the early nineties. At that time, the main goal for introducing an abstraction for widgets was the desire to specify them independently of any underlying technology, mainly the different operating systems and the different window managers working with the same operating system. For this purpose, the notion of Abstract Interaction Object (AIO) has been introduced to provide an abstraction of the same widget across those different toolkits, window managers, and operating systems so as to manipulate one single specification of this widget [18]. Another goal was the desire to entirely specify the presentation and the behavior of the widget [6,8].

Since that time, much progress has been accomplished towards improving the expressiveness of these abstractions, to the ultimate point of having extensive specifications of an entire UI in a User Interface Description Language (UIDL). The most representative examples are XML-compliant UIDLs such as UIML [1], UsiXML [13,14], and XIML [9]. A noticeable example is also the effort of specifying domain-oriented widgets such as those covered by the ARINC 661 Specifications in the domain of widgets for automated cockpits [2,3]. In order to be rigorous for the abstraction with respect to which the specification needs to be expressed, a reference framework is first introduced.

3.1 A Reference Framework for User Interfaces in Multiple Contexts

The foundation of the approach adopted here is to rely constantly on the same User Interface Description Language (UIDL) throughout the development life cycle. This UIDL is UsiXML (User Interface eXtensible Markup Language – http://www.usi xml.org) which is characterized by the following principles [13,14]:

- *Expressiveness of UI*: any UI is expressed depending on the context of use thanks to a suite of models that are analyzable, editable, and manipulable by a software agent.
- *Central storage of models*: each model is stored in a model repository where all UI models are expressed similarly.
- *Transformational approach*: each model stored in the model repository may be subject to one or many transformations supporting various development steps. Each transformation is itself specified thanks to UsiXML [14].

Contrarily to other UIDLs such as UIML and XIML, UsiXML [12] enables specifying various levels of information and details until a final UI is obtained and depending on the project. It is not necessary to specify all models at all levels involved in the UI development life cycle. For this purpose, UsiXML is structured according to four basic levels of abstractions defined by the Cameleon reference framework [5] (Figure 1).

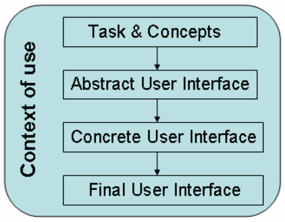


Fig. 1. The four levels of the Cameleon reference framework (source: [5]).

At the top level is the ***Task & Concepts*** level that describes the various interactive tasks to be carried out by the end user and the domain objects that are manipulated by these tasks. These objects are considered as instances of classes representing the concepts.

An ***Abstract UI*** (AUI) provides a UI definition that is independent of any modality of interaction (e.g., graphical interaction, vocal interaction, 3D interaction etc.). An AUI is populated by *Abstract Containers* (ACs), *Abstract Individual Components* (AICs) and abstract relationships between. AICs represent basic system interactive functions, which are referred to as *facets* (i.e., input, output, navigation, and control). In this sense, AICs are an abstraction of widgets found in graphical toolkits (like windows, buttons) and in vocal toolkits (like vocal input and output widgets in the vocal interface). Two AUI relationships that can be defined between AICs:

1. *Dialog transition*: specifies a navigation transition within a abstract container or across several abstract containers.
2. *Spatio-temporal relationship*: characterizes the physical constraints between AICs as they are presented in time and space.

As an AUI does not refer to any particular modality, we do not know yet how this abstract description will be concretized: graphical, vocal or multimodal. This is achieved in the next level.

The ***Concrete UI*** (CUI) concretizes an AUI for a given context of use into *Concrete Interaction Objects* (CIOs) so as to define layout and/or interface navigation of 2D graphical widgets and/or vocal widgets. Any CUI is composed of CIOs, which realize an abstraction of widgets sets found in popular graphical and vocal toolkits (e.g., Java AWT/Swing, HTML 4.0, Flash DRK6, VoiceXML, and VoxML). A CIO is defined as an entity that users can perceive and/or manipulate (e.g., push button, text field, check box, vocal output, vocal input, vocal menu). The CUI abstracts a Final UI in a way that is independent of any toolkit peculiarities.

The ***Final UI*** (FUI) is the operational UI, i.e. any UI running on a particular computing platform either by interpretation (e.g., through a Web browser) or by execution (e.g., after the compilation of code in an interactive development environment). The ***Context of use*** describes all the entities that may influence how the user's task is carrying out with the future UI. It takes into account three relevant aspects, each aspect having its own associated attributes contained in a separate model: *user type* (e.g., system experience, task experience, task motivation), *computing platform type* (e.g., mobile platform vs. stationary), and *physical environment type* (e.g., office conditions, outdoor conditions). These attributes initiate transformations that are applicable depending on the current context of use. In order to map different elements belonging to the models described above, UsiXML provides the designer with a set of pre-defined *mappings* [14]:

- *Manipulates*: maps a task onto a domain concept.
- *Updates*: maps an interaction object and a domain model concept (specifically, an attribute).
- *Triggers*: maps an interaction object and a domain model concept (specifically an operation).
- *Is Executed In*: maps a task onto an AUI or CUI element.
- *Is Reified By*: maps an abstract object into a concrete one through an abstraction transformation.

3.2 The Concrete User Interface in UsiXML

The semantics of the UsiXML are defined in a UML class diagram (Fig. 2 is illustrating a portion of this metamodel). Each class, attribute or relation of this class diagram is transformed into a XML Schema defining the concrete syntax of the UsiXML language in general. All other levels of the reference framework depicted in Figure 1 are equally expressed in UsiXML to support seamless transition between any level of abstraction to any other one. A CUI is assumed to be expressed without any reference to any particular computing platform or toolkit of that platform. A CUI model consists of a hierarchical decomposition of CIOs (any UI entity that users can perceive such as text, image, animation and/or manipulate such as a push button, a list box, or a check box) that are linked together with *cuiRelationships* between. A CIO is characterized by [14]:

- *id*: an internally attributed identifier of a CIO;
- *name*: a name given to the CIO to reflect its function, purpose;
- *icon*: a reference to an icon attached to the CIO, if any;
- *content*: a reference to the textual contents of a CIO, if any;
- *defaultContent*: the default value of its textual contents, if any;
- *defaultIcon*: the default icon of this CIO, if any;
- *defaultHelp*: the default text for helping the user on this CIO;
- *help*: the extended help system for helping the user on this CIO;
- *currentValue*: the current value of the CIO at run-time, if any.

At the second level, each CIO is sub-typed into sub-CIOs depending on the interaction modality chosen: *graphicalCIO* for GUIs, *auditoryCIO* for vocal interfaces, etc. Each *graphicalCIO* inherits from the above properties of the CIO. Specific attributes include, but are not limited to:

- *isVisible*: is set to true if a graphicalCio is visible;
- *isEnabled*: is set to true if a graphicalCIO is enabled;
- *fgColor* and *bgColor*: are the foreground and background colors;
- *toolTipDefaultContent*: for the default content of the tooltip;
- *toolTipContent*: the contents of the tooltip depending on the context of use, which may vary from one user to another;
- *transparencyRate*: for supporting translucid interfaces;

Each *graphicalCIO* can then belong to one category: *graphicalContainer* for all widgets containing other widgets such as window, frame, dialog box, table, box and their related decomposition or *graphicalIndividualComponent* for all other traditional widgets that are typically found. UsiXML supports (Figure 2) *textComponent*, *videoComponent*, *imageComponent*, *imageZone*, *radioButton*, *toggleButton*, *icon*, *checkbox*, *item*, *comboBox*, *button*, *tree*, *menu*, *menuItem*, *drawingCanvas*, *colorPicker*, *hourPicker*, *datePicker*, *filePicker*, *progressionBar*, *slider*, and *cursor*.

Thanks to this progressive inheritance mechanism, every final elements of the CUI inherits from the upper properties depending on the category they belong to. The properties that have been chosen in UsiXML have been decided because they belong to the intersection of property sets of major toolkits and window managers, such as Windows GDI, Java AWT and Swing, HTML. Of course, only properties of high common interest were kept. In this way, a CIO can be specified independently from the fact that it will be further rendered in HTML, VRML or Java. This quality is often referred to as the property of **platform independence.** Therefore, the CIOs defined at the CUI level remain independent of any computing platform (and thus of any underlying toolkit) since the same CUI could be specified in principle for different computing platforms and devices.

In the next section, we will see how this Concrete User Interface can be sketched in SketchiXML and stored internally in terms of UsiXML tags.

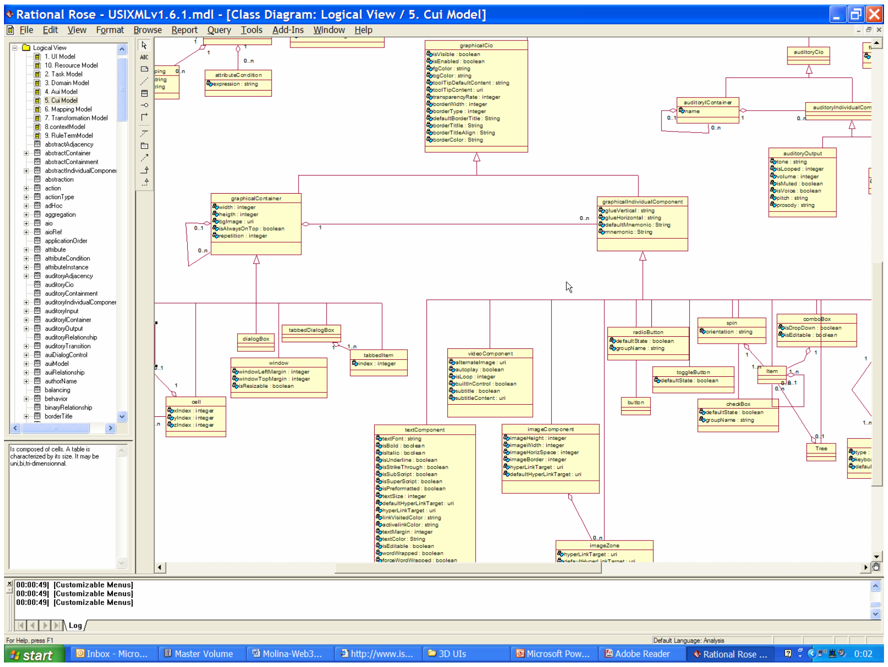


Fig. 2. The Concrete User Interface level defined in UsiXML as a UML Class Diagram [14].

4 SketchiXML Development

The main requirements to address are the following: to support shape recognition, to provide spatial shape interpretation, to provide usability advice at design time, to handle several kinds of input, to generate UsiXML specifications at design-time, and to operate in a flexible way. To address these requirements, a BDI (*Belief-Desire-Intention*) agent-oriented architecture [7] was considered appropriate: such architecture allows building robust and flexible applications by distributing the responsibilities among autonomous and cooperating agents. Each agent is in charge of a specific part of the process, and cooperates with the others in order to provide the service required according to the designer's preferences. This kind of approach has the advantage of being more flexible, modular and robust than traditional architecture including object-oriented ones [7].

4.1 SketchiXML Architecture

The application was built using the SKwyRL-framework (its usage is summarized in [7]), a framework aimed at defining, formalizing and applying socially based catalogues of styles and patterns to construct agent and multi-agent architectures. The joint-venture organizational style pattern was applied to design the agent-architecture of SketchiXML. It was chosen on basis of non-functional requirements R*i*, as among all organizational styles defined in the SKwyRL framework, the joint venture clearly matches the aforementioned requirements as the most open and distributed organizational style.

The architecture (Fig. 3) is structured using *i** [7], 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.

When a user wishes to create a new SketchiXML project, she contacts the *Broker* agent, which serves as an intermediary between the external actor and the organizational system. The *Broker* queries the user for all the relevant information needed for the process, such as the target platform, the input type, the intervention strategy of the *Adviser* agent,... According to the criteria entered, the coordinator chooses the most suitable handling and coordinates all the agents participating in the process in order to meet the objectives determined by the user. For clearness, the following section only considers a situation where the user has selected real time recognition, and pen-input device as input. So, the *Data Editor* agent then displays a white board allowing the user to draw his hand-sketch interface. All the strokes are collected and then transmitted to the *Shape Recognizer* agent for recognition. The recognition engine of this agent is based on the CALI library [4], a recognition engine able to identify shapes of different sizes, rotated at arbitrary angles, drawn with dashed, continuous strokes or overlapping lines.

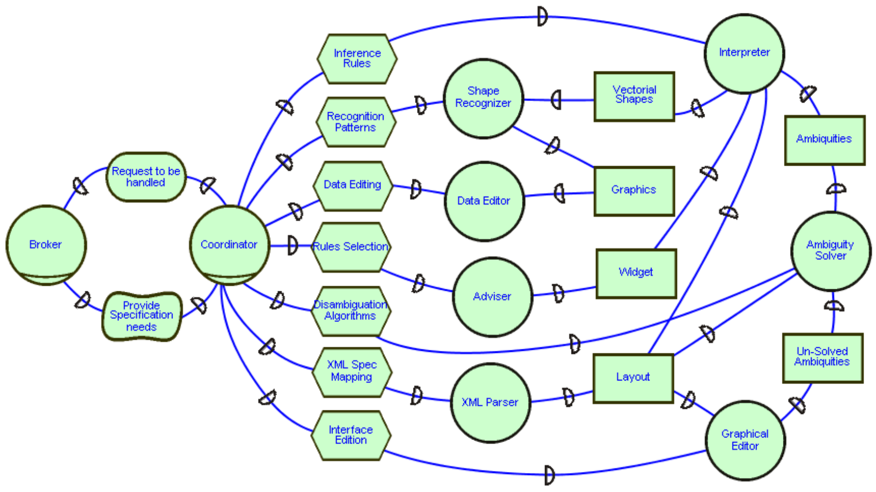


Fig. 3. i* representation of SketchiXML architecture as a Joint-Venture.

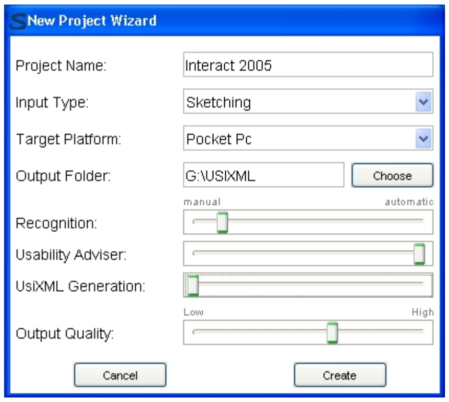
Subsequently, the *Shape Recognizer* agent provides all the vectorial shapes identified with relevant information such as location, dimension or degree of certainty associated to the *Interpreter* agent. Based on these shape sets, the *Interpreter* agent attempts to create a component layout. The technique used for the creation of this layout takes advantage of the knowledge capacity of agents. The agent stores all the shapes identified as his belief, and each time a new shape is received all the potential candidates for association are extracted. Using its set of patterns the agent then evaluates if shape pairs form a widget or a sub-widget. The conditions to be tested are based on a set of fuzzy spatial relations allowing to deal with imprecise spatial combinations of geometric shapes and to fluctuate with user preferences. Based on the widgets identified by the *Interpreter*, the *Adviser* agent assists the designer with the conception of the UIs in two different ways.

Firstly, by providing real-time assistance to the designer by attempting to detect UI patterns in the current sketch in order to complete the sketch automatically. Secondly in a post operational mode, the usability adviser provides usability advice on the interface sketched. If the *Interpreter* fails to identify all the components or to apply all the usability rules, then the *Ambiguity Solver* agent is invoked. This agent evaluates how to solve the problem according to the initial parameters entered by the user.

The agent can either attempt to solve the ambiguity itself by using its set of disambiguation algorithms, or to delegate it to a third agent, the *Graphical Editor* agent. The *Graphical Editor* displays all the widget recognized at this point, as classical element-based software, and highlights all the components with a low degree of certainty for confirmation. Once one of these last three agents evoked has sufficient certainty about the overall widget layout, the UI is sent to the *XML Parser* agent for UsiXML generation.

4.2 Low-Fidelity Prototyping with SketchiXML

The first step in SketchiXML consists of specifying parameters that will drive the low-fidelity prototyping process (Fig. 4): the project name, the input type (i.e. on-line sketching or off-line drawing that is scanned and processed in one step-Fig. 5), the computing platform for which the UI is prototyped (a predefined platform can be selected such as mobile phone, PDA, TabletPC, kiosk, ScreenPhone, laptop, desktop, wall screen, or a custom one can be defined in terms of platform model [10]), the output folder, the time when the recognition process is initiated (ranging from on-demand manual to fully automatic each time a new widget can be detected- this flexibility is vital according to experiments), the intervention mode of the usability advisor (manual, mixed-initiative, automatic), and the output quality stating the response time vs. quality of results of the recognition and usability advisor processes. In Fig. 7, the UsiXML parsing is set on fully manual mode, and the output quality is set on medium quality. The quality level affects the way the agents consider a widget layout to be acceptable, or the constraints used for the pattern matching between vectorial shapes. The sketching phase in that situation is thus very similar to the sketching process of an application such as FreeForms [17]. Of course, the designer is always free to change these parameters while the process is running.



Project Name:	Interact 2005
Input Type:	Sketching
Target Platform:	Pocket Pc
Output Folder:	G:\USIXML
Recognition:	
Usability Adviser:	
UsiXML Generation:	
Output Quality:	

Fig. 4. Creating a new SketchiXML prototype

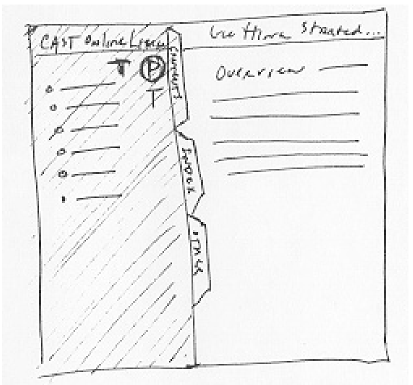


Fig. 5. Scanned UI sketching

Fig. 6 illustrates the SketchiXML workspace configured for designing a UI for a standard personal computer. On the left part we can observe that shape recognition is disabled as none of the sketches is interpreted, and the widget layout generated by the *Interpreter* agent remains empty. The right part represents the same UI with shape recognition and interpretation. Fig. 7 depicts SketchiXML parameterized for a PocketPC platform and its results imported in GrafiXML, a UsiXML-compliant graphical UI editor that can generate code for HTML, XHTML, and Java (http://www.usixml.org/index.php?view=page&idpage=10).

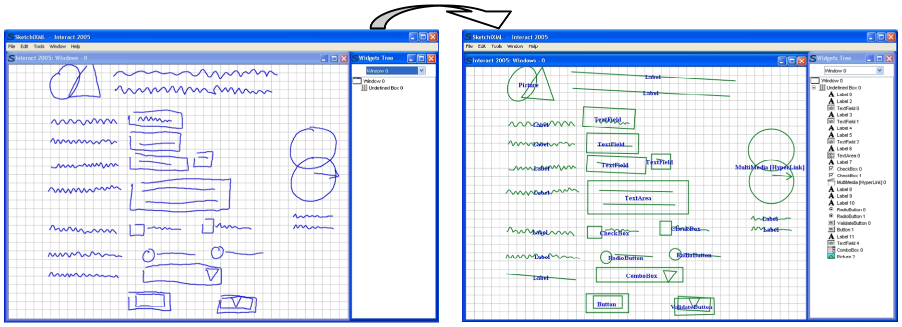


Fig. 6. SketchiXML workspace.

When shape recognition is activated, each time a new widget is identified the color of the shapes turns to green, and the widget tree generated by the *Interpreter* is updated. Changing the context has a deep impact on the way the system operates. As an example, when a user builds a user interface for one platform or another, adaptations need to be based on the design knowledge that will be used for evaluation, by selecting and prioritizing rule sets, and on the set of available widgets. As the size of the drawing area is changing, the set of constraints used for the interpretation needs to be tailored too, indeed if the average size of the strokes drawn is much smaller than on a standard display, the imprecision associated with each stroke follows the same trend. We can thus strengthen the constraints to avoid any confusion.

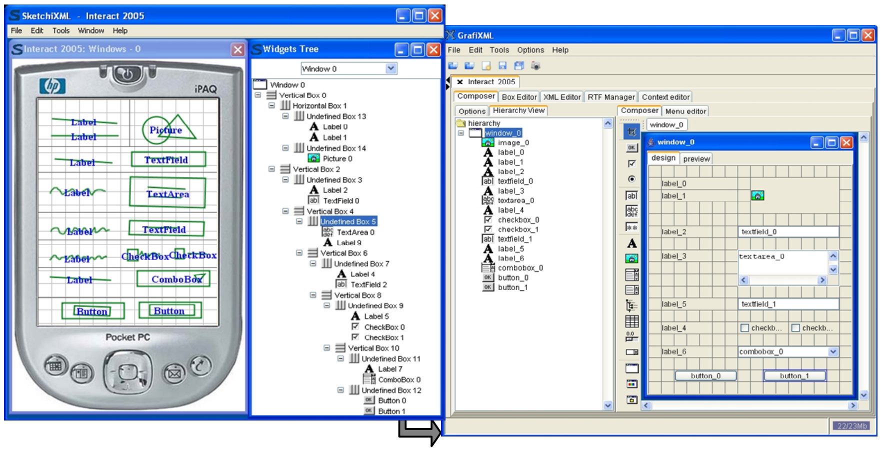
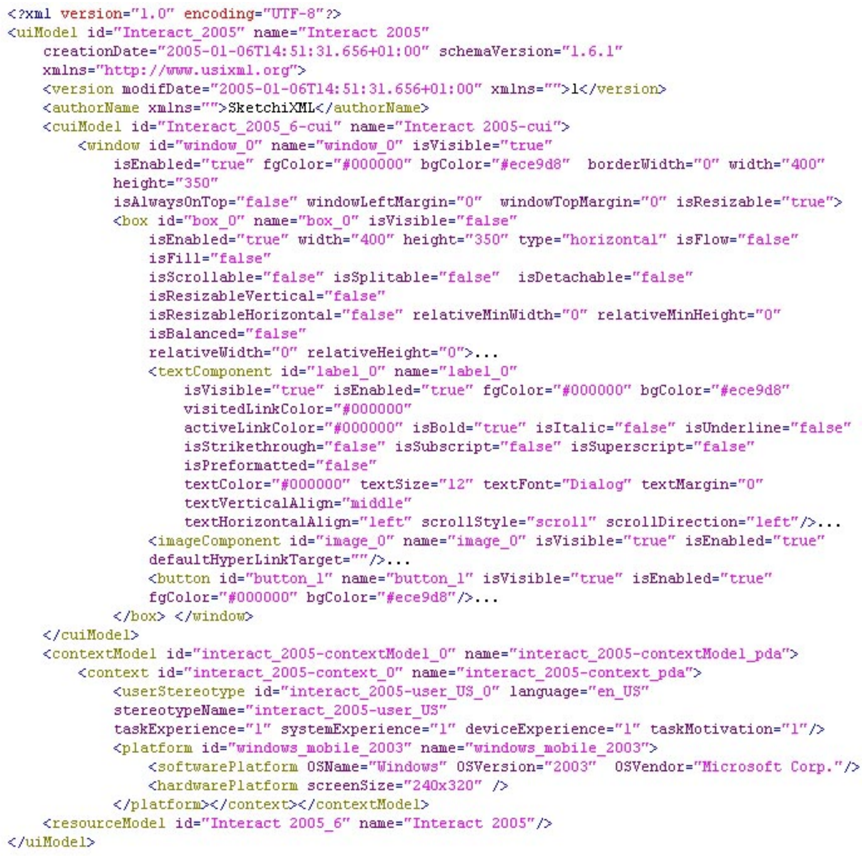


Fig. 7. SketchiXML workspace configured for a PDA and its import in GrafiXML.

Once the design phase is complete, SketchiXML parses the informal design to produce UsiXML specifications. Fig. 8 gives an overview of the UsiXML specifications generated from UI drawn in Fig. 7. Each widget is represented with standard values for each attribute, as SketchiXML is only aimed at capturing the UI core properties. In addition, the UsiXML specifications integrate all the information related to the context of use as specified in the wizard depicted on Fig. 7: information for the user model, the platform model, and the environment model [10]. As UsiXML allows defining a set of transformation rules for switching from one of the UsiXML models to another, or to adapt a model for another context, such information is thus required. Fig. 7 illustrates the SketchiXML output imported in GrafiXML, a high fidelity UI graphical editor. On basis of the informal design provided during the early design, a programmer can re-use the output without any loss of time to provide a revised version of the UI with all the characteristics that can and should not be defined during the early design phase. This contrasts with a traditional approach, where a programmer had to implement user interfaces on basis of a set of blackboard photographs or sheets of paper, and thus start the implementation process from the beginning.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<uiModel id="Interact_2005" name="Interact 2005"
    creationDate="2005-01-06T14:51:31.656+01:00" schemaVersion="1.6.1"
    xmlns="http://www.usixml.org">
    <version modifDate="2005-01-06T14:51:31.656+01:00" xmlns="">1</version>
    <authorName xmlns="">SketchiXML</authorName>
    <cuiModel id="Interact_2005_6-cui" name="Interact 2005-cui">
        <window id="window_0" name="window_0" isVisible="true"
            isEnabled="true" fgColor="#000000" bgColor="#ece9d8"  borderWidth="0" width="400"
            height="350"
            isAlwaysOnTop="false" windowLeftMargin="0"  windowTopMargin="0" isResizable="true">
            <box id="box_0" name="box_0" isVisible="false"
                isEnabled="true" width="400" height="350" type="horizontal" isFlow="false"
                isFill="false"
                isScrollable="false" isSplitable="false"  isDetachable="false"
                isResizableVertical="false"
                isResizableHorizontal="false" relativeMinWidth="0" relativeMinHeight="0"
                isBalanced="false"
                relativeWidth="0" relativeHeight="0">...
                <textComponent id="label_0" name="label_0"
                    isVisible="true" isEnabled="true" fgColor="#000000" bgColor="#ece9d8"
                    visitedLinkColor="#000000"
                    activeLinkColor="#000000" isBold="true" isItalic="false" isUnderline="false"
                    isStrikethrough="false" isSubscript="false" isSuperscript="false"
                    isPreformatted="false"
                    textColor="#000000" textSize="12" textFont="Dialog" textMargin="0"
                    textVerticalAlign="middle"
                    textHorizontalAlign="left" scrollStyle="scroll" scrollDirection="left"/>...
                <imageComponent id="image_0" name="image_0" isVisible="true" isEnabled="true"
                defaultHyperLinkTarget=""/>...
                <button id="button_1" name="button_1" isVisible="true" isEnabled="true"
                fgColor="#000000" bgColor="#ece9d8"/>...
            </box> </window>
    </cuiModel>
    <contextModel id="interact_2005-contextModel_0" name="interact_2005-contextModel_pda">
        <context id="interact_2005-context_0" name="interact_2005-context_pda">
            <userStereotype id="interact_2005-user_US_0" language="en_US"
            stereotypeName="interact_2005-user_US"
            taskExperience="1" systemExperience="1" deviceExperience="1" taskMotivation="1"/>
            <platform id="windows_mobile_2003" name="windows_mobile_2003">
                <softwarePlatform OSName="Windows" OSVersion="2003"  OSVendor="Microsoft Corp."/>
                <hardwarePlatform screenSize="240x320" />
            </platform></context></contextModel>
    <resourceModel id="Interact 2005_6" name="Interact 2005"/>
</uiModel>
```

Fig. 8. Excerpt of the UsiXML specifications generated by SketchiXML.

As the Usability Advisor intervention time has been specified as “automatic” (Fig. 4), each time a usability deviation is detected with respect to usability guidelines, a tool tip message is produced in context, attached to the widget on concern. For this purpose, a set of form-based usability guidelines have been encoded in GDL (Guideline Definition Language), a XML-compliant description of guidelines that can be directly related to UsiXML widgets.

5 Conclusion

The main difference between SketchiXML considered here as a tool for sketching the UI during the prototyping phase is that the effort done during this phase is not lost: it is automatically transformed into specifications written in UsiXML in order to pass them to other software which communicate by exchanging UsiXML files. It could be in particular a high-fidelity UI editor such as GrafiXML (as illustrated in Fig. 7) or any other UsiXML-compliant editor. Therefore, the current level of fidelity of the prototyped UIs may be increased by recuperating these specifications into another editor and continuing to refine their specifications until a final UI is reached. From this moment, any UsiXML-compliant rendering engine (such as a code generator or interpreter) could render the UI at run-time, even if this is during the prototyping phase [11,19].

It is obvious that at the beginning of the UI development life cycle, the UI requirements are not yet well done, especially if the UI concerns a new domain of activity, where little or no previous experience or history exists. For those cases where a substantive experience already exists, this prototyping phase may be reduced to reopening previously existing UI specifications and tailoring them to the new project. In both cases, the sketching tool is able to support designers, developers, or even end-users to refine their ideas until a final UI is obtained with consensus between the stakeholders

ACKNOWLEDGMENTS

We gratefully thank the support from of the Request research project under the umbrella of the WIST (Wallonie Information Société Technologies) program under convention n°031/5592 RW REQUEST) and from the SIMILAR network of excellence (The European research taskforce creating human-machine interfaces SIMILAR to human-human communication), supported by the 6th Framework Program of the European Commission, under contract FP6-IST1-2003-507609 (http://www.similar.cc). We warmly thank J.A. Jorge, F.M.G. Pereira and A. Caetano for allowing us to use JavaSketchIt and the CALI library in our research.

REFERENCES

1. Ali M.F., Pérez-Quiñones M.A., Abrams M.: *Building Multi-Platform User Interfaces with UIML*. In: Seffah, A., Javahery, H. (eds.): Multiple User Interfaces: Engineering and Application Framework. John Wiley, Chichester (2004) 95–118.
2. Barboni, E., Navarre, D., Palanque, Ph., Basnyat, S.: *Model-Based Engineering of Widgets, User Applications and Servers Compliant with ARINC 661 Specification*. In: Proceedings of the 13th Conference on Design, Specification, and Verification of Interactive Systems DSV-IS'2006 (Dublin, July 26-28, 2006). Lecture Notes in Computer Science, Springer Verlag, Berlin (2006).

3. Bastide, R., Navarre, D., Palanque, P.A.: *A Tool-supported Design Framework for Safety Critical Interactive Systems*. Interacting with Computers **15**,3 (2003) 309–328.
4. Caetano, A., Goulart, N., Fonseca, M., Jorge, J.: *JavaSketchIt: Issues in Sketching the Look of User Interfaces*. In: Proc. of the 2002 AAAI Spring Symposium - Sketch Understanding (Palo Alto, March 2002). AAAI Press, Menlo Park (2002) 9–14.
5. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: *A Unifying Reference Framework for Multi-Target User Interfaces*. Interacting with Computer **15**,3 (2003) 289–308.
6. Carr, D.A., *Specification of Interface Interaction Objects*. In: Proc. of ACM Conf. on Human Aspects in Computing Systems CHI'94 (Boston, April 24-28, 1994). Vol. 2, ACM Press, New York (1994) p. 226.
7. Coyette, A., Vanderdonckt, J.: *A Sketching Tool for Designing Anyuser, Anyplatform, Anywhere User Interfaces*. In: Proc. of 10th IFIP TC 13 Int. Conf. on Human-Computer Interaction INTERACT'2005 (Rome, 12-16 September 2005), Lecture Notes in Computer Science, Vol. 3585, Springer-Verlag, Berlin, 2005, 550–564.
8. Duke, D.J., Harrison, M.D.: *Abstract Interaction Objects*. Computer Graphics Forum **12**,3 (1993) 25–36.
9. Eisenstein, J., Vanderdonckt, J., Puerta, A.: *Model-Based User-Interface Development Techniques for Mobile Computing*. In: Lester J. (ed.): Proc. of 5th ACM Int. Conf. on Intelligent User Interfaces IUI'2001 (Santa Fe, January 14-17, 2001). ACM Press, New York (2001) 69–76.
10. Florins, M., Vanderdonckt, J.: *Graceful Degradation of User Interfaces as a Design Method for Multiplatform Systems*. In: Proc. of Int. Conf. on Intelligent User Interfaces IUI'04 (Funchal, January 13-16, 2004). ACM Press, New York (2004) 140–147.
11. Grolaux, D., Vanderdonckt, J., Van Roy, P.: *Attach me, Detach me, Assemble me like You Work*. In: Proc. of 10th IFIP TC 13 Int. Conf. on Human-Computer Interaction INTERACT'2005 (Rome, September 12-16, 2005), Lecture Notes in Computer Science, Vol. 3585, Springer-Verlag, Berlin (2005) 198–212.
12. Landay, J., Myers, B.A.: *Sketching Interfaces: Toward More Human Interface Design.* IEEE Computer 34, 3 (March 2001) 56–64.
13. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Lopez, V.: *UsiXML: a Language Supporting Multi-Path Development of User Interfaces*. In: Proc. of 9th IFIP Working Conf. 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). Lecture Notes in Computer Science, Vol. 3425. Springer-Verlag, Berlin (2005) 200–220.
14. Limbourg, Q., *Multi-path Development of User Interfaces*, Ph.D. thesis, Université catholique de Louvain, Louvain-la-Neuve, November 2004.
15. Newman, M.W., Lin, J., Hong, J.I., Landay, J.A.: *DENIM: An Informal Web Site Design Tool Inspired by Observations of Practice*. Human-Computer Interaction 18 (2003) 259–324.
16. Plimmer, B.E., Apperley, M.: *Software for Students to Sketch Interface Designs*. In: Proc. of 9th IFIP TC 13 Int. Conf. on Human-Computer Interaction INTERACT'2003 (Zurich, 1-5 September 2003). IOS Press, Amsterdam (2003) 73–80.
17. Plimmer, B.E., Apperley, M.: *Interacting with Sketched Interface Designs: An Evaluation Study*. In: Proc. of ACM Conf. on Human Aspects in Computing Systems CHI'04 (Vienna, April 24-29, 2004). ACM Press, New York (2004) 1337–1340.
18. Vanderdonckt, J., Bodart, F.: *Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection*. In: Proc. of the ACM Conf. on Human Factors in Computing Systems INTERCHI'93 (Amsterdam, April 24-29, 1993). ACM Press, New York (1993) 424–429.

19. Vanderdonckt, J.: *A MDA-Compliant Environment for Developing User Interfaces of Information Systems*. In: Pastor, O. & Falcão e Cunha, J. (eds.), Proc. of 17th Conf. on Advanced Information Systems Engineering CAiSE'05 (Porto, 13-17 June 2005). Lecture Notes in Computer Science, Vol. 3520. Springer-Verlag, Berlin (2005) 16–31.
20. van Duyne, D.K., Landay, J.A., Hong, J.I..: *The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience*. Addison-Wesley, Reading (2002).